Evolution of Low-Code Platforms

Michiel Overeem

SIKS Dissertation Series No. 2022-14
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-94-6458-189-8

Last update: 2022-07-29
Cover design: Mark van der Peet
Printing: Ridderprint | `www.ridderprint.nl`

Evolution of Low-Code Platforms

Evolutie van Low-Code Platformen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. H.R.B.M. Kummeling, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 15 juni 2022 des middags te 2.15 uur door

Michiel Overeem

geboren op 15 juni 1984, te Baarn

Promotoren:	Prof.dr. S. Brinkkemper
	Prof.dr.ir. H.A. Reijers
Copromotor:	Dr. R.L. Jansen

This research was partially accomplished with financial support from NWO in the AMUSE project [project code 628.006.001], a collaboration between Universiteit Utrecht, Vrije Universiteit Amsterdam, and AFAS Software, The Netherlands.

“Talk is cheap. Show me the code.”

Linus Torvalds

Preface

During my Master's degree, I never had the intention of pursuing a PhD. I valued industry with its practical outcome higher than the, what I thought to be, purely theoretical results of science. During the following years, however, I started to read more and more scientific literature. So when an opportunity presented itself to combine the job of an engineer and architect with scientific research, I decided to jump on it. This jump turned out to be a journey of six and half years, which by no means was an easy walk in the park.

At the time I did not know what I was getting into and, to be honest, I felt an imposter for 75% of the time. I learned many things while doing the scientific work presented in this dissertation, not only about conducting research but also about academic writing and selling ideas to peers. More than once a paper was rejected because the writing was not clear enough, I have learned the hard way. Although it was not always easy to receive peer review feedback, the article always turned out better. This might be the most important thing that I have learned about the way I work: I need feedback from peers to do my best work. Unfortunately submitting an article does not result in immediate feedback, making the process a bit slow. Luckily I received help from many people, without whom I never got this far. Although I may forget someone, I am obligated to give thanks where it is due.

This work would not have been in this state without the continuous support of Slinger Jansen. Thank you for your enthusiasm, constant encouragement, and sharing your experience with me throughout these years. I have learned so much about the academic world and conducting science through you. The way you always spot an opportunity is inspiring. Sjaak Brinkkemper and Hajo Reijers, you fulfilled your role in the background, but I enjoyed our collaboration. You are both experienced scientists willing to share your experience with a new generation, thank you.

I could not have done all of this without the opportunity given by Machiel de Graaf, Rolf de Jong, and Dennis van Velzen. Thank you for allowing me to combine the pursuit of a PhD with my job at AFAS. The flexibility, freedom, and support were everything I could wish for. My colleagues at AFAS did not always understand why anyone would do this for 'fun', but thank you for your interest, support, and patience. Special thanks to the AFAS Focus teams: Buccaneers, Orioles, Braves, Red-Sox, and Giants. This dissertation would not exist without all the work we did in the past years. Special thanks to Janine and Delina for proof reading my dissertation, it is so much better because of your feedback. Mark, a big thank you for creating the most wonderful cover.

Conducting scientific research is in no way a one-person job, but rather a community effort. During my research I have gotten support from many different people. The AMUSE group was an indispensable feedback group in the first years of my research. Dennis, Erik, Henk, Guru, Siamak, Ünal, and all the master students: thank you. Henk, thanks for the support and the occasional review after leaving AFAS and the AMUSE group! I am especially grateful to Marten, Max, and Sven for collaborating on our joint papers. Although I was never an active member of the *Organization and Information* group and the *Software Ecosystems* lab in Utrecht, I always received supportive feedback and interest. Much of my research is based on the experience and knowledge of industry experts and could not have been conducted without the cooperation of these experts. Thanks also to all the (anonymous) reviewers; without you, this work would not be what it is today. Most importantly, special thanks go to the reading committee, Jordi Cabot, Michel Chaudron, Gabrielle Keller, Jurgen Vinju, and Joost Visser for reviewing this dissertation.

I thank all my friends and family that have shown interest in the past years. I know it took some time, but its done now. I could not have completed this work without Mirjam, Ezra, and Silas. This dissertation marks the end of a period in which I have been busy in the evenings and weekends. Thank you for giving me the space and time to work on this project, but also for the distractions that you gave me when I needed it (although I could not always see that at the time). I love you with all my heart. Now I just have to find a new hobby ...

Michiel

Contents

Part I

Introduction

1

Introduction

The last decade has shown a rapid growth of the software industry. Statements such as “software is eating the world” [5] and “every company is now a software company” are frequently used in business articles to emphasize the importance of software for every company. To keep up with this growth, businesses are required to go through a transformation, a *digital* transformation. In order to meet this new digital world they will have to give automation through software solutions a central position in their vision and policies.

However, these companies face two challenges when they undergo this digital transformation. First, they have to attract IT professionals that are trained in software development. In doing so they will quickly learn that there is a lack of trained IT personnel [263] and that scarcity is only expected to increase in the coming years [22]. Although investments are made to train more IT personnel we should not expect the scarcity to be resolved any time soon. Secondly, the company needs to be organized in such a way that software development is done efficiently. Methods such as *Agile* and *DevOps* show that cooperation in cross-functional teams is crucial to decrease the time to market and increase the quality of software solutions. *Domain-Driven Design* [74] and the upcoming *BizDevOps* [82] emphasize the cooperation by bringing the business side closer to the process of software development.

A recent development, Low-Code Platforms (LCPs) (a term first used by Richardson & Rymer [206]), could be the solution to both these challenges [156]. Through the introduction of higher-level abstractions, such as domain-specific models or languages, these platforms enable *citizen developers* (professionals without training in software development) to develop software systems. The term *low-code* emphasizes that these platforms enable the development of software systems with a low effort of coding. Enabling untrained professionals to participate in the software development process not only means that fewer trained IT personnel are needed, but that the business side is also automatically more involved in the development of the software. Many of the ideas in LCPs appear to come from research domains such as Domain-Specific Languages [56, 143], and Model-Driven Engineering (MDE) [128]. Although both Bock & Frank [19] and Luo et al. [156] argue that there are no radical innovations in LCPs, Cabot [28] sees LCPs as an opportunity to bring existing knowledge and techniques under attention and use the new momentum to further the scientific knowledge. Ruscio et al. [213] do see differences between MDE and LCPs such as platform

(cloud versus desktop), users (citizen developers versus professional developers), and domain (business applications versus more technical domains).

In order for LCPs to enable citizen developers to develop increasingly large and complex software systems they need to incorporate approaches that enable the development of such large systems. Examples of these approaches are architectural patterns such as the microservice architecture style (MSA) and event-driven architectures. Both these patterns provide new ways of modularizing large systems. Abstractions that improve modularization are essential to manage the complexity of increasingly complex and large software systems [191]. The idea behind the MSA style is to develop a number of smaller (and thus less complex) interconnected software systems instead of a large monolithic system [121]. An event-driven architecture assures that different parts of a software system, such as microservices, communicate in an asynchronous manner [87, 141]. This results in a loosely coupled system in which the different microservices can be developed autonomously. Development teams can focus on a smaller part of the system, lowering the complexity of their tasks. These architectural patterns are already starting to find their way into the design of LCPs [202, 231], which enables citizen developers to create scalable and manageable solutions.

In this software-driven world, software applications can no longer be seen as isolated systems, by taking part in software ecosystems they are connected to the outside world [124]. Software ecosystems are used by Software Producing Organizations (SPOs) to increase the value of their software for their customers through collaboration with third parties. These ecosystems are formed around software platforms, which in turn are managed by software platform orchestrators [126]. Software platforms are a set of organizations collaboratively serving a market for software and services. Application Programming Interfaces (APIs) are indispensable in these software ecosystems. API management is, therefore, an crucial activity for these orchestrators. The third parties that extend the software systems rely on these APIs, making it essential for these APIs to be available and scalable.

We are convinced that LCPs that incorporate an event-driven MSA style, combined with a strong support of software ecosystem enabling capabilities such as API management, present a solution for the pressing challenges faced for companies that want to take part in the new software-driven world. However, the development of new software solutions through LCPs is not enough for these companies. As Visser [258] states: "Software is not built to last; it is built to change." Companies do not operate in a vacuum; they are part of a larger world inhabited by customers, suppliers, governments, and competitors. These outside forces cause requirements and demands to change to which the software needs to respond. In order to be successful, companies have to invest in their software solutions for the long haul, making sure that they meet the changing demands.

Software evolution is the activity of responding to these outside forces that result in changing requirements. Lehman's law of *continuing change* [151] is still valid: if software does not respond to these outside forces, it becomes less useful. Vinju [257] sees the ideas of MDE that underpin modern LCPs as a solution for the challenge of software evolution. The higher abstraction used in LCPs results in more precise specifications and requirements. These specifications and requirements can be used

to guide the software evolution. In the future, this could enable the development of trustworthy systems as discussed by Klein et al. [139].

LCP providers need to make sure that their platforms can support companies in their digital transformation, therefore, they need to meet the requirements of these companies that want to partake in the software-driven world. So what is or could be the role of the software architect in all of this? The architect's role is to lead the LCP through the changes that the platform most likely will face [232]. The platform has to apply modern architectural patterns, support the engineering of software ecosystems, and support the maintainability of applications for the long haul without the platform becoming increasingly complex and eventually collapsing under all its complexity. Software architects responsible for LCPs need to guide the evolution of the platform. Unfortunately for these architects, little actionable knowledge is available on software evolution in LCPs.

LCPs enable citizen developers to create business-critical software systems. These systems are increasingly large and complex and need to respond to a constant stream of outside forces that require change. This dissertation contributes to the body of knowledge available to software architects in three areas:

Event Sourced Systems and Evolution - Event-driven architectures enable the development of large and complex software systems. Event sourcing [84] and Command Query Responsibility Segregation (CQRS) [275] are two specific architectural patterns applied in MSA style and event-driven architectures. However, little knowledge and tooling is available for software architects that need to address software evolution.

API Management in Software Ecosystems - Software ecosystems enable the enrichment of software systems by external complementors. API management in software ecosystems is essential. If LCPs want to support the engineering of software ecosystems they need to offer API management capabilities. Therefore the software architects of LCPs need to plan and develop API management capabilities so that the LCPs can grow in maturity.

Evolution Supporting Architecture - Changes that are made to either the LCP or the applications on top of the LCP have an impact on other parts of the platform or even on complementors. Certain changes, for instance, require follow-up changes, such as data conversion. Software architects need tools that allow them to assess the impact of changes. These tools or techniques allow them to design the processes that give them control of the software evolution.

These three areas are central in this dissertation, illustrated in the simplified architecture of the low-code platform in Figure 1.1. It hosts event sourced applications that expose APIs to the complementors in a software ecosystem.

In the remainder of this chapter we introduce the central innovations of this dissertation, the research approach, and research questions.

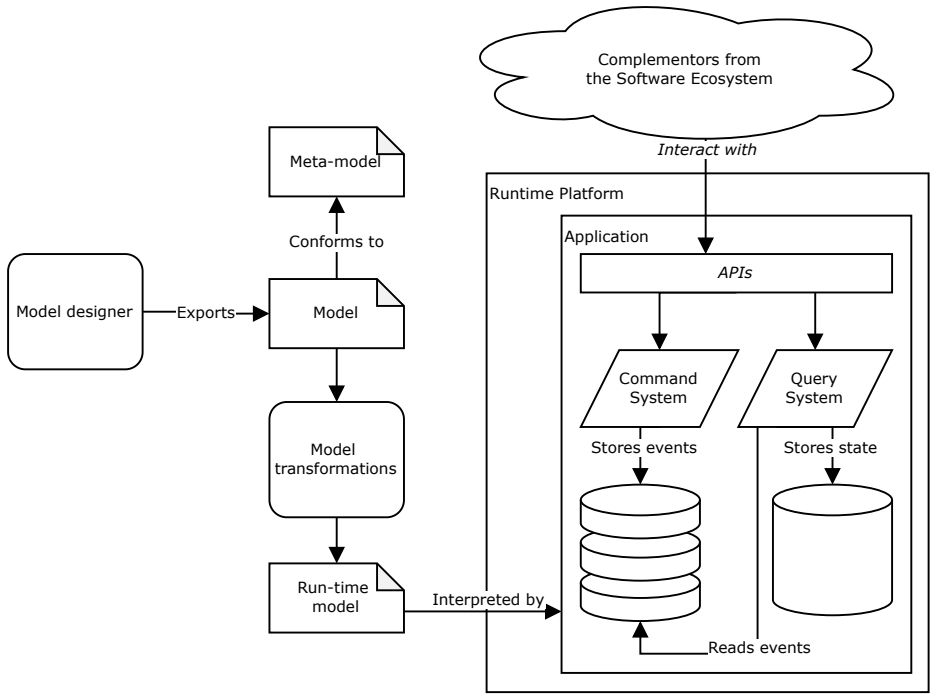

Figure 1.1: A low-code platform, hosting an event sourced application part of a software ecosystem. The model designer is used to create a model (conforming to a meta-model). The model is transformed into a run-time model interpreted by an event sourced application. The application is hosted on a run-time platform, which provides standardized capabilities such as load balancing and monitoring. The application publishes APIs that are used by complementors in a software ecosystem.

1.1 Innovations in Software Systems

Four central innovations are at the heart of this dissertation. These innovations present new challenges in software evolution that need to be dealt with by software architects. First, the evolution of the software industry is caused by trends such as Software-as-a-Service [132], platformization, and the common utilization of cloud infrastructures. A second innovation is the introduction of LCPs, which enable citizen developers to create advanced software solutions by raising the level of abstraction to domain concepts instead of software concepts. The third innovation is the introduction of so-called event-driven architectures. These architectures give events a central role, enabling a better match between the reality of business events versus system events. Finally, the fourth innovation is the connectivity of software systems through published APIs that enable third parties to access data and features. In this section we describe these four innovations and state per innovation the research challenge that our work focuses on.

1.1.1 Product Software and Software-as-a-Service

The software industry has gone through several major transitions in the last half-century. Initially, SPOs developed stacks of technologies on top of hardware platforms, typically as monolithic systems. However, soon operating systems [48] that could operate on multiple hardware platforms were introduced, and end-user applications were developed by third parties to run on top of these operating systems.

The internet era enabled SPOs to use a networked infrastructure of different services for offering their features online. The Software-as-a-Service model [35] was born, making it possible for SPOs to offer their products to a larger group of customers, generally at a lower cost of ownership. The customers no longer needed to own the hardware or other software systems, such as the operating system, to operate the software. With the introduction of Software-as-a-Service the SPO became not only responsible for the software itself, but also for operating the software. The demands for performant, reliable, and scalable software systems became ever more pressing, which is one of the major concerns that we address in this dissertation.

Increasingly, traditional SPOs, i.e., organizations whose main activities include the production of software, such as software vendors and open source organizations, are developing platforms as a vehicle to increase the value of their software for their customers through collaboration with third parties. This transformation from a product towards a platform is called 'platformization' [193]. Platforms are a vehicle for software ecosystems and are defined as a set of organizations collaboratively serving a market with software and services [126]. These ecosystems form around software platforms, which are managed by software platform orchestrators.

Research Challenge (RC1): Software systems are offered as a service, making SPOs responsible for operating the software. Techniques for reliable systems that continue operating through the deployment of new versions need to be designed and evaluated.

1.1.2 Low-Code Platforms

The first use of the term *low-code* is attributed to the Forrester report [206] on development platforms for customer-facing applications. LCPs are defined in the report as "Platforms that enable rapid application delivery with a minimum of hand-coding." The two essential characteristics hereof are the increase of software delivery and the decrease of the amount of coding required. The term *no-code* is closely related to low-code; in general it is regarded as a particular flavor of LCP.

LCPs introduce an abstraction that makes the development of software more efficient and the resulting software of higher quality. The Forrester report claims three benefits for LCPs: accelerated application delivery, quicker response to customer feedback, and support for multiple platforms. A recent study by Mendix [165] claims that LCPs result in 53% less costs, 56% more speed, and 58% more revenue. However, the work of Luo et al. [156] paints a more nuanced picture of the benefits of LCPs by also listing a number of liabilities. While initially these platforms have started with relatively simple applications that automate a single task, the applications targeted nowadays are becoming increasingly complex. Examples are enterprise services [281], Internet of Things applications [189], and enablers of digital transformations in the

manufacturing industry [219]. Sahay et al. [217] presents an elaborate comparison of different LCPs that shows that they target increasingly complex application domains and systems. Developers are invited to develop business critical systems that are connected to a landscape of other systems in so-called software ecosystems.

The work of Bock & Frank [18] shows that there is no new technology in LCPs, but they do integrate different technologies in a new way. The origin of LCPs can be traced back to model-driven engineering, as stated by Cabot [28]. He regards low-code as a synonym for model-driven engineering, while both Bock & Frank [18] and Luo et al. [156] see LCPs as a bundling of existing technologies such as model-driven engineering, database management systems, and dependency managers.

One of the technologies that is central in LCPs is model-driven development (MDD). The approach taken by LCPs is the model-centric approach [24]: a software system is derived automatically from a model. The term *Model-Driven Engineering Environments* (MDEE) as used in Chapter 6 should be interpreted as a synonym for LCPs.

A specific characteristic of LCPs is that they target the *citizen developer* [177]. These developers do not have training in software engineering but do possess specific domain knowledge in their field. LCPs enable these professionals to build applications that serve a specific need in their domain. Mendix, one of the leaders in the low code market, explicitly names the citizen developer as a solution to the growing shortage of professional software engineers [165]. This makes LCPs an important technology to invest in.

In their overview of Product Software, Xu & Brinkkemper [273] discuss the key differences between different categories of software. LCPs fall in the development tooling category. Well-known LCPs such as Mendix, OutSystems, and Pega are development tools that are licensed to many users. Applications built on top of these LCPs are generally tailor-made software systems. That is what LCPs do well: they make it more accessible for companies to develop software.

Research Challenge (RC2): LCPs are increasingly used to build business-critical systems and companies depend on the stability and reliability of the platform. Changes made to the platform and applications threaten these characteristics; the evolution of the platform and applications need to be controlled to mitigate these risks.

1.1.3 Event-Driven Architecture

The core idea of event-driven architectures is that the components in a software system communicate through events. These events can originate from within or outside systems, and components react to these events [32]. Event sourcing takes this idea of events and applies it to the state management in a system. Instead of storing the current state of a system, every change to the state is recorded as an event. The current state can be built by the systems as a derivation of the events. There can be multiple derivations that each present the state in a different manner. This increases the flexibility of the system.

Command query responsibility segregation (CQRS) originated together with event sourcing from the Domain-Driven Design (DDD) community. The foundations were laid by Meyer [169] in the Command-Query Separation (CQS) principle. He defined a *command* as “serving to modify objects” and a *query* as “to return information about ob-

jects," or informally worded: "asking a question should not change the answer." This principle is applied in the CQRS pattern: commands are accepted by the command-side; there, it produces events that are processed by the query-side to answer the queries it receives. Figure 1.1 illustrates this principle.

The command- and query-sides use their own data store and data model. To validate and accept commands, the command-side uses a data model optimized for that particular task. The query-side often uses a more versatile and diverse range of data models and stores to answer questions, such as a full-text index together with a relation database.

The changes accepted on the command-side are communicated to the query-side trough events. This communication is where CQRS and event sourcing are combined (although it is not strictly necessary to combine the two patterns). If the command-side uses event sourcing as a data model, the query-side can consume those recorded events. The query-side often processes these events asynchronously, creating a weak coupling between the two sides. This results in two independent and autonomous components. The benefit is that these components do not influence each other from a development and operations perspective, i.e. the performance of the command-side is independent of the query-side. However, this does lead to eventual consistency, which Vogels [261] defines as "when no updates are made to the object, the object will eventually have the last updated value." Eventually, the query-side will reflect the events produced by the command-side but there are no guarantees on how fast this will be done.

Eventual consistency is not the only challenge in event sourced systems. The most prominent challenge is the evolution of the data model. As every state change is recorded, the system should be able to read and process all of these events for eternity. When the software system can no longer read and process all of the events, the state can no longer be re-hydrated.

Research Challenge (RC3): Event Sourcing and CQRS are identified by industry as techniques for performant, reliable, and scalable software systems. However, the evolution of event sourced systems require new techniques and strategies that software architects can employ.

1.1.4 Software Ecosystems and APIs

As already mentioned, more and more SPOs are turning their software products into platforms. This enables them to create more value by leveraging the solutions developed by other organizations. To form software ecosystems, enterprises need to open up and provide access to their data through APIs [53, 266]. *APIs* are defined by De [53] as "a software-to-software interface that defines a contract for applications to communicate with one another over a network, without the need for any user interaction." For LCP providers it is advantageous to enable the design, development, and publishing of APIs. First of all, this will enable their customers to develop software platforms that other SPOs can enrich. Secondly, support for APIs also enables their customers to build more complex software systems by integrating several smaller software applications with each other. APIs enable the modularization of LCP applications, increasing the maintainability of these solutions.

Challenges in software ecosystems such as stability, security, and scalability are related to API management capabilities [6]. API management is the activity that enables organizations to design, publish, and deploy their APIs for (external) developers to consume. When SPOs transform their software systems from products to platforms, and offer them as a service, it becomes crucial to not only implement basic and standard platform features but also support advanced platform features. SPOSs must evaluate the maturity of their API management capabilities and plan improvements. The way in which they manage their APIs might turn out to be one of the determining factors for the success of their platform. Such an assessment should not only state how mature the API management capabilities are but it should also provide a roadmap for improvement.

Research Challenge (RC4): More and more SPOs turn their software products into platforms and, at the same time, allow external complementors to access their platforms. To effectively grow their system into an ecosystem, they need to be supported in the management of their integration capabilities.

1.2 Research Approach

With LCPs targeting increasingly more complex software applications and systems, new research challenges arise. Business-critical software systems model activities from the real world and are subject to constant change [151]. Constant change requires control over the evolution of these software systems. These complex applications function within a larger ecosystem and thus have to deal with integration and connectivity issues while undergoing change. Our research focuses on the evolution of LCPs and their applications.

The challenge that software evolution in LCPs presents to software architects is the topic of this dissertation. We conduct our research in the context of LCPs that support the development of cloud-based software ecosystems (see Figure 1.1). Our research on how evolution in an LCP can be managed is done in three parts. Chapters 2 and 3 discuss how event sourcing can be applied to create performant, reliable, and scalable software systems and how software architects can evolve these systems. The management of APIs within a software ecosystem is discussed in Chapter 4, while Chapter 5 discusses how maturing the API management capabilities of an LCP can enable citizen developers to grow software ecosystems. The last chapters, 6 and 7, present how software architects can create evolution supporting architectures for LCPs. We visualize the different parts in Figure 1.2.

1.2.1 Research Questions

As stated previously, the research presented in this dissertation is divided into three parts: **Event Sourced Systems and Evolution**, **API Management in Software Ecosystems**, and **Evolution Supporting Architecture**. The research questions that guided the research are divided into these three areas.

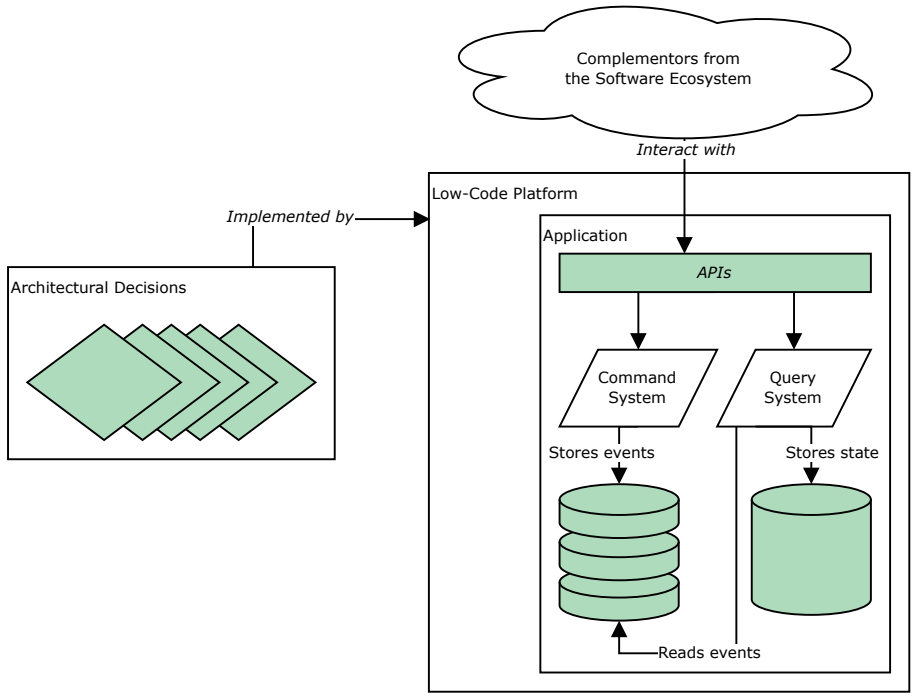

Figure 1.2: This dissertation studies the challenge of software evolution in LCPs and the applications developed with these platforms. We zoom in on three areas: (1) evolution in event stores and event sourced systems, (2) APIs published by applications, and (3) architectural decisions that support software evolution.

Event sourcing is a relatively new pattern and has not received much attention in existing research. However, software architects appear to identify the pattern as beneficial for large software systems. The main research question for this research area is

> ***MRQ1*** *- What are the challenges software architects face in the evolution of event sourced systems and how can they be mitigated?*

We will answer this question by answering four sub-research questions that focus on the pattern as perceived by industry and on existing techniques for evolution of event stores. These questions are answered in Chapters 2 and 3, where we discuss the pattern in general and the benefits and challenges that are experienced.

> ***SRQ1.1*** *- What types of systems apply event sourcing and why?*
>
> ***SRQ1.2*** *- How should event sourced systems be defined?*
>
> ***SRQ1.3*** *- How should event sourced data structures be evolved?*
>
> ***SRQ1.4*** *- What are the challenges faced by software engineers in applying event sourcing?*

Collaborating systems within a software ecosystem, or microservices within an MSA system, use APIs to communicate. API management is an essential activity in these systems because the quality of communication is largely dependent on the quality of the APIs. Evolution introduces new challenges as the different parties communicating undergo different changes at different speeds. Our focus is directed at the support of API management activities offered by LCPs to their customers, the citizen developers.

> ***MRQ2*** *- What kind of support for API management practices is offered by LCPs, and how should they evaluate and improve that support?*

This question is answered by two separate sub-questions. First, we look at the evaluation and improvement of API management practices in general. The results of that research are applied to API management support in LCPs. These questions are answered by the research discussed in Chapters 4 and 5.

> ***SRQ2.1*** *- How should SPOs that expose their APIs to third parties evaluate their API management practices?*
>
> ***SRQ2.2*** *- How mature are the API management capabilities that LCPs offer to their customers?*

Software architecture is understood as the composition of a set of architectural design decisions [122]. These decisions influence many operational characteristics of a software system, such as reliability and scalability. However, they also influence the manageability of a system as it undergoes evolution.

> ***MRQ3*** *- How should the architecture of an LCP support the evolution of both the platform as well as the applications?*

In the last two chapters, we discuss how software architectures can support evolution. The first question is an example of how SPOs can make informed decisions to achieve the quality characteristics that are demanded. As explained above, LCPs combine different existing technologies, one of them being MDE. In MDE, a model drives the software, i.e. a model is executed to produce the running software. Different approaches to this execution are possible, with different quality characteristics. We conclude the dissertation with research on the overall process of change impact analysis and its role in LCPs. These questions are answered in Chapters 6 and 7:

> ***SRQ3.1*** *- How should SPOs make an informed decision between a generative or interpretive model execution approach?*
>
> ***SRQ3.2*** *- What is the role of change impact analysis in an LCP?*

1.2.2 Research Methods Applied in this Thesis

Software engineering is a multidisciplinary field that crosses many social and technological boundaries [66]. To effectively study software evolution in LCPs different research methods are applied. While each chapter details the specifics of the method used, we summarize the different methods in this section, and in Table 1.1 we present the relation between the chapters of this dissertation, the research questions, and the research methods applied.

Ch.	Research Questions	Research Methods				
		Design Science	*Case Study*	*Expert Interview*	*Systematic Literature Review*	*Grounded Theory*
2	1.3	✓		✓		
3	1.1, 1.2, 1.3, 1.4			✓		✓
4	2.1	✓		✓	✓	
5	2.2		✓	✓		
6	3.1	✓	✓	✓	✓	
7	3.2	✓	✓			

Table 1.1: The mapping between the research questions and research methods that have been employed in each chapter (Ch.).

Design Science

In *Design Science Research* artifacts are built to solve problems in information systems [267]. These artifacts have different forms, such as methods, algorithms, and software systems. Proposed questions in design science research are answered through the creation and evaluation of artifacts. Design science research proposes solutions for design problems and seeks answers to knowledge problems. The acquired knowledge should contribute to the body of scientific evidence.

According to Hevner & Chatterjee [106] design science research has two interpretations: 'design as research' and 'researching design.' Ralph & Wand [205] label these the science of design (in which design is the topic of inquiry) and design science. Design as research is the idea that scientific knowledge is gathered through innovative design. The proposed artifact attempts to solve a problem, but in the attempt knowledge is accumulated. The other interpretation studies design, designers, and the process of design. Using the label 'design science research' for two different interpretations could be problematic. Therefore Ralph [203] proposes the label 'engineering research' to identify the design as research interpretation. This label is also used in the proposed *Empirical Standards for Software Engineering Research* [204].

Design science research is an established paradigm in the field of information systems [107]. Engström et al. [69] illustrate how the design science paradigm can also be useful to showcase the contributions of software engineering research. Software engineering research is, similar to design science research, solution oriented and provides design knowledge.

In this dissertation we apply design science most prominently in Chapter 4. A maturity model is constructed to communicate knowledge of practices and processes in a particular domain [14]. A focus area maturity model consists of a sequence of maturity levels that represents an anticipated, desired, or typical evolution path for an organization [235, 236]. To establish an organization's degree of maturity in the functional domain, the capabilities in the model are assessed through a set of questions. When the current maturity is known, the organization can be guided towards incremental development of the domain. The API-m-FAMM attempts to solve the problem of measuring and developing API management maturity. The iterative approach taken (through a systematic literature review, expert interviews, and case studies) shows how the design of the artifact emerged.

Case Study

According to Yin [274], case studies are useful when you want to understand a real-world case, and such an understanding is likely to involve an important context. Unlike experiments, case studies take the full context into account in order to understand contemporary phenomena. Case studies come in different lengths (brief or longitudinal), but all present a detailed account of a phenomenon at a site.

Ralph et al. [204] identify different types of case studies, of which *exploratory* and *evaluative* case studies are the most relevant for this dissertation. Case studies are a valuable tool for studying a phenomenon or an artifact in the context. Chapters 4, 5, 6, and 7 apply this method.

Expert Interviews

Expert interviews are an essential technique for qualitative research. In this dissertation, expert interviews are used for both artifact construction and evaluation (Chapters 2 and 4), as well as general knowledge acquisition (Chapter 3).

The experts interviewed in Chapters 2, 4, and 6 were selected through purposive sampling [73]. This non-randomized sampling technique refers to the deliberate choice of a participant due to the qualities the participant possesses. In both cases the interviewees were experts in, respectively, event sourcing, API management, and model-driven development.

For the exploratory research conducted in Chapter 3 a randomized set of experts was required. Therefore, we invited volunteers through different channels, such as Google Groups and Slack channels. We identified these people based on our experience in the event sourcing community. In addition to these invitations we also contacted a number of well-known experts. Finally, we executed 'interview snowballing': we asked each interviewee for further expert references.

None of the interviewees were compensated for their cooperation. Each interview was recorded and reviewed by a minimum of two authors. The interviews held for Chapter 3 were fully transcribed and made available [188].

Our experience in conducting interviews matches that of Elisabeth Hove & Anda [67]: it is time-consuming. However, we found that the experts were more than willing to take the time and share their information with us.

Systematic Literature Review

We conducted two systematic literature reviews which are presented in this dissertation. These reviews summarize and synthesize knowledge from a prior body of research. In Chapter 4 we applied the guidelines from Okoli [176] and Kitchenham & Charters [138] to design and populate an early version of the API-m-FAMM, our maturity model for API management. We collected the literature by searching scientific libraries for relevant keywords, and further narrowed down the results by applying inclusion and exclusion criteria. The review process and the results are made available [158].

For the research in Chapter 6 we applied the snowballing approach as described by Wohlin [270]. In this case keywords were difficult to construct because of the broad range of topics. Instead we selected an initial set of papers relevant to our study and followed both forward and backward references.

Grounded Theory

Adolph, Hall & Kruchten [2] explain how Grounded Theory (GT) as a method is useful for research in areas that have not been studied before. GT is a methodology that is used to induce theories from empirically collected data (for instance, through interviews or case studies). Constructivist GT [33] assumes that neither data nor theories are discovered but are constructed by the researchers out of the interactions with the field and its participants.

We present our application of GT in Chapter 3, in which we explore event sourced systems and improve our understanding of the pattern. Our research has an exploratory nature and, therefore, GT is a useful approach for this research. We knew we would find a description of the pattern, but were not aware what other concepts, challenges, and motivations would be identified.

1.3 Relevance and Empirical Evidence

In this dissertation we build knowledge that furthers the research on software evolution in LCPs, as well as practical tools that support the SPOs that develop these platforms.

Similar to Bider, Johannesson & Perjons [16] we believe that empirical research and design science research provide a suitable combination to balance these two goals. Research that only focuses on knowledge runs the risk on being irrelevant for both other researchers and practitioners as well. However, research that only focuses on practical tools might end without any scientific contribution at all.

This dissertation contributes to the scientific body of knowledge on LCPs and information systems in general by answering the research questions as stated in Section 1.2.1. Chapters 2 and 3 presents empirically gathered knowledge on event sourced systems based on interviews with 25 industry experts. The API-m-FAMM model in Chapter 4 presents a compiled body of knowledge on API management capabilities. This knowledge is based on a synthesis of existing literature and interviews with industry experts. Chapter 6 compiles knowledge from existing literature into a decision support model that explains how different model execution approaches influence the quality characteristics of a software system. Finally, the proposed framework from Chapter 7 is based on evidence empirically gathered during the development of an LCP.

Not only do we contribute to scientific knowledge, we also contribute to the development of research methods. In Chapter 3 we show how Grounded Theory can be used to describe a software design pattern and uncover its benefits and liabilities. Existing methods for the development of maturity models are extended in Chapter 4 where we show how design science methods can be used to develop and evaluate the constructed maturity model. The use of a *do-it-yourself* kit shows how the implementation of maturity models in industry can be improved.

Next to advancing the scientific knowledge and methods, we also aim to produce artifacts that advances the industry. The framework for event schema evolution from Chapter 2, the API-m-FAMM from Chapter 4, the decision support framework for

model execution (Chapter 6), and the proposal for an impact analysis framework in Chapter 7; all these artifacts communicate scientific knowledge. In line with the challenges stated by Wohlin [269], the exchange of knowledge should be the focus of academia and industry collaboration.

These artifacts not only compile and communicate knowledge, they are also directly actionable. The framework for event schema evolution is utilized in *SB+*; an event sourced ERP system developed on top of an LCP (more details follow in the remainder of this chapter). Chapter 6 details a case study in which the decision support framework for model execution is applied during the development of an LCP. The API-m-FAMM demonstrates how SPOs can improve their API management maturity, and experts indicate that the model appears to be easy to use, useful, and effective. These artifacts communicate the benefits in an actionable manner, adhering to the best practices listed by Garousi, Petersen & Ozkan [92].

1.3.1 Partnership between Academia and Industry

The research carried out for this dissertation was executed within the AMUSE research project. This research project was an academic collaboration between the Universiteit Utrecht and Vrije Universiteit Amsterdam, and the industry partner AFAS Software. The goal was to address software composition, configuration, deployment, and monitoring on heterogeneous cloud ecosystems through ontological enterprise modeling. The questions, problems, and challenges originated from the development of the new ERP system that AFAS Software is developing. In addition to the real-world challenges that were encountered while developing the ERP system, we were also aware of an extra tension in our research between industry relevance and scientific novelty. Although academia should be aware of this tension, we strongly believe in the importance of partnership between academia and industry.

Kruchten et al. [146] argue that research that is aware of the context (often industry, real-world context) leads to scientific contributions that have impact on industry and society. Research that lacks this context is at risk of leading to contributions that are not adaptable and usable. This in turn could harm the impact that the research has on society. In this dissertation, the research done on event sourcing [186, 187] is an example of context aware research. Event schema evolution was a real-world challenge experienced not only by AFAS Software, but through the community as a whole. Our initial research [186] led to invitations to speak at user groups and conferences (See the listing on page 212 for a complete list). These talks led to valuable interactions that we used in our follow-up research [187].

Another aspect of the partnership between academia and industry is the communication of research results. In our experience, scientific knowledge is often scattered among many articles or hidden within theoretical and abstract contributions. In our work on API management [185], model execution approaches [182], and impact analysis [181] we aimed to synthesize the already available knowledge in a form that serves industry. Artifacts as decision support frameworks and maturity models are well suited to communicate these kinds of results. Not only do they categorize and present the knowledge in a clear format, they also provide actionable information to industry. This dissertation's contribution hopefully furthers the partnership between

academia and industry as an example of actionable research results.

1.3.2 AFAS Software

This dissertation would not have been possible without the support of my employer, AFAS Software, which provided significant amounts of knowledge about the process of developing evolvable and scalable software. This Dutch SPO, a privately held company, is based in Leusden, the Netherlands (with additional offices in Belgium, Curaçao, and Aruba). It currently employs over 500 people, and generated 191 million of revenue last year (2020). AFAS' main software product is called *Profit*, which is an ERP system consisting of different modules such as Taxes, Finance, HRM, Order Management, Payroll, and CRM. This product has over 2 million users across 11.000 organizations of all sizes, ranging from companies with a single employee to companies with thousands of employees.

After 25 years, AFAS recently launched a new version of its ERP system, which is called *SB+* (which stands for *small business plus*). The development of *SB+* was started in 2010 as an internal research project with the goal of exploration. This exploration evolved into the development of an internal LCP, called *AFAS Focus*. This platform is the context of the research carried out in this dissertation as well as the AMUSE project. *AFAS Focus* uses an ontological enterprise model [228] (the platform was formerly called NEXT). The system is cloud-based and its architecture applies event sourcing and Command-Query Responsibility Segregation (CQRS) to satisfy quality characteristics such as availability and responsibility.

At the start of 2020, after nine years of development, the first customers started using *SB+* for their day-to-day accounting. In 2021 the product was launched to the public[1]. By the end of 2021 the initial group of companies had expanded to over two hundred companies using *SB+*. In the years to come, this number will rapidly increase into the thousands.

1.4 Dissertation Outline

The chapters in this dissertation are formed by independently published articles. These articles form a portfolio of our research on software evolution in LCPs. This section gives a brief overview of the chapters and their contribution and explain how the original article came into being. As explained in the previous sections, the six main chapters are grouped into three parts and every part discusses a specific evolution topic. Chapters 2 and 3 discuss the topic of data evolution in event sourced systems. The evolution and management of APIs in software ecosystems are discussed in Chapters 4 and 5. Finally, Chapters 6 and 7 discuss the architecture and processes around software evolution.

Chapter 1 - Introduction

The first chapter starts with an introduction of the main topics of this dissertation. It further states the research questions, explains the research methods, and discusses the relevance of this research.

[1]The (Dutch only) product website can be visited: `https://kleinzakelijk.afas.nl/`.

Chapter 2 - The Dark Side of Event Sourcing: Managing Data Conversion

Event searching is a relatively new architectural and data modeling pattern. Historically there had been no research on event sourcing that discussed how to approach event schema evolution. Chapter 2 discusses several techniques found in literature and links them to event schema operations and upgrade strategies. We formulated a framework for event schema evolution and validated that with three experts.

This chapter was originally published as a conference article [186] and is based on a research project collaboratively performed with Marten Spoor, who did an internship at AFAS Software.

Chapter 3 - An Empirical Characterization of Event Sourced Systems

In this study we address the lack of event sourcing knowledge in scientific literature by presenting the results of interviews with 25 event sourcing practitioners. We applied Grounded Theory to extract a detailed pattern description. This description discusses rationale for applying event sourcing, characteristics of 19 event sourced systems, the relation to other patterns, five challenges that practitioners experience, and finally an overview of event schema evolution techniques.

The chapter was originally published as a journal article [187]. The first author conducted all of the interviews, while the transcription was done by the first and third author. Coding and categorization was done by the first, second, and third author. The transcriptions, along with our codes and categories, were anonymized and made publicly available [188].

Chapter 4 - API-m-FAMM: a Focus Area Maturity Model for API Management

In Chapter 4 we present the results of our study on API management and its relevance to software ecosystems. To further the scientific knowledge we formalize a Focus Area Maturity Model on API management: the API-m-FAMM. This model synthesizes available knowledge on API management from both scientific and grey literature, and adds new knowledge that we gathered through expert interviews. Through a do-it-yourself assessment kit we evaluate the usability and relevance of the model for practitioners.

We applied the method for developing Focus Area Maturity Models as presented by Steenbergen et al. [235, 236], and described the different iterations and the relevant source data in detail. This chapter is published as a journal article [185], and both the results of the literature review as well as the details of the different intermediate versions are publicly available [158, 159]. This work is based on a research project collaboratively performed with Max Mathijssen, who did an internship at AFAS Software.

Chapter 5 - API Management Maturity of LCDPs

In Chapter 5 we apply the API-m-FAMM to assess LCPs. We selected four different platforms based on the list of LCPs from Vincent et al. [256]. We assess their maturity in API management through case studies based on interviews and available documentation. Based on the evaluations we discuss the importance of API management capabilities for LCPs in the enabling of the development of software ecosystems.

This chapter was originally published as a conference article [183], and the evaluation data was made publicly available [184].

Chapter 6 - Generative versus Interpretive MDD: Moving Past 'It Depends'

Chapter 6 is an example of a study that aims to bring scientific knowledge closer to practitioners. While model-driven development is a well-established research topic, the scientific articles are not always aimed at supporting platform developers. In this chapter we synthesize existing literature on model-driven development into a decision support framework for model execution approaches. This framework supports LCP developers in the design of their platform by presenting scientific knowledge in an applicable form.

The chapter was originally published [182] as an extension of a conference article [180]. The survey of software producing organizations is based on the thesis work of Sven Fortuin.

Chapter 7 - Proposing a Framework for Impact Analysis for LDCPs

Chapter 7 proposes an impact analysis framework, based on observations made during the development of a low-code framework. Applications developed with an LCP are impacted by many different components and artifacts. The proposed framework solidifies this impact in a framework to enable reasoning and analysis. Through a case study we show how this framework can support practitioners while simultaneously serving as a carrier for available scientific knowledge.

This chapter was originally published as a workshop paper [181].

Chapter 8 - Conclusion

In the final chapter we summarize our contributions, answer the research questions posed in Section 1.2.1, and discuss future work.

Part II

Event Sourced Systems and Evolution

2

The Dark Side of Event Sourcing: Managing Data Conversion

Evolving software systems include data schema changes, and because of those schema changes data has to be converted. Converting data between two different schemas while continuing the operation of the system is a challenge when that system is expected to be always available. Data conversion in event sourced systems introduces new challenges, because of the relative novelty of the event sourcing architectural pattern, because of the lack of standardized tools for data conversion, and because of the large amount of data that is stored in typical event stores. This paper addresses the challenge of schema evolution and the resulting data conversion for event sourced systems. First of all a set of event store upgrade operations is proposed that can be used to convert data between two versions of a data schema. Second, a set of techniques and strategies that execute the data conversion while continuing the operation of the system is discussed. The final contribution is an event store upgrade framework that identifies which techniques and strategies can be combined to execute the event store upgrade operations while continuing operation of the system. Two utilizations of the framework are given, the first being decision support for the upfront design of an upgrade system for event sourced systems, the second being a framework for an automated upgrade system that can be used for continuous deployment. The event store upgrade framework is evaluated in interviews with three renowned experts in the domain and has been found to be a comprehensive overview that can be utilized in the design and implementation of an upgrade system. The automated upgrade system has been implemented partially and applied in experiments.

This work was originally published in *Proceedings of the 24th Conference on Software Analysis, Evolution, and Reengineering (SANER 2017)*, titled 'The Dark Side of Event Sourcing: Managing Data Conversion'. It was co-authored by Marten Spoor and Slinger Jansen.

2.1 Introduction

Applications that do not evolve in response to changing requirements or changing technology become less useful, as Lehman [152] in his law of *continuing change* stated many years ago. Neamtiu & Dumitraş [173] show that this is a reality for modern cloud systems as many of them update more than once a week. Chen [34] describes how they applied continuous delivery on multiple projects to achieve a shorter time to market, and an improved productivity and efficiency. Several technical challenges including seamless upgrades are identified by Claps, Berntsson Svensson & Aurum [37]. The fast pace of evolution and deployment of cloud systems conflicts with the requirement to always be available and support uninterrupted work. For modern cloud systems to support the fast pace of evolution, upgrade strategies that are fast, efficient, and seamless have to be designed and implemented.

One of the architectural patterns that in recent years emerged in the development of cloud systems is Command Query Responsibility Segregation (CQRS). The pattern was introduced by Young [275] and Dahan [50], and the goal of the pattern is to handle actions that change data (commands) and requests that ask for data (queries) in different parts of the system. By separating the command-side (the part that validates and accepts changes) from the query-side (the part that answers queries), the system can optimize the two parts for their very different tasks.

Young [276] describes CQRS as a stepping stone for event sourcing. Event sourcing is a data storage model that does not store the current (or last) state, but all changes leading up to the current state. Fowler [84] explains event sourcing by comparing it to an audit trail: every data change is stored without removing or changing earlier events. The events stored in an event store are stored as schema-less data, because the different events often do not share properties. A store with an explicit schema would make it more difficult to append events in the store to a single stream. Data in schema-less stores is not without schema, but the schema is implicit: the application assumes a certain schema. This makes the problem of schema evolution and data conversion more difficult as observed by Scherzinger, Klettke & Störl [225]. Schema-less data is more difficult to evolve as the store is unaware of structure and thus cannot offer tools to transform the data into a new structure. Relational data stores that have explicit knowledge of the structure of the data can use the standardized *data definition language* (DDL) to upgrade the schema and convert the data. Another problem in the evolution of event sourced systems is the amount of data that is stored. Not only the current state, but also every change leading up to that state is stored in the system. This huge amount of data makes the problem of performing a seamless upgrade even more important: upgrades may need more time, but they are required to be imperceptible.

The frequency of schema changes is researched by Qiu, Li & Su [201]. Although the storage model is different and the architectural pattern is relatively new there is no indication that (implicit) schema changes in event sourcing are less of a challenge. Recovery of the implicit schema does not solve the problem for event stores, it only helps to find the right operations to transform into a new schema.

This paper answers the question *“How can an event sourced system be upgraded effi-*

ciently when the (implicit) event schema changes?" This question is answered by defining event store upgrade operations that can be used to express the data conversion executed by the upgrade of an event store in Section 2.4. Existing techniques that are capable of execution these operations to convert the events are discussed in Section 2.5. The efficiency of these techniques is judged on the basis of four quality attributes: functional suitability, maintainability, performance efficiency, and reliability. In Section 2.6 the deployment strategies, categorized by application and data upgrade strategies are discussed that lead to an upgrade system with zero downtime. The final framework that describes how to design and implement either an ad-hoc upgrade strategy, or a fully automated upgrade system, is proposed in Section 2.7. The final framework is evaluated by three Dutch experts in the field of event sourcing, each having six or more years of experience in building and maintaining event sourced systems, and these results can be found in Section 2.8. Section 2.9 summarizes the contributions and states future work.

2.2 Command Query Responsibility Segregation

The foundations of CQRS were laid by Meyer [169] in the Command-Query Separation (CQS) principle. He defined a command as "serving to modify objects" and a query is "to return information about objects," or informally worded as "asking a question should not change the answer." Figure 2.1 shows the CQRS pattern: commands are accepted by the command-side and produce events which are processed by the query-side. The query-side projects these events into a form that is suitable for querying and presenting. The command-side and the query-side both have their own data store: the first store is used to maintain data that is used in validating requested changes, and the second store is used to retrieve data for displaying or reporting.

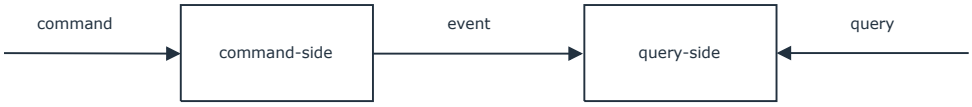

Figure 2.1: The architectural pattern CQRS.

The command-side communicates with the query-side by asynchronously sending events. These events are used by the query-side to build a view of the state that can be used to query and present data. By doing this asynchronously the query-side does not influence the performance of the command-side. However, this does result in eventual consistency. This is a weaker form of consistency that Vogels [261] defines as "when no updates are made to the object, the object will eventually have the last updated value." The system guarantees that the query-side eventually will reflect the events produced in the command-side. However, there are no guarantees on how fast this will be done. A system with a large delay is unfeasible, because in that case queries will often return data that does not reflect the latest changes sent to the command-side. There are difficulties introduced by eventual consistency, such as returning items to a client that are in fact already deleted through commands sent to the command-side.

The patterns to overcome this difficulty and others are out of scope for the current paper.

The asynchronous sending of events between the command-side and query-side results in a weak coupling. The resulting freedom and flexibility in designing the system lead to availability, scalability, and performance among other advantages. The store used in the command-side is often an event store, because it is natural to store the events that are produced by the command-side. This proposed data storage model has a number of benefits that make it especially useful as a store for the command-side of a CQRS system. First of all, the command-side is only used for accepting changes and never for queries, and the performance of the store is thus not hampered by concurring reads and writes. Second, the store contains every change ever accepted into the system, making it easy to inspect when and by whom a change was done. A third benefit is the possibility to rebuild the current state (for instance the query-store) in the system by replaying the events. The replaying of events also enables easy debugging. The fourth benefit is the possibility to analyze the events for patterns in usage. This information is impossible to extract from a store that only persists the last state of the data. In the query-side a diverse range of stores can be used, such as relational, graph, or NoSql databases. The main goal of this store is to support the easy and fast retrieval of data, in whatever form the application requires.

The loosely coupled nature of CQRS combined with the benefits of the event sourcing approach makes it a fitting architectural pattern for cloud systems. Event sourcing itself is not tied exclusively to CQRS; the coupling based on events is similar to that in more general event-driven architectures, as described by Michelson [170]. The events in the event store are processed by the system to build the query-side or execute complex processes. The CQRS pattern and its sub-patterns are described in more detail by Kabbedijk, Jansen & Brinkkemper [133]. Korkmaz [145] studies CQRS from the point of view of practitioners to gain a better understanding of the benefits and challenges. Maddodi et al. [157] study a CQRS system in the context of continuous performance testing.

2.3 Related Work

The work related to this paper is divided into data conversion, specifically schema-less data conversion, and application deployment.

Data Conversion - Two approaches to data conversion are defined by Jensen et al. [127]: *schema versioning* and *schema evolution*. Schema versioning is accommodated when a database system allows the accessing of all data, both retrospectively and prospectively, through user-definable version interfaces. Schema evolution is accommodated when a database system facilitates the modification of the database schema without the loss of existing data. Section 2.5 will show that both schema versioning and schema evolution are suitable techniques for event store upgrades.

The event store used as a storage for the command-side of the CQRS system is schema-less and, in that respect, similar to a NoSQL database as described by Scherzinger, Klettke & Störl [224] and Saur, Dumitraş & Hicks [223]. Although the store is schema-less the data itself does have a schema, but it is implicit (as defined by Fowler [86]):

the application assumes a certain schema without this schema being actually present in the store. Within relational stores the standardized DDL can be used to upgrade the schema and convert the data, a possibility missing in NoSQL stores. Scherzinger, Klettke & Störl [224] approach the implicit schema and lack of a DDL for NoSQL by proposing a new language that can be used to convert the data in a NoSQL store. Although this fills a gap in the standardization of NoSQL stores, without support in the stores the problem of data conversion in NoSQL stores remains. To aid the evolution of the data stored, Saur, Dumitraş & Hicks [223] describe an extension to one specific NoSQL database. This extension implements an approach that Sadalage & Fowler [215] describe as *incremental migration*: migrating data when it is accessed. While the research of Saur, Dumitraş & Hicks [223] is similar to the research described in this paper, their solution is tied to a specific technology and is not applicable in systems that use a similar data model with a different database technology.

Both Cleve et al. [40] and Qiu, Li & Su [201] quantify schema changes occurring in the evolution of applications. Their work is aimed at relational models, and it is unclear how these results translate to event stores. Future studies need to be conducted before these results can be applied to event stores.

The impact of schema changes on application source code is studied by Meurice, Nagy & Cleve [168] and Maule, Emmerich & Rosenblum [161]. However, the direction of the impact differs between schema-less stores and implicit schemas. The change originates in the application holding the implicit schema and impacts the data in the schema-less store.

Application Deployment - Blue-green deployment is an upgrade strategy that utilizes two slots to which different versions of an application can be deployed. One of the slots is active, while the other one is inactive. Upgrading is always done in the inactive slot, and the user is not hindered while upgrading. This strategy is followed by different authors. Callaghan [29] describes a tool written by Facebook to perform online (and zero downtime) upgrades on MySql in four phases: (1) copy the original database, (2) upgrade the copy to the new schema, (3) replay any changes happened on the original database during the copy/build phase, and (4) finally switch active databases. This approach is very similar to the pattern described by Keller [135] who applied it in the migration of a legacy system. With IMAGO, Dumitraş [63] and Dumitraş & Narasimhan [64] use blue-green deployment for their *parallel universe*: they reduce upgrade failures by isolating the IMAGO production system from the upgrade operations and completing the upgrade as an atomic operation. QuantumDB, a tool created by Jong & Deursen [130], applies the expand-contract strategy (explained in Section 2.6) with blue-green deployment.

Hick & Hainaut [108] and Domínguez et al. [60] developed and used MeDEA: a tool that focuses on the traceability of artifacts. MeDEA makes it possible to translate changes from a conceptual model of a relational database to schema changes in the actual database. Curino et al. [46] and Curino, Moon & Zaniolo [47] worked on PRISM and PRISM++, a database administrator tool that calculates the SQL statements needed to upgrade a schema. While calculating those statements it can check for information preservation, backward compatibility, and redundancy. These approaches solve the problem of analyzing schema changes and generating data conversion state-

ments, which is not part of the solution presented in this paper.

The main differences between event store data conversion and the existing research are the usage of an implicit schema and the amount of data in an event store. Furthermore, this paper does not propose a new tool specific to a certain technology or database type, but rather proposes strategies that can be applied regardless of specific technologies. In this paper, the techniques and strategies from existing work are extracted and applied to event sourcing. This results in an event store upgrade framework that can be used to design and implement an upgrade system.

2.4 Event Store Upgrade Operations

An event store contains different event streams and events. An example is given: the event store of a *WebShop* application, shown in Figure 2.2. The two streams contain many events, but only two events per stream are shown as an example.

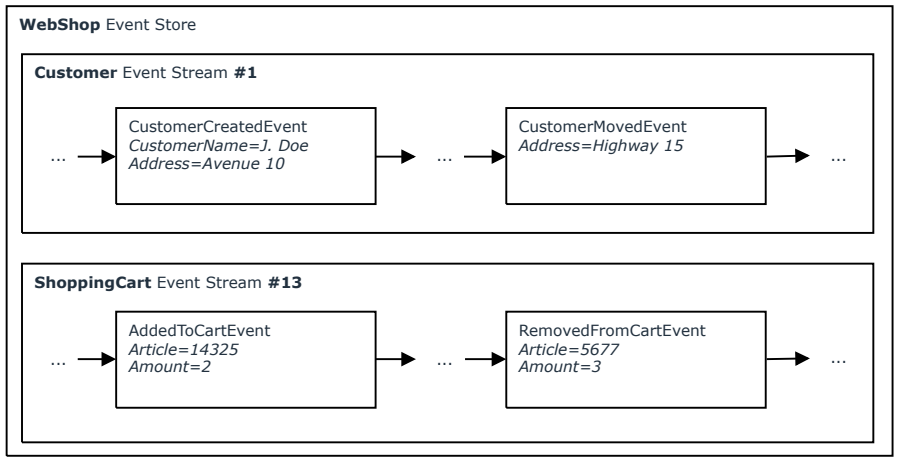

Figure 2.2: An example event store with different stream and event types.

The Figure shows two streams of the store: one for customer #1 and the other for shopping cart #13. In the application these streams belong to two separate and independent event sources. These event sources produce these events as the result of certain actions. For example, adding a product to a cart by a user should result in the *added to cart* event. The different event types such as *added to cart*, *removed from cart*, and *customer created* contain different attributes. The *added to cart* event contains the *article* (an identifier) and the amount (an integer) among others. Event listeners receive these events and create a view of the data that can be queried. This knowledge of event types and their properties is the implicit schema that is part of the application code.

Level	Complexity	Operation	Description
Event	Basic	Add attribute	An attribute is added to an event.
		Delete attribute	An attribute is deleted from an event.
		Update attribute	An attribute is updated by changing the name or value(type).
	Complex	Merge attributes	Two attributes are combined into a single attribute.
		Split attribute	One attribute is split into two attributes.
Stream	Basic	Add event	A new event is added to the stream.
		Delete event	An event is deleted from the stream.
		Rename event	An event type is renamed.
	Complex	Merge events	Multiple events are combined into one
		Split event	One event is split into two events.
		Move attribute	One attribute is moved from one event type to another.
Store	Basic	Add stream	A new stream is added to the store.
		Delete stream	A stream is deleted from the stream.
		Rename stream	A stream identifier is renamed, or a type is renamed.
	Complex	Merge streams	Multiple streams are combined into one stream.
		Split stream	One stream is split into two streams.
		Move event	An event is moved from one stream to another stream.

Table 2.1: The Event Store Upgrade Operations, categorized by level and complexity.

An event store has a structure of three levels:

The event store - A collection of streams, and every stream is of a certain type and uniquely identified by its identifier.

The event stream - Every stream is a collection of events that originated from a single source and is ordered by the event creation date. In an event sourced system there should be a single source of all the events in a single stream. The boundary of the stream is very important: a source has a one-to-one relationship with its stream. This boundary makes the event sourced systems scalable: every event source is the owner of a stream and has no relation with other streams. An event source and its event stream can be moved between machines in a cluster without difficulty. The different event streams could also be stored in different event stores. This is possible because the event source is not dependent on other sources.

The events - An event consists of a type and content in the form of key-value pairs. The type is used to route events to the projectors that are interested in specific types of events.

The event store upgrade operations are used to express how an event store version 1.0 can be transformed into version 2.0. These operations have the same purpose as the NoSQL schema evolution language proposed by Scherzinger, Klettke & Störl [224]: they give a common language to express the conversion of an event store. The complete list of operations is shown in Table 2.1. Two categorizations are applied: structure level and complexity. The operations on the store level are executed on one or multiple streams, the stream level operations convert one or more events within the same stream, and the event level operations convert a single event. The update of a stream is expressed by the stream level operations while the update of an event is expressed by one or more event level operations. Every level of operations is also categorized into basic and complex operations. Basic operations are seen as foundational operations: they cannot be expressed by other operations. Complex operations can be built by combining several basic operations, making the categories with complex operations infinitely large.

The operations presented are agnostic of the business domain of the application and its functionality. The process of expressing the transformation in these operations should be done manually, because it should reflect the intent of the upgrade. Schema changes can be expressed by different sets of operations and these different sets have their own effects. An example is given: the *WebShop* application is upgraded to a new version, and part of the upgrade is a change in storing addresses. Figure 2.2 shows the old event definition: a single attribute for both street and number. In the new version, this should be stored in two separate attributes. This data conversion can be done in multiple ways and two possibilities are given:

1. Every event could be updated with the *split attribute* operation, and this would split every address attribute in both a street and number attribute. This increases the maintainability of the system because all event handling code can assume the presence of the two new attributes.
2. Every customer stream is updated with an *add event* operation that represents the conversion. In this transformation the old information is preserved (to repair mistakes in the split operation for instance). However, now the application should be able to deal with both old and new addresses, because events can contain either of the two forms.

Although both options transform the event store differently the two resulting versions of the *WebShop* application are functionally equivalent to the users of the system. However, the inner workings differ significantly. In the first solution knowledge of the initial address property together with the values is removed. The conversion itself has changed the event store, and now it appears that the events always contained two separate attributes. The second solution retains the old addresses and adds the split in two attributes as an event to this system. This conversion keeps the old events intact and does not remove information from the store. This example illustrates the need for requirements defined by stakeholders to guide the data conversion.

2.5 Event Store Upgrade Techniques

In this section five existing techniques that can convert an event store between two schemas by means of the event store upgrade operations are discussed.

Multiple versions - In this technique multiple versions of an event type are supported throughout the application. The event structure is extended with a version number as suggested by Betts et al. [15]. This version number can be read by all the event listeners, and they have to contain knowledge of the different versions in order to support them. In this technique the event store remains intact as old versions are never transformed. There are no extra write operations needed to convert the store.

Upcasting - Upcasting centralizes the update knowledge in an *upcaster*: a component that transforms an event before offering it to the application. Different than in the *multiple versions* technique is that the event listeners are not aware of the different versions of events. Because the upcaster changes the event the listeners only need to support the last version. This technique is suggested by both Betts et al. [15] and AxonIQ [11].

Lazy transformation - This technique also uses an *upcaster* to transform every event before offering it to the application, but the result of the transformation is also stored in the event store. The transformation is thus applied only once for every event, and on subsequent reads the transformation is no longer necessary. This technique is similar to the ones described by Sadalage & Fowler [215], Roddick [209], Tan & Katayama [242], and Scherzinger, Klettke & Störl [225].

In place transformation - A technique applied by many systems using a relational database. These systems convert the data by executing SQL statements such as `ALTER TABLE` (to alter the schema) and `UPDATE` (to alter the data). As described by Scherzinger, Klettke & Störl [225], NoSQL databases do not have such a possibility. In those cases a batch job is run that reads the data, transforms it, and writes the updated data back to the database. The documents in the database are updated by this job: adding, deleting, renaming properties, and transforming the values. This technique can be applied to event stores in the same manner.

Copy and transformation - This technique is similar to the one described by Callaghan [29] and Dumitraş [63]: it copies and transforms every event into a new store. In this technique the old event store stays intact, and a new store is created instead.

The event store upgrade techniques have their own strengths and weaknesses. To make this visible the techniques are judged on four quality characteristics from ISO [118]: functional suitability, maintainability, performance efficiency, and reliability. The other four characteristics are regarded as not relevant for these upgrade techniques. Compatibility is a requirement for every upgrade system: it should be compatible with the overall system. End-users of the system will not interact with the upgrade system and thus usability is not relevant. The upgrade system is one of the components in the system, and therefore the security should not be different than in other components. Finally, portability is not considered a requirement for the upgrade systems.

Functional suitability - All five techniques can be implemented to achieve functional completeness. However, to execute complex store operations such as merging

multiple streams the technique needs to read from multiple streams. When the technique is a run-time technique such as *multiple versions*, *upcasting*, and *lazy transformation* this violates the independence of the streams. The streams could be spread out over different databases and reading them together at the same time in the application is unfeasible. Therefore, the techniques *multiple versions*, *upcasting*, and *lazy transformation* are not functionally complete. The other two techniques are executed by a separate batch job that does not adhere to the principle of reading a single stream at a time.

Maintainability - *Multiple versions* is the least maintainable technique because the support of multiple versions is spread throughout the application code. The techniques *upcasting* and *lazy transformation* have a better maintainability, because the transformation code can be centralized in those implementations. However, they all do accumulate conversion code because either the conversion result is not stored, or there is no way in telling when everything is converted. The implementation of the *lazy transformation* technique should apply all conversions that are not yet applied to specific events when needed. *In place transformation* and *copy and transformation* score the highest on maintainability because in those techniques older transformations and their code do not have to be kept. After the execution of the data conversion, every event is transformed into a new version and thus the conversion code is no longer necessary.

Performance efficiency - *Multiple versions* and *upcasting* are the most efficient, because they only transform events when they need to be transformed without adding extra write operations to the store. The transformations are done in-memory as needed, without writing the events back to the store. The techniques *lazy transformation* and *in place transformation* score a bit worse, because they add the extra write operations that permanently store the changes. *Copy and transformation* has the worst performance efficiency, because every event is read and copied to a new store, even if there are no operations affecting the event.

Reliability - Three techniques score high on reliability; either they do not change the store (*multiple versions* and *upcasting*) or make a backup (*copy and transformation*). The other two techniques change the event store permanently, making a backup mandatory.

Table 2.2 shows the overview of the different techniques and their evaluation with respect to the four quality characteristics. A plus means that the technique satisfies the quality characteristic, a minus means that the quality characteristic is not satisfied. A plus-minus expresses an acceptable satisfaction, but there is room for improvement. These ranks are the result of both literature study and evaluation with the experts as described in Section 2.8.

Table 2.2 shows a preference for *upcasting* on the four quality characteristics, but specific context or requirements could steer companies towards a different technique such as *multiple versions*. These requirements could be a short time to market (and thus not having the time to implement a more maintainable technique such as *upcasting*). The event store upgrade operations related to multiple event sources are considered to be executed by non-run-time techniques only. However, the choice for a run-time technique when complex store operations are not supported is not compul-

	Functional suitability	Maintainability	Performance efficiency	Reliability
Multiple versions	+/−	−	+	+
Upcasting	+/−	+/−	+	+
Lazy transformation	+/−	+/−	+/−	−
In place transformation	+	+	+/−	−
Copy and transformation	+	+	−	+

Table 2.2: The event store techniques compared on four quality characteristics. A + means that a characteristic is satisfied, +/− indicates room for improvement, while − means that the characteristic is not satisfied.

sory. Systems can implement a non-run-time technique even if they do not plan to support complex store operations.

2.6 Application and Data Upgrade Strategies

According to Humble & Farley [113] and Jansen, Ballintijn & Brinkkemper [123], deploying software involves three phases: *Prepare and manage*, *Installing*, and *Configuring*. In the first phase, the environment in which an application is deployed should be prepared and managed: both hardware and software dependencies should be in place. During the Installing-phase the application itself is deployed. In the final phase the Configuring-phase is used to configure the application and make it ready for use.

The techniques that are discussed in the previous section are performed in different phases. Three of the five techniques were already identified as run-time techniques in the previous section: *multiple versions*, *upcasting*, and *lazy transformations*. They execute the event store upgrade operations at run-time and are deployed along with the application binaries, therefore they are part of the Installing-phase.

The last two techniques, *in place transformation* and *copy and transformation*, are not part of the actual application. Both techniques perform the data conversion within a separate batch job that needs to be run before the new application version is deployed, and therefore belongs to the Configuring-phase. Although the code that performs the technique should be deployed it cannot be part of the application as the application itself is only deployed in the Installing-phase. These two techniques require a second deployment strategy aimed at the deployment of the data conversion logic.

The simplest deployment strategy is to copy the new application onto the machine(s) replacing the older version. Brewer [23] refers to this approach as a *fast reboot*. The time that it takes to bring down the application process, copy the new application, and start the application process again is the downtime that is observed with this strategy. Its simplicity is its biggest selling point, but its biggest downside is that this strategy is not without downtime. Deployment strategies described by Pulkkinen [200] such as feature flagging, dark launching, and canary release are excluded from the list of discussed strategies, because they are specifically used to gain more knowledge about the users and/or (system) performance. Four strategies found in literature, suitable

for upgrading an event sourced system, are discussed:

	Application upgrade strategy	**Data upgrade strategy**
Multiple version	*big flip*, *rolling upgrade*, *blue-green*	
Upcasting	*big flip*, *rolling upgrade*, *blue-green*	
Lazy transformation	*big flip*, *rolling upgrade*, *blue-green*	
In place transformation	*big flip*, *rolling upgrade*, *blue-green*	*expand-contract*
Copy and transformation	*big flip*, *rolling upgrade*, *blue-green*	*expand-contract*, *blue-green*

Table 2.3: Combinations of techniques and strategies that result in zero downtime.

Big flip - This strategy, described by Brewer [23], uses request routing to route traffic to one half of the machines, while the other half is made available for the upgrade. The traffic is rerouted again when the first half is upgraded after which the second half can be upgraded. When all machines are upgraded, the load balancer again can route the traffic to every machine. During the upgrade only half of the machines can be used to handle the traffic.

Rolling upgrade - This strategy also uses some form of request routing to make sure that some machines do not receive requests. The machines in this strategy are upgraded in several upgrade groups defined by Dumitraş, Narasimhan & Tilevich [65]. Because a small number of machines is being upgraded at a time, more machines are available to handle the traffic. However, the machines that are available are running mixed versions of the application: both those that are not yet upgraded and those that are already upgraded. This makes rolling upgrades complex, and the application should be able to handle these types of rolling upgrades.

Blue-green - Blue-green deployment is described by both Humble and Farley [113] and Fowler [85]. According to Humble [113] this is one of the most powerful techniques for managing releases. Every application is always deployed twice: a current version and either a previous version or a future version. One of the deployments is active at a given time, either the *green* slot or the *blue* slot. When the application is upgraded, the inactive slot is used to deploy the new version. Blue-green deployment can be done without downtime, as no traffic is going to the version that is upgraded. After the upgrade, the traffic can be rerouted to the upgraded slot, switching between blue and green. This strategy is similar to the *big flip* strategy, but reserves extra resources for the upgrade while the *big flip* strategy uses existing resources and thus limits the capacity during an upgrade.

Expand-Contract - A strategy described by Sato [222] as consisting of three phases, also known as *parallel change*. The first phase is the expand phase: an interface is created to support both the old and the new version. After that the old version(s) are (incrementally) updated to the new version in the migrate phase. Finally in the contract phase, the interface is changed so that it only supports the new phase. This strategy is suitable for upgrading components that are used by other components. By first expanding the interface of the component, the dependent components can be upgraded. When all dependent components use the new interface the old interfaces can be removed. This strategy is not suitable for application upgrades, however, it can be utilized in upgrading the database.

An upgrade of an event sourced system needs an application deployment strategy. This deployment strategy executes the run-time event store upgrade technique, but if the upgrade uses a non-run-time technique a data upgrade strategy is also required. The three run-time techniques *multiple versions*, *upcasting*, and *lazy transformation* only need an application deployment strategy as they do not alter the data store. The other two techniques, *in place transformation* and *copy and transformation*, do need a data upgrade strategy.

Not all combinations result in an upgrade that does not affect the operation of the system in a negative manner. Table 2.3 summarizes the combinations that would lead to a zero downtime upgrade. For the run-time techniques, *multiple versions*, *upcasting*, or *lazy transformation*, an application upgrade strategy is sufficient, and the *big flip*, *rolling upgrade*, and *blue-green deployment* strategies will all result in a zero downtime upgrade. All three strategies upgrade part of the machines while maintaining operations on the other parts, and the techniques are performed at run-time.

For the non-run-time techniques, *in place transformation* and *copy and transformation*, the same three application upgrade strategies can be used and result in zero downtime upgrades. However, a data upgrade strategy is also needed to execute the batch job that converts the data. The strategy *blue-green* in combination with *in place transformation* is not possible, because the in place nature of the technique conflicts with the strategy that needs to have two slots available. Therefore, the technique *in place transformation* only works with the *expand-contract* strategy. *Copy and transformation*, the other non-run-time technique works with both the data upgrade strategies, *blue-green deployment* and *expand-contract*.

2.7 Event Store Upgrade Framework

This section explains how the event store upgrade operation, techniques, and strategies form the event store upgrade framework that can be utilized in two distinct manners. Figure 2.3 shows the different event store upgrade operation, techniques, and strategies and their combinations.

The first row of Figure 2.3 shows the event store upgrade operations, the darker yellow identifies the category of operations that crosses event streams and cannot be executed run-time. The event store upgrade techniques are colored green, the darker elements identify schema evolution techniques, while the others are schema versioning techniques. The last two rows identify both application and data upgrade

strategies. The arrows between single elements, or groups of elements, identify the valid combinations. The valid combinations are explained in more detail along with the utilization of the framework.

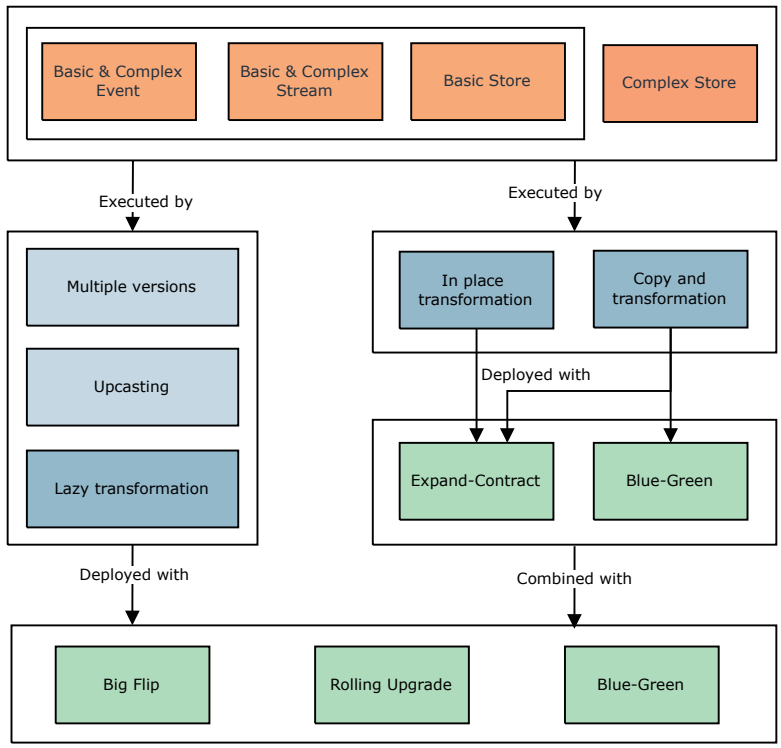

Figure 2.3: The Event Store Upgrade Framework.

The first utilization of the event store upgrade framework is a decision tree that supports the upfront design and implementation of an upgrade system for event sourced systems, presented in Figure 2.4. This tree shows the decisions that form the design and implementation of an upgrade system.

The design starts with the question if complex store operations need to be supported. This decision influences the possible techniques that can be applied, because these complex store operations cannot be executed with run-time techniques. When support for the complex store operations is not needed the next step is the choice of *event store run-time upgrade technique*. Any of the three run-time techniques, *multiple versions*, *upcasting*, or *lazy transformation*, will be sufficient (shown with a single arrow and the || combinator) and Table 2.2 can be used to decide what technique fits the context. When the upgrade system should support complex store upgrade operations, the choice is between the non-run-time techniques *in place transformation* and *copy and transformation*. The *expand-contract* application strategy follows automatically if *in place transformation* is chosen as a data upgrade strategy (shown with a single arrow and the && combinator). Two different data upgrade strategies can be chosen

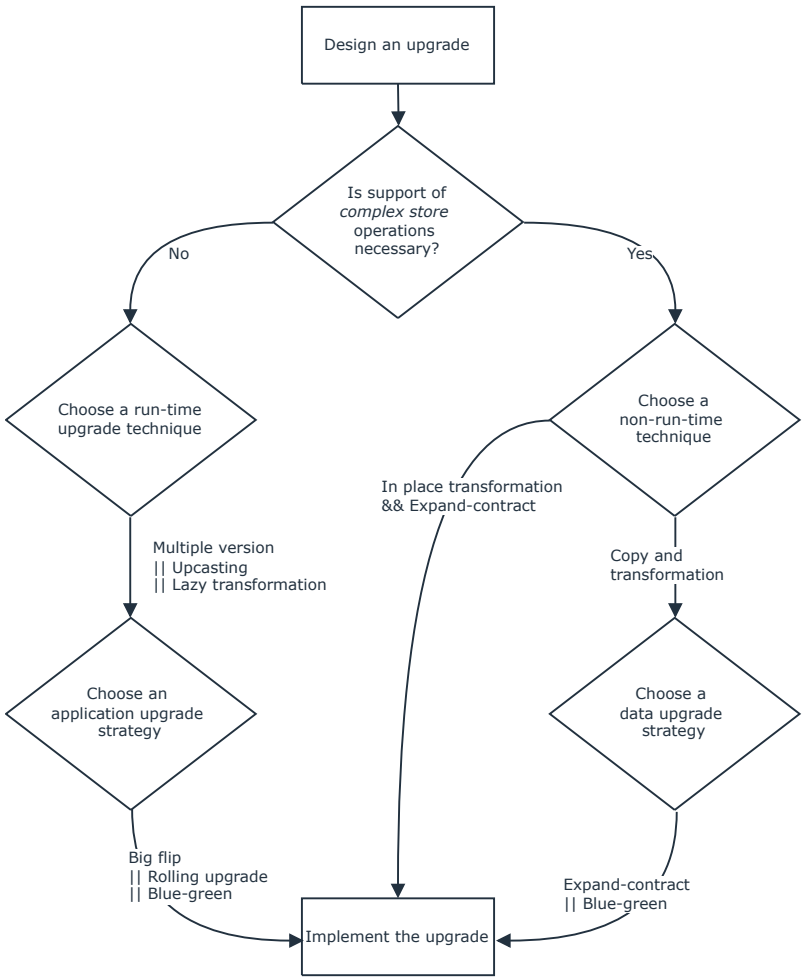

Figure 2.4: Decision Tree for the Design and Implementation of an Event Store Upgrade System.

with the technique *copy and transformation* as follows from Table 2.3. Although the utilization of the framework for upfront design has much room for context-specific choices, it shows what the possibilities are and makes the trade-offs explicit.

The second utilization of the event store upgrade framework is a run-time decision-making system. This system is implemented in the event store upgrade system, and is visualized in Figure 2.5. In this system the analysis of the event store upgrade operations that need to be executed is done at upgrade time. When the operations do not contain complex store operations, the system can apply the run-time technique in combination with the application upgrade strategy. If there are complex store operations the system can deploy the non-run-time technique with the data upgrade strategy, and then apply the application upgrade strategy. In this utilization, the choice for run-time technique, non-run-time technique, data upgrade strategy, and application upgrade

strategy is made upfront. The system implements both a run-time and non-run-time technique that fits the requirements. The two techniques are completed with an implementation of a data upgrade and an application upgrade strategy. Having these implementations in the system allows for a fully automated upgrade system based on Figure 2.5.

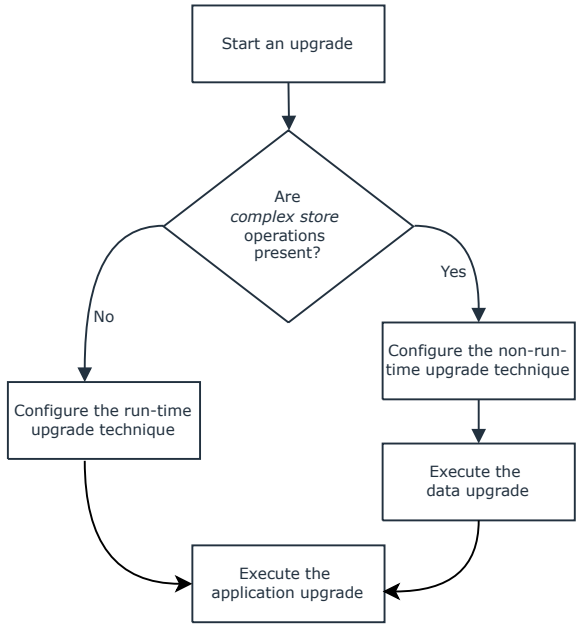

Figure 2.5: Decision Tree for an Automated Event Store Upgrade System.

2.8 Evaluation

This work was done in the context of the development of a large CQRS system at AFAS Software. Two authors are working as architect and developer on this system, and an earlier list of the event store upgrade operations was discussed with the team that also works on this system. The operations were implemented in combination with the *copy and transformation* technique and the *blue-green* strategy. Multiple data conversions were executed with this upgrade system in a smaller experimental setting. The results of these conversions were promising, but more systematic experimentation is necessary. Initial results showed that the event store upgrade operations executed with *copy and transformation* and deployed with *blue-green* were able to handle a diverse range of scenarios. These operations could all be performed while maintaining the operation of the system, no downtime was observed. However, it also showed that the time needed to perform the conversion was longer than expected, because the query store needed to be rebuilt as well.

To evaluate the event store upgrade framework, interviews were held with three Dutch experts in the field of CQRS and event sourcing. They were selected because of their experience with CQRS and event sourcing, and for their presence in the community through speaking engagements. Allard Buijze is the founder and architect of the Axon CQRS Framework[1], and has more than six years of experience with CQRS and event sourcing both as a developer and consultant. Dennis Doomen is the lead architect of a large CQRS system. He has six years of experience with CQRS, and four years with event sourcing. He shares this experience with the community as an international speaker and often discusses this with other practitioners. Pieter Joost van de Sande is the founder of NCQRS[2], an open-source CQRS framework. He started applying CQRS and event sourcing more than six years ago, and recently started working on a large event-driven architecture. All three interviewees received training on CQRS and event sourcing from Greg Young. Multiple goals were set for conducting the interviews. The questions asked were directed towards these goals, and the interviews followed this order.

1. Reveal what their experiences of upgrading CQRS applications are, and what problems and situations they ran into.
2. Evaluate the utility and completeness of the event store upgrade operations.
3. Evaluate the completeness and the judgment on the quality characteristics of the event store upgrade techniques.
4. Evaluate the usefulness and completeness of the event store upgrade framework.

The interviews led to small adjustments in the overviews and the event store upgrade framework, as summarized in the remainder of this section.

Naming Issues - Many small misunderstandings were experienced around naming event store upgrade operations, techniques, and application and data upgrade strategies. This shows the immaturity of the field, the relatively new concept of event sourcing, and the joining of different fields (domain-driven design, distributed systems, and event-driven architecture). The event store upgrade technique *lazy transformation* was initially named *lazy upcasting*, but this name caused confusion. It is not the upcasting that is done lazy, but the transformation of the store that is executed lazily. The technique *copy and transformation* was initially named *replay of events*, which was confused with the normal procedure that is used to load aggregate roots and rebuild the projectors by replaying the events in a CQRS system.

Frequency of Operations, and Business Requirements - All interviewees agreed that the event store upgrade operations across the boundaries of event streams were not common. One of the interviewees stated "You will not see a lot of complex event store operations. There is an exponential relation between the level and the times you encounter an operation." These operations cause the need for a data upgrade strategy, which is something that two interviewees found conflicting with the expected immutability of the event store. As a result of this discussion, the support of complex store operations became the first question in the event store upgrade framework. The interviewees explained that an event store upgrade system can be useful, even if it

[1] `http://www.axonframework.org/`
[2] `https://github.com/pjvds/ncqrs`

does not support these complex store operations, because they see other possibilities of solving these schema changes. The same holds true for the operations that delete information from the store, and all interviewees suggested that deprecating or archiving data is preferred over the actual deletion.

These discussions led to the distinction between *functional immutability* and *technical immutability*. *Technical immutability* is defined as the most strict form of immutability: no changes to stored events are allowed to be made. If this level of immutability should be preserved the schema evolution techniques (*lazy transformation*, *in place transformation*, and *copy and transformation*) cannot be applied. However, as one of the interviewees stated, another level of immutability is *functional immutability*. Functional immutability allows the transformation of events as long as the information in the events is preserved from a business perspective. Within functional immutability there is far more room for techniques that alter the stored events.

Variation in Implementation - Two out of three interviewees described a variation of the technique *multiple versions* that improved the code reusability and maintainability. By re-using the already existing code to read older versions the maintainability is improved. These comments show that there is a large design space in implementing the techniques that removes some of the disadvantages. However, Table 2.2 was not changed, because the conclusion was that the average quality of the technique was not changed by these implementation variations.

Projections - An event sourced system always has a data store for querying and presenting data next to the event store, because the event store itself cannot be utilized for that purpose. This query store is built from the event data by projectors resulting in *projections*. These projections are the data that is shown to users while the event store is used for the validation of new changes. Two interviewees explained that many schema changes can be applied by not changing the event schema at all, but by only changing the projectors and projections. The feasibility of this approach and its up- and downsides are regarded as out of the scope of this paper, and should be studied in more detail.

Upfront Design and Prototyping - The interviews show two ways of looking at event sourced systems. Two out of three interviewees emphasized the importance of upfront design: by designing the event store, the streams, and the events with enough upfront thought, upgrades are less often necessary. One interviewee stated "Event sourcing needs a lot of upfront thought, which is hard to do with agile development." This line of thought is also seen in the application of *event storming*[3] in the design of event sourced systems. This design technique is applied to design the events in a system before implementation, and a good design is said to forestall some of the more complex event store upgrade operations, such as those on multiple event streams. The other interviewee stressed the importance of doing event store data upgrades to prevent the accumulation of conversion code, and thus found less value in defending technical immutability at all cost.

Completeness and Usefulness - The interviewees found the event store upgrade framework unanimously useful and complete. Interest in the end result was shown and encouragement was given to publish this material. One interviewee stated that

[3]See `https://www.infoq.com/news/2016/06/event-storming-ddd`.

"You are maybe the only one who has created such an overview and also thought about edge cases which I hope never to encounter."

2.9 Conclusion and Future Work

This paper contributes to the research on event sourcing and data conversion in the following ways. First, event store upgrade operations are presented to explicitly express the data conversion needed to evolve an event sourced system to a new data schema. With these operations a common language is proposed for to express schema evolution for event sourced applications, frameworks, and their upgrade systems. The operations can also be used to analyze the impact of an event store upgrade: one category of operations, the complex store operations, cannot be executed at run-time without violating the independence of the different event streams.

The second contribution is an overview of upgrade techniques and strategies that are used in event sourcing to execute the event store upgrade operations. This overview summarizes best practices and literature and makes it accessible to other practitioners. The last contribution is the event store upgrade framework, which is utilized upfront to design and implement an upgrade system. The framework makes the trade-offs explicit, and supports the making of design decisions. The automated utilization can be used to implement an event store upgrade system that handles every event store upgrade operation in an efficient way. The framework enables decision-making regarding upgrades downtime and enables the selection of the most performant technique and strategy. When there are no complex store operations the conversion can be done at run-time, and techniques that transform events in the event store are not needed. This leads to upgrades that only need an application upgrade strategy, which can be applied faster than the upgrades that also need a data upgrade strategy. The maintainability problem that run-time techniques have can be solved by executing those accumulated conversions whenever a data upgrade is performed.

The event store upgrade framework is also usable as a tool to analyze applications with respect to their level of readiness for the cloud, for continuous delivery, and rapid software evolution. Applications that do not have a clear upgrade system, but use ad-hoc data transformation are not ready. Upgrades are done manually and are error-prone. However, applications that implement an automated upgrade system and can handle the complete list of event store upgrade operations are ready for continuous delivery. This allows those applications to incorporate improvements and prevent errors in doing manual upgrades.

Part of the upgrade framework is implemented in a CQRS system. The *copy and transformation* technique together with the *blue-green* strategy are used in multiple experiments to transform an event store. This showed that more work is needed to enable the co-evolution of the stores on the command-side and query-side. The framework was evaluated with three Dutch experts in the field of event sourcing. Although only three experts were interviewed, and they had different opinions, the event store upgrade framework was found to be valuable by all three. The relative novelty of event sourcing can cause problems in understanding concepts and definitions. The combination of literature study and expert interviews prevents validity problems in

definitions and their interpretation and in making sure that the result of this paper is usable by other practitioners.

To validate the event store upgrade framework the authors plan to implement the full automated upgrade system that uses the event store upgrade operations to select an upgrade technique and apply the upgrade strategies. A follow-up study on the frequency of schema changes in event sourced systems, and the possible operations should support this implementation. The results of such a study could also help to uncover business decisions in expressing different schema changes with regard to, for example, data loss. Finally, the upgrade system could be extended by also including the query-side of an event sourced system. This paper only focuses on the event store, but as the interviewees stated, schema changes can be implemented by upgrading the projection, and not the event store. Furthermore, a change in the event store also changes these projections and the rebuilding of projections with the additional performance costs is a problem that also needs more study.

Acknowledgment

Acknowledgment The authors thank the three experts, Allard Buijze, Dennis Doomen, and Pieter Joost van de Sande, for their valuable experience and willingness to contribute to this study.

3

An Empirical Characterization of Event Sourced Systems

Event sourced systems are increasing in popularity because they are reliable, flexible, and scalable. In this article, we point a microscope at a software architecture pattern that is rapidly gaining popularity in industry, but has not received as much attention from the scientific community. We do so through constructivist grounded theory, which proves a suitable qualitative method for extracting architectural knowledge from practitioners.
Based on the discussion of 19 event sourced systems we explore the rationale for and the context of the event sourcing pattern. A description of the pattern itself and its relation to other patterns as discussed with practitioners is given. The description itself is grounded in the experience of 25 engineers, making it a reliable source for both new practitioners and scientists. We identify five challenges that practitioners experience: event system evolution, the steep learning curve, lack of available technology, rebuilding projections, and data privacy. For the first challenge of event system evolution, we uncover five tactics and solutions that support practitioners in their design choices when developing evolving event sourced systems: versioned events, weak schema, upcasting, in-place transformation, and copy-and-transform.

This work was originally published in *Journal of Systems and Software In Practice, volume 178 (2021)*, titled 'An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry'. It was co-authored by Marten Spoor, Slinger Jansen, and Sjaak Brinkkemper.

3.1 Introduction

Software systems are increasing in complexity, used in increasingly critical processes, and serve increasing numbers of end-users. Architectural patterns enable engineers to build these systems using knowledge acquired by other engineers. Influential books such as *Patterns of Enterprise Application Architecture* by Fowler [83] and *Enterprise Integration Patterns* by Hohpe & Woolf [111] demonstrate the impact of pattern descriptions on software engineering. Architectural patterns are part of the trend of knowledge-based architecture design; Li, Liang & Avgeriou [154]. Kassab et al. [134], Taibi, Lenarduzzi & Pahl [240], and Harrison, Avgeriou & Zdun [103] show how patterns are instrumental in the capturing of architectural design decisions. In this article, we describe such a pattern in detail and provide the design decisions that were employed in practice, with the goal of providing a comprehensive source of knowledge for practitioners.

Recently, the event sourcing pattern has become a popular answer to the challenges of complex, mission-critical, scalable systems. Examples of organizations that apply event sourcing are Netflix [9], and Walmart's Jet.com [96], both which have as goal the creating scalable and reliable critical systems. Event sourcing is informally described by Fowler [84] as a pattern that "ensures that all changes to application state are stored as a sequence of events." Flexibility, debug-ability, and reliability are given by Avery & Reta [9] as a rationale for using event sourcing. Debski et al. [55] and Erb & Hauck [71] show how event sourcing can be applied to achieve scalable, reactive systems. Kabbedijk, Jansen & Brinkkemper [133] describe event sourcing as a subpattern of Command Query Responsibility Segregation (CQRS) in their work on the improved variability and scalability of systems applying CQRS.

The events in event sourcing, as opposed to general event-driven architectures (EDAs) [87], are stored as an append-only log of all state changes. Two key characteristics separate event sourcing from event-driven approaches, such as stream processing, transactional processing, and blockchain. First, events in Event Sourced Systems (ESSs) are stored as the state of the application. Other approaches use the events to communicate, while the communication aspect comes second in ESSs. The second difference is that events are closely related to events occurring in real world business processes. This allows event sourcing to also be used as a design approach. Domain-Driven Design (DDD), as described by Evans [74], advocates events as a design tool for the process flow of a software system. Brandolini [21] proposes *event storming* (analogous to brainstorming), a group design process that focuses on the events that take place in a software system. Further details on these analogous approaches are found in Section 3.3.

Although event sourcing is related to existing ideas such as EDAs, the pattern itself has not yet been thoroughly studied. Most knowledge exists in so-called 'grey literature': practitioner blogs, and anecdotal experience reports. In previous work [186], which focused on the evolution of ESSs, we experienced this lack of literature. This work fills this gap by deriving an integral description of event sourced systems through interviews with 25 engineers. Together with this description we identify four categories of rationale for the application of event sourcing, such as a decrease of com-

plexity. In this "In Practice" submission, we also identify five engineering challenges around the pattern, with schema evolution being one of the most complex challenges. With the pattern description and its liabilities presented in this article, we enable engineers to make a considered choice. Our work is not dissimilar to the work of Musil, Musil & Biffl [172], who conducted an extensive study on collective intelligence system pattern variations, with the goal of enabling architects to predict the outcomes of different design decisions. Similarly, Slotos [230] describes the Star pattern for enabling flexible business applications, also with the goal of supporting software architecture researchers and practitioners and promoting the pattern itself.

Our study regards a new research area, therefore, we apply Grounded Theory (GT). Adolph, Hall & Kruchten [2] describe GT as a useful approach for research in areas that have not previously been studied.

A GT explains how people resolve their main concern by employing a certain process. This process is called the 'core category' of the GT. The core category of the work presented in this article is the process of designing and implementing event sourced systems, as performed by software engineers. The theoretical definition of event sourcing helps both researchers and practitioners to understand, reason about, and teach the pattern and its consequences. Section 3.2 explains how we applied GT to form a basis for the conceptualization of ESSs from 25 interviews, and how the three essential elements are covered. From the gathered data we distill the pattern description and its consequences. This work has the following contributions:

- Section 3.3 contrasts ESSs with other existing architectural patterns, such as EDAs and blockchain, and shows that ESSs are insufficiently described in existing literature.
- Section 3.4 describes the rationale for using ESSs: they provide audit functionality, are highly flexible and scalable, enable the development of highly complex systems, and are a current trend. The overview of 19 different ESSs elaborates on the context of the pattern, showing that event sourcing is applied in different kinds of systems, from small to extremely large.
- Section 3.5 provides a thorough description of ESSs based on the findings of the interviews and presents the pattern itself including its relation to CQRS. It also reflects on the role of the (implicit) schema present in ESSs.
- Section 3.6 presents the **engineering challenges** surrounding the use of the pattern that engineers encounter during the development of ESSs, such as a steep learning curve, poor ESSs performance, and dealing with privacy regulations such as the General Data Protection Regulation (GDPR).
- Section 3.7 focuses on the most prominent challenge encountered in ESSs: **schema evolution.** Five empirically established methods are presented that support ESS evolution. We advise that systems should start out using versioned events and weak schema, while later evolving to upcasting and even copy-and-transform techniques.

The validity threats of this work, such as the fact that the interviewees were pragmatically collected, are discussed in Section 3.9. We conclude that ESSs enable complex scalable systems with auditing capabilities and that our theoretical definition enables further research and development of these systems.

3.2 Research Approach: Constructivist Grounded Theory

In our early literature search, we identified that there is little academic material available when it comes to the topic of event sourcing. Grounded Theory (GT) is defined as a systematic methodology involving the construction of theories through methodical gathering and analysis of data. Adolph, Hall & Kruchten [2] explain how GT is particularly useful for research in areas that have not been studied before. Our investigation of ESSs has an exploratory nature, therefore, we use GT to structure our research approach. Furthermore, we aim to inspire researchers to experiment with novel approaches in gathering architecture knowledge.

GT is a common research strategy in software engineering research and induces theory from empirically collected material, such as through interviews or case studies. For instance, Hoda, Noble & Marshall [110] explore the practices of self-organizing agile teams using GT. Greiler, Deursen & Storey [99] apply GT to improve the understanding of testing practices for plug-in systems. Tamburri & Kazman [241] recover software architectures by applying GT. Last, Santos et al. [220] study common vulnerabilities in plug-and-play architectures through GT.

Similarly, we use GT to explore event sourcing, and improve our understanding of the pattern, the applications, and the challenges. Constructivist GT assumes that neither data nor theories are discovered, but are constructed by the researchers out of the interactions with the field and its participants. Data are co-constructed by researchers and participants, and coloured by the researchers' perspectives, and values. Within this approach, a literature review is used in a constructive and data-sensitive way without forcing it on data. We have employed constructivist GT [33] in our research; we knew we would find a description of the pattern, but were not aware what other concepts, challenges, and motivations would be identified.

3.2.1 Research Questions and Motivation

The motivation of our research is formed by five years of experience in the development of an event sourced system and by earlier research on schema evolution in ESSs [186]. This experience guided our research and the direction of our exploration. Effectively, our previous work is also part of the GT data set, and has been translated directly into the research protocol. The main goal of the research project was to come to a cohesive theory around the event sourcing architecture pattern. The research questions guided the research and were formulated, as per constructivist GT, a priori, but evolved into the following final set:

RQ1 What types of systems apply event sourcing and why?

RQ2 How can event sourced systems be defined?

RQ3 How can event sourced data structures be evolved?

RQ4 What are the challenges faced by practitioners in applying event sourcing?

Our previous study in the domain [186] gained significant industry interest, which led us to attend many industry events, where we were often invited as keynote speakers. This provided us extensive access to practitioners in the field, who would offer

their support and advice. Through these rich interactions it became obvious that an extensive interview study could lead to new results and research challenges in the domain.

Foundations for the Study. While in GT it is recommended that the researchers do not perform an extensive literature study before the research project, many have acknowledged that this is almost impossible and at times even impractical [33, 237]. As little academic literature was available, it was easy to fulfill this major GT guideline. This research project was started after we had already published in this domain [186] ourselves. We made our previous work part of the initial data set and also included the works of Fowler [87], e.g., the main concepts were extracted from these works and subsequently used to create an interview protocol. Throughout the project, as we gathered new evidence and encountered new concepts, we performed exploratory literature study projects for each. Furthermore, if the interviewees mentioned an academic paper, it became part of our literature set. New concepts were extracted from this literature and integrated with the interview protocol where necessary. The literature was explored by snowballing forward and backward one level.

3.2.2 Sampling and Interviewees

The interviewed engineers volunteered to contribute to our research after being invited through different channels. Based on our experience in developing ESSs in the past years we identified the primary locations through which the event sourcing and DDD community communicates. We invited the engineers through channels such as Google Groups and Slack. In addition to this open invitation, we explicitly contacted and invited a number of well-known community members. We executed *interview snowballing*, a process similar to snowballing in systematic literature studies [270]: we explicitly asked each interviewee for further references. The interviewees were not compensated for their cooperation.

Our direct and indirect invitations resulted in interviews with 25 engineers. The engineers are event sourcing practitioners in the roles of developers, architects, and product owners. A number of these engineers were consulting with the company, while others were employed by the company. The consultants operate as external advisers (in addition to being hired as developer or architect) and are hired by multiple companies because of their experience. Table 3.1 summarizes the engineers, including their role, years of experience with ESSs, and the number of ESSs they worked on. Combined they have 103 years of experience, with an average of four years per engineer. For two of the engineers (E14, E16) it is hard to tell how many systems they worked on over the years, because their consultancy work exposed them to many different systems. A number of the engineers worked on the same system(s), and were interviewed together. We conducted 22 distinct interviews with the 25 engineers. Three interviews were conducted with two engineers together as these engineers worked on the same system. In the case of E4 and E5, and E20 and E21 the engineers had a different role, and their experiences complemented each other during the interview. Engineers E9 and E10 shared their role, and their answers showed more overlap. The systems are discussed in Section 3.4. We will refer to the engineers by the number given to them in Table 3.1.

	Role	Location	Experience (years)	Nr ESSs
E1	Architect, Developer	North America	4	3
E2	Developer	Europe	2	1
E3	Developer	North America	2	1
E4	Architect	Europe	2	1
E5	Developer	Europe	2	1
E6	Architect, Developer	Asia	15	3
E7	Architect, Developer	Europe	4	3
E8	Consulting Developer	Europe	2	1
E9	Consulting Developer	Europe	3	2
E10	Consulting Developer	Europe	3	2
E11	Architect, Developer	North America	9	3
E12	Developer	Europe	3	1
E13	Developer	Europe	2	1
E14	Consulting Architect	Europe	10	*multi*
E15	Developer	Europe	1	1
E16	Consulting Architect, Developer	Europe	7	*multi*
E17	Architect	Europe	2	1
E18	Architect	Europe	2	1
E19	Architect	North America	3	1
E20	Product Manager	Europe	2	1
E21	Architect	Europe	2	1
E22	Architect	Asia	5	1
E23	Architect	Europe	9	1
E24	Architect	Asia	5	3
E25	Developer	Europe	2	1

Table 3.1: Summary of the interviewed engineers. We list roles (all technical except one), location, years of experience with ESSs and number of ESSs worked on.

3.2.3 Interview Techniques and GT

Each interview took 30-90 minutes, either in person or via video conference. The protocol presented in Section 3.11 was created using the guidelines of Castillo-Montoya [31]. During the interviews, we asked open-ended questions exploring event sourced systems. The questions asked during the interviews were based on a protocol that is downloadable with the interview transcripts [188].

The protocol was followed freely: the answers given by the engineers guided the interviews. The four parts of the protocol remained stable during the interviews. Some of the interview questions were sharpened and added as the interviews progressed, a technique encouraged by practitioners of GT. The protocol used in the last interview is presented in Section 3.11. The first part of the interviews focuses on the context of the event sourced system and the engineer: what are the characteristics of the system, and why is event sourcing applied. Versioning of event sourced systems is discussed in the second part of the interviews, based on our experience in this topic we identified this as an important challenge. The third part deals with the relation of event sourcing

with CQRS, DDD, and other challenges. Finally, we discuss whatever the engineers thinks should be discussed in relation to event sourcing.

3.2.4 Coding, Analysis, and Creativity

Each interview transcript was analyzed, as part of the GT approach, through an open coding process. The interviews were conducted by the first author, the transcripts were reviewed by another author after creation. The first and second author performed the codification and categorization, while the third author validated and confirmed the steps. The authors maintained a shared *memo-ing* document where ideas and emerging concepts were noted for discussion with all co-authors. Disagreements in the codification and categorization were resolved through discussions among the authors until agreement was found, while older versions of concepts were maintained in the memo-ing document. The coding process was both organic and methodical.

We provide an example of the coding process. One of the concepts that was discussed extensively was that of auditing and the ability to have a change log for all events in the system. E2: *it "has saved the finger of blame from pointing at us so many times ... that bit is worth its weight in gold to me."* E4, translated: *"I would save the old version forever ... for if we end up in court."* Many of the interviewees put equal emphasis on the role of the audit log. The paragraphs from the transcripts mentioning the audit log were first coded and linked to the concept *audit*. From those codes *audit* emerged as one of the prevalent rationales behind the pattern. After further grouping the statements linked to *audit* we added more detailed codes, particularly addressing specializations of this rationale such as *customer service support* and *regulations*.

This example explains how we started with highlighting important paragraphs and sentences in the transcripts. Those highlights were coded with short summary sentences. After that the sentences were grouped by linking them to codes: topics described by a few words. From those codes we derived concepts, such as the previously mentioned *audit*, which was later related to the category *rationale*. During this process we iterated until we ended with simplified categories and concepts (also known as the *parsimony* principle) that reflected the linked paragraphs. This process was iterative and organically executed until the first and second authors agreed on the categories and concepts.

While we cannot claim that saturation was reached, this article is a presentation of the coherent concepts that emerged from the research. The nature of our study is exploratory and the research questions are broad on purpose. To reach saturation on such a large topic one would have to conduct, transcribe, and codify an impractical number of interviews. Although saturation based on the codes and concepts was not reached, we are confident that the results we present represent the general sentiment among practitioners. While we always had the concepts of how to present an architecture pattern in the back of our minds, we decided to structure the presentation according to the results of the GT concepts and codes. The guidelines as, for instance, stated by Gamma et al. [90] on describing a pattern through the elements *problem*, *solution*, and *consequences* were used during the memo sorting process to match our concepts, but not as a predefined framework in which our concepts were painstakingly framed. Section 3.8 discusses the relation between our concepts and the guidelines

of Gamma et al. [90]. The categories, concepts, and codes found during the interviews are presented in Sections 3.4, 3.5, 3.6, and 3.7. Tables 3.3, 3.2, 3.4, 3.5, 3.6, and 3.7 summarize the results.

The interview protocol, the anonymized transcripts of the interviews, and the classification codes with links to the interviews are made available as a data package [188].

3.3 Background

The foundational idea of event sourcing is the domain event as described by Evans [75]. His seminal book on Domain-Driven Design (DDD), however, does not mention the pattern. Vernon [255] describes event sourcing only briefly in his book on the implementation of various DDD patterns. Young [277], as one of the original proposers of event sourcing, discusses the challenge of versioning ESSs. Event sourcing is also discussed in the context of CQRS Young [275], a pattern strongly related to event sourcing. Recent academic literature [70, 278] shows an interest in applying event sourcing for research projects.

Three related areas and their differences with respect to ESSs are discussed: transactional processing and database systems, stream processing and EDAs, and blockchain.

Event sourcing is related to database systems techniques used for persistence guarantees and replication. Gray & Reuter [97] describe how transaction logs can be used to replicate the state between database systems. Every state change is recorded as a transaction, which is similar to event sourcing where every state change is recorded as an event. Kleppmann [140] discusses event sourcing in the context of data-intensive applications; he relates the pattern to the *change data capture approach*, often used in Extract-Transform-Load (or ETL) processes [253]. ETL solutions are often used for creating data warehouses. The primary difference between event sourcing and these techniques is that a transaction or a data change is a technical entity without relation to the real world, while an event in event sourcing resembles an event in the real world.

Kleppmann also relates event sourcing to the *chronical data model* described by Jagadish, Mumick & Silberschatz [120]. Another data model that deals with the temporal aspects of data is time series, described by Dreyer, Dittrich & Schmidt [62]. Both techniques are only used as a data modeling technique, while event sourcing is a software architecture pattern.

Event sourcing also shares commonalities with stream processing [272], applied in, for instance, Internet of Things (IoT) systems to process sensor events. Events in IoT systems are often used to communicate between different (sub)systems and are not stored as the state of the system. Also, the events represent technical events such as sensor data as opposed to real-world business domain events. Another closely related topic is Complex Event Processing (CEP) as described by Luckham [155]. In CEP the focus is on pattern recognition within a stream of events. CEP itself could be applied in the processing components within an ESS, similar to the event calculus formalism. Event calculus, as described by Sadri & Kowalski [216], is a logical language that represents the effects of events. This language, however, cannot be used to describe event sourcing as an architectural pattern. Similarly, process mining deals with the

analysis of event logs from process-driven systems. The work of Murillas, Aalst & Reijers [171] shows the complexity of mining processes from systems that do not record historical data. ESSs support process mining by default, which makes them suitable for enterprise systems.

Anh et al. [7] describe another append-only data structure: blockchain. While the data structure is similar to event sourcing, the goals of the two techniques are different. A blockchain focuses on solving problems related to distribution, consensus, and trust, while event sourcing solves problems with history, temporal complexity, and audit trails. The blockchain approach enforces the immutability of the data to solve its problems, while in event sourcing this immutability is self-imposed. Event sourced systems could be built using a blockchain solution. However, the distribution and consensus features offered by blockchain do not improve the goals targeted by event sourcing.

3.4 Event Sourcing In Practice

The 25 interviewed engineers have an accumulated experience of at least 35 event sourced systems (ESSs). However, a number of those systems were either not yet in production, or the engineer could not recall enough details of the system. Of the 35 systems, 19 ESSs were discussed in more detail and are summarized in Table 3.2. Still, the experts' experience on all of these systems is reflected in the answers that they gave, and is thus reflected in the challenges, the definitions, and the schema evolution techniques. The categories in this characterization are based on the interviews, and were selected based on the categorization of the concepts deduced from the interviews.

Event sourcing is applied in enterprise applications, either business-to-business or business-to-consumer, as illustrated by the interviews. We did not encounter systems using event sourcing for IoT systems, or other stream processing systems. This reflects the community from which event sourcing originated, which focuses on enterprise applications.

The systems overview shows that the event sourcing pattern is not tied to a particular technology stack. This diversity in technology confirms that event sourcing is indeed a pattern, and not a technology.

3.4.1 Rationale for ESSs

The reasons for applying event sourcing can be grouped into four categories. Remarkably, all systems under study benefit from event sourcing, and no system returned to a current state model. Still, most engineers state that they would not apply event sourcing in every system. The reason given for this opinion is the added complexity of introducing event sourcing. Engineer E2 would apply event sourcing by default, because of the benefits it gives. The different rationales as discussed with the engineers are summarized in Table 3.3.

One of the main benefits of applying event sourcing is the retention of all state changes. According to E24, event sourcing prevents prematurely data deletion: *"as a software developer building a data-driven system and you are modifying data, you are*

System Code	Engineers	Type of application	Technology platform	Rationale	Degree of immutability	DDD	MSA	CQRS
MarketingSys	E22	Marketing automation	.NET, DynamoDB	audit	strict	✓	✓	✓
HealthSys	E23	Health record management	JVM, MySQL	audit, flexibility	cut-off moments	✓		✓
WebBuildSys	E24	Website building	Scala, MySQL	audit	strict		✓	✓
B2CSys	E1	B2C communication	JVM, MongoDB	flexibility	strict			✓
EmailSys	E2	E-mail template management	.NET, MSSQL	audit	strict	✓		✓
LendingSys	E3	Micro-lending	Ruby	flexibility	mutable		✓	✓
ObjectSys	E4, E5	Object registration	JVM, Oracle	audit, flexibility	strict	✓		✓
VideoSys	E6	Streaming video	JVM, EventStore, Neo4J	flexibility	mutable	✓	✓	✓
CMSys	E7	Content management	PHP, CouchDB, PostgreSql	complexity	mutable	✓		✓
PaymentSys	E9, E10	Payment processing	JVM, Groovy, MongoDB, MySQL	trending	mutable			✓
ApproveSys	E13	Approval processing	.NET, RavenDB	complexity	mutable	✓		✓
MeetSys	E15	Appointment management	.NET	flexibility, complexity	mutable	✓		✓
ProjectSys	E17	Project administration	.NET, RavenDB, PostgreSql	audit, flexibility	cut-off moments	✓		✓
IdentitySys	E20, E21	Identity management	PHP, MariaDB	audit, flexibility	strict	✓		✓
P-PaySys	E25	Payment platform	Golang, PostgreSql	trending, flexibility	strict	✓	✓	✓
DocumentSys	E19	Document automation	.NET, MongoDB	audit, flexibility	cut-off moments	✓	✓	✓
Advert1Sys	E8	Classified advertising	JVM, MongoDB	audit, flexibility	mutable	✓		✓
Advert2Sys	E12	Classified advertising	.NET, MSSQL	trending	strict	✓	✓	✓
InventorySys	E11	Inventory management	.NET, LMDB	flexibility, complexity	mutable	✓	✓	✓

Table 3.2: Characterization of the ESSs under study, including the technology platform, the rationale for event sourcing and the chosen degree of immutability. The application of Domain-Driven Design (DDD), the Microservice Architecture (MSA) style, and Command Query Responsibility Segregation (CQRS) is also indicated.

Concepts	Codes
Audit	Regulations (E4, E5, E14, E17, E19, E20, E21); Customer service support (E2, E4, E5, E7, E8, E9, E10, E12, E17, E18, E22, E23); Explanation (E14, E23, E24)
Complexity	Decoupling (E2, E11, E13, E16); Distribution (E1, E3, E6, E12, E13); Temporal logic (E2, E3, E13, E15, E16, E18, E24); Process versus data (E4, E5, E7, E8, E9, E10)
Flexibility	Multiple views on data (E3, E6, E7, E8, E14, E15, E17, E19, E20, E21, E23, E25); Data is not discarded (E11, E24); Data replication (E1, E4, E5, E6, E24); Scalability (E2, E4, E5, E11)
Trend	Experiment (E2, E12, E14, E25); Learn (E1, E9, E10, E25)

Table 3.3: The rationales given by the engineers, categorized in four concepts: audit, complexity, flexibility, and trend.

essentially destroying your older copy of the data. And who told you you're allowed to delete data?" We classified this group of rationale with the category **audit** [1] (9/19 systems). Compliance with regulations (such as system ProjectSys) is one of the reasons in this category. Improving customer support (ProjectSys, Advert1Sys) is another reason. In those systems the state changes are used to explain the system and its behavior to customers. Finally, simply explaining why and by whom data is changed (in debugging scenarios for instance) is given as a reason too (EmailSys).

The second category is **flexibility** [149] (12/19 systems). These systems chose event sourcing (and CQRS), because of the flexibility it provides in the architecture of the system.

Examples of this flexibility are the creation of secondary indexes for search (VideoSys), building and refreshing caches (B2CSys), replacing event queues (MarketingSys, WebBuildSys, LendingSys), and scaling out to multiple read databases (VideoSys). Section 3.5 explains how this flexibility is achieved through the implementation of different *projections* and *projectors*.

The third category is **complexity** [17] (4/19 systems). These applications were considered to contain complex business logic, heavily process driven instead of data driven. Therefore, the architects designed the system as an event-driven system, starting out with the modeling of processes instead of data.

The final category, which is identified as the rationale for three of the 19 systems, is **trending** [39] (3/19 systems). The systems PaymentSys, P-PaySys and Advert2Sys started with event sourcing, because the (lead) architects picked up on a trend. They were curious about the details of the pattern, and started to implement it in the new system. In hindsight, the systems did benefit from this decision, although E9, E10, and E12 ascribe this to luck, and not to the design practices.

3.4.2 Characteristics of Event Sourced Systems

The core category of the GT process is *the process of designing and implementing event sourced systems, as performed by software engineers*. As we needed to make sure that event sourced systems are not a technology but a technology agnostic pattern, we wanted to assure the types of applications and the technology platforms used to realize the implemented systems. Three dimensions, the size of the event store, the workload handled by the application, and the size of the schema, are listed to indicate what kind of systems benefit from event sourcing. These dimensions assure that event sourcing is not biased towards systems of a certain size. Three related topics emerged during the coding process: DDD as a software design approach, CQRS being a related architecture pattern, and the Microservice Architecture (MSA) style. Together with the degree of immutability and the type of application, these different aspects form the characteristics that are listed in Table 3.2.

Event sourcing is a pattern that stores every state change; immutability is thus at the core of the pattern. Helland [105] states that immutability of data is a crucial aspect for distributed systems. Although often seen as the defining characteristic of event sourcing, immutability is not enforced in any manner, as opposed to a blockchain. In a number of the systems under study, immutability is sacrificed for a simpler schema evolution technique (see Section 3.7). We observed different degrees of immutability. The first degree is **strict**, 8 out of the 19 ESSs never change an event. The second degree of immutability is used by 3 out of 19 systems, which allow for **cut-off moments**. In such a cut-off moment, the event store is changed, but back-ups guarantee that no information is deleted. The goal of these back-ups is to satisfy regulations or service-level agreements, therefore, they are kept around forever. This degree of immutability still guarantees an audit trail, because the back-ups can be used to retrieve all the state changes. The last degree level of immutability is **mutable**, 8 out of 19 systems allow events to change. In these systems, the event store is changed on some occasions, and the back-ups are not kept forever. These systems do not satisfy the goal of a complete audit trail. However, the events can still be used to explain how the current state was reached. None of the ESSs lose information regarding the current state of a system. Events that are changed, or transformed, are in most cases changed because of technical reasons.

In 14 of the 19 ESSs under study DDD is used as the design approach. DDD is an approach to software development that aims at *tackling complexity in the heart of software* (as the subtitle of the seminal book by Evans [74] states). DDD focuses on the explicit modeling of the domain, including its boundaries and events. However, only four of the 25 engineers argue that DDD is a prerequisite for event sourcing. Although the other engineers do not see DDD as a prerequisite, without a doubt DDD inspired the design of many ESSs. Events, as expressed by E11, *"should represent real-world business events"*. This is different from transactional processing, or stream processing. In those systems events can have a more technical nature. According to E11, en ESS that contains events not representing real-world business domain events will undergo more changes to the software. E11 explains: *"You align the events with real-world events, so you are dealing with changes that have a native equivalence. Doing DDD leads to a less fragile design."* For E16, the understanding of the domain is a prerequisite for

doing event sourcing: *"A high level of maturity of the domain knowledge is a prerequisite. When the domain knowledge is still evolving, applying event sourcing introduces more risk."*

CQRS is a closely related pattern that also originated from the community around the DDD approach (the pattern itself will be explained in more detail in Section 3.5). Although engineer E14 has seen a few solutions that apply CQRS without event sourcing, they are almost always used together. All of the systems that we discussed with the engineers applied both CQRS and event sourcing. The interviews give no explanation for this co-appearance. A possible explanation, based on the experience of the authors, could be the fact that they are often 'advertised' together in the community.

Also closely related to event sourcing is the MSA [61] style. Similar to DDD, the MSA style also attacks the complexity of large software systems. This is confirmed by 8 of the 19 systems that were discussed in the interviews. They implement microservices to break up a large application and control complexity by spreading the business logic over these services. We observed two approaches in the systems that combine MSA and event sourcing. The first approach uses event sourcing as an implementation detail of the microservices. In the second approach, the events are not only used to store state changes, the event store is also used to communicate these events between microservices.

Unfortunately, the experts could not uniformly report on event store size, traffic, and schema size of the characterized ESSs. Some of them could not disclose these details due to commercial reasons, while others no longer had access to the discussed system. Table 3.4 summarizes the details that were reported per discussed system. The systems have a size ranging from smaller than three gigabytes, up to 250 gigabytes (or more than a billion events). Eleven systems (including HealthSys that reports a growth rate of 4 million events per day) have more than a million events in the store, representing more than half of the systems. Two systems (WebBuildSys and InventorySys) even report sizes over a billion events. Advert2Sys shows a small event store size, but that is due to the active pruning that they do. The growth of 4 million events per day shows that the total number of events is much higher than the reported five million. The growth rate of the systems shows that a number of systems report a growth that passes a million events per day (HealthSys, Advert2Sys, and InventorySys), but most show a number far less than a million new events per day. The schema sizes show that none of the reported systems passes the 500 event types, but is rather spread out between 20 and 450 types. In general, Table 3.4 shows a wide variety of event store sizes, handled traffic, and event store schema sizes. Systems VideoSys, PaymentSys, ApproveSys, and InventorySys show that ESSs are not only used for small business domains. The event store size shows that event sourcing can be used for both small and large systems.

System Code	Event store size	Growth of the store	Schema size
MarketingSys	⩾ 50,000,000 events	10,000 events per day	
HealthSys		4,000,000 events per day	
WebBuildSys	More than 100,000,000 active sites, every site owns hundreds or maybe even thousands of events		A single event stream type per site
B2CSys	⩽ 5 gigabyte		50 event types
EmailSys	*"It is probably approaching the half a million events mark by now"*	⩽ 50 or 60 events per day	
LendingSys	*"We processed I think half a million account transactions"*		6 microservices
ObjectSys	200,000,000 events		50 event types
VideoSys	7,000,000 events		400 event types
CMSys	⩽ 3 gigabyte		
PaymentSys	5,000,000 events		300 event types
ApproveSys	1,000,000 events	100,000 events per 2 months	300 event types
MeetSys	100,000 events	1,000 events per day	20 event types
ProjectSys		10 events per minute	
IdentitySys	50,000 events		20 - 30 event types
P-PaySys	*"I don't think our scale is particularly high"*		20 - 30 stream types
DocumentSys	5,000,000 events	1,000,000 events per month.	100 event types
Advert1Sys	50,000,000 events	60,000 events per day	115 event types.
Advert2Sys	5,000,000 events (active event store)	1,000,000 events per day	50 event types
InventorySys	1,100,000,000 (250 gigabyte)	77,000,000 events per month	450 event types

Table 3.4: The size and growth of the event store and the schema size of the ESSs under study, as reported by the experts. Empty cells represent unknown data points.

3.5 Event Stores and Event Sourced Systems

This section defines key concepts and operations in an event sourced system (ESS). These definitions are based on our experience building ESSs, and confirmed by the interviews that were conducted. They are used to conceptualize event sourcing and the identified challenges. When coding the interviews, different characteristics and variability of the concepts and operations were identified, which are described in this section. These concepts and operations should be used in discussing, and teaching ESSs.

3.5.1 The Event Store

We propose the following definitions for the concepts and operations related to an event store. First the concepts are defined, starting with events and working all the way up to the store. After that the operations on the event store are given.

Event. *An event is a discrete data object specified in domain terms that represents a state change in an ESS.*

An example of an event from the Netflix case [9] that represents a real world business event is given in JSON format:

```
{
  "LicenseCreated":
  {
    "customerId": "BlackMirror",
    "titleId": "TheNationalAnthemS01E01",
    "date": "2014-01-06"
  }
}
```

The importance of the relation to the business domain is stated by E5: *"business analysts are telling us what the events should be."* E11 adds: *"you capture business changes as a flow of events, you align these events with real-world events."* A more general definition is given by Michelson [170]: *"a notable thing that happens"*. It lacks the relation to the business domain as it is used for event-driven architectures in general. The data in the events can be stored in different formats such as JSON, XML, AVRO [245], or Protobuf [95]. Events are stored in a sequence, in event streams.

Both E14 and E25 do see a distinction between *internal* and *external* events. Internal events are fine-grained and contain more detail, while external events are more coarse grained and meant for other systems to communicate. Through this distinction it is possible to hide internal business logic from external consumers. Multiple engineers (E12, E14, E16, E17, and E22) also acknowledge the usefulness of state propagation through events. Instead of events that mark a business event, events can also be used to simply propagate the state of an object.

Event Sequence. *Every event is stored together with a sequence number. Its sequence number represents the position of the event in the stream.*

Event Stream. *An event stream* ***s*** *is a sequence of tuples, each tuple containing an event and its sequence number*

$$s = \langle (\mathrm{e}_1, 1), (\mathrm{e}_2, 2), ..., (\mathrm{e}_n, n) \rangle$$

The sequence numbers are consecutive natural numbers, starting with the number 1.

The sequence numbers are not handed out by the event stream, but are supplied by the producer of the new events. The event stream does validate if the sequence numbers are consecutive natural numbers. E3 explains how this is used by event subscribers: *"you get this monotonically increasing sequence of events that you can use to record your position."* The streams together are stored in the event store.

Event Store. *An event store is a set of event streams. These streams form the partitions of the event store, and are disjoint.*

The event store has two foundational operations on event streams: $read$ and $append$. The $read$ operation enables systems to read an event stream from a given sequence number. Events are appended to the event stream with the operation $append$. E20 explains how append is the only operation that changes the event stream: *"I only append new events, and never throw away old events."* The $append$ operation has an

extra validation: the caller should supply the sequence number for the new event, which is validated and an error is returned if it is not the expected number. Through this validation the store achieves *optimistic concurrency control*. According to engineer E24, this is the strongest guarantee that the event store should offer. A caller will first need to $read$ from the event stream, before $append$ can be called. If another caller calls $append$ in between, the $append$ of the first caller will fail, because the highest sequence number has changed.

Both the $read$ and $append$ operation operate on single streams, this emphasizes the fact that the streams in an event store are disjoint. The $append$ function can either append a single event, or multiple events, depending on the implementation. For instance, Event Store [76] implements the $append$ function with a version that atomically appends multiple events to the stream.

3.5.2 The Event Sourced System

Enterprise software applications support at least two foundational use cases: storing information and retrieving information. The event store is used to store the state changes in the system, however, the event store is not optimized for retrieving information.

In ESSs the $project$ function is central in both storing new information and retrieving information. First we define and characterize this $project$ function. Second we discuss storing and retrieving information by presenting two parameterized operations.

Project function. *The project function takes one or more event streams and creates a projection with the data from the given events. The projection itself can take different forms, for instance, it can be a relational database that is updated through SQL statements, or a search index manipulated through the filesystem.*

The $project$ function operates on one or more event streams. The event streams are disjoint, and the $project$ function thus can not assume an order between the events from the different streams. While the order of events in a single stream is guaranteed, the events from different streams have no relation.

The projection that is built by the $project$ function in an ESS is similar to the concept of projections in relational algebra [52]: projections contain a selection of the data present in events. Projections are similar to *views* in a relational database: a selection and transformation of one or more database tables.

Projections. *A projection π is a selection of the data stored in events, transformed into a specific model. The selection and transformation depend on the purpose of the projection. The data in a projection is transient, a projection can be rebuilt from its source events at any point.*

Examples of different variations of projections are frequently given by the engineers. Engineer E6 for instance explains how they project the event data to both Neo4J (a graph database) and ElasticSearch (a document database). The graph database serves the navigation through the data, while the document database serves the search functionality. Other examples given are a specific storage technology for indexing (used, for instance, by E8, E12, E23), an analysis to report the abuse of accounts information, and a relational table with all issued licenses for downloaded content.

The primary design question of the $project$ function and its target $projection$ is its purpose. The importance of the $project$ function lies in its encapsulation of the variability in storage technology, data selection, and data model. Choices can be made per project function, which enables a huge potential for optimized projections for their purpose. The flexibility as a reason for choosing event sourcing (Section 3.6) is in large part caused by the $project$ function.

The $project$ function also poses a risk to the performance of the system, a challenge we discussed in Section 3.6. The time it takes to build a projection depends on two factors: the number of events that are read and the time it takes to update the projection. Engineers E11, E13, and E14 discuss their search for improved implementations of projectors. Quick improvements can be found in faster storage technology, or better use of hardware. Engineer E12 explains how they prune the event stream by moving older events into a different stream. This pruning decreases the number of events that the $project$ function needs to process, making the rebuilding faster. Engineer E14 discusses how they plan the rebuilding in weekends, rather than investing developer effort in optimization.

The retrieval of information from the event store is done by building a projection. Queries are answered using the data available in the projection. Projectors can build the projection on-demand, or opportunistically: the given projection is build first and then the specific query is answered. However, it is also possible to pre-build the projection: the projector constantly watches the event streams and updates the projection whenever new events arrive. This decision depends on the ratio between reads of the projection and new events being appended to the stream. If a projection is read infrequently, it is unnecessary to constantly project new events, and thus consume resources. However, if a projection is read frequently projecting the new event directly on arrival improves the performance of the query.

The behavior of the projector is similar to that of the higher-order function *fold* [115], a recursion operator that works on lists, as stated by Meißner et al. [164]. The projector *folds* over the specific event streams and creates a projection. The integration of functional programming and domain-driven design is further explored by Wlaschin [268].

Storing new information is done using the $append$ operation. The $append$ operation is the only operation that is capable of storing new events in the store. However, before storing these new events, they have to be produced. Events in an ESS are produced as a result of an action (the commonly used name is *command*) that is accepted by the system. The validation, resulting in an acceptance or rejection of the command is done by the $accept$ function.

Accept function. *The accept function takes a projection π and a command c. The command is validated using the data in the projection, and the accept function either results in an error or in an event.*

The command follows the *Command pattern* described by Gamma et al. [90]. The system first builds a projection, and then validates the command using the $accept$ function. Validation of the command can result in either a new event or an error (in case of a validation error). The new event is appended to a specific event stream, which is selected based on properties present in the command. This appended event

is the new information stored in the system. While the projection is built in order to validate the command, it is only used to validate the command and is volatile.

A command can only affect a single stream, because the $append$ operation appends to a single stream. To guarantee the consistency of information, the system should not append events to two streams in one request. One append might fail, leaving the system in an inconsistent state. This rule increases the importance of the design of the schema of an event store.

3.5.3 The Schema

An event store contains no schema for the specific structure of events. The data schema is not explicitly defined at all, but is implicitly encoded in the ESS. The knowledge of the data schema inside an ESS is encoded in the source code of the $accept$ and $project$ functions. This is similar to other systems with a so-called implicit schema [86], such as document-oriented data storage systems.

In general, events can take any form and thus the schema as well, therefore, we left these definitions abstract on purpose. However, we believe that these abstract definitions can be used to support the discussion of schema evolution, as we show in Section 3.7. This section defines event, event stream, and event store schemas, along with the *conforms* relation.

Event Schema. *An event schema ε describes the type and form of events. conforms(e, ε) holds if event e conforms to the specification ε.*

An event schema could be implemented by, for instance, XML Schemas or AVRO [245]. The latter uses the schema not only for validation, but also for serialization to a binary format. Two other options that can be applied to create a more formal event schema are domain-specific languages (suggested by E11 and E14) and strongly typed classes (see Table 3.5).

Event Stream Schema. *An event stream schema ς describes an event stream and the events that can occur in the stream. The event stream schema contains the event schemas of the events that can occur in the stream, along with the patterns of occurrence. conforms(s, ς) holds if event stream s conforms to the specification ς.*

An event stream schema contains both the specification of the events, and the specific patterns. An example schema contains both the schema (or specification) of the 'registered' event, and the fact that the 'registered' event occurs before a 'checkout' event.

Event Store Schema. *An event store schema θ describes an event store and the streams that are stored in the event store. conforms(es, θ) holds if event store es conforms to the specification θ.*

The event store schema contains more knowledge than only the event stream schemas, similar to the event stream schema. For instance, the cohesion between streams can also be specified in the event store schema. An example of this is that when a specific stream contains a certain event, another stream should exist and be present in the event schema. An explicit implementation of event stream schemas or event store schemas was not encountered during the interviews.

3.5.4 Event Sourced Systems based on CQRS

As we have seen in Section 3.4, every ESS under study also applies CQRS. CQRS was introduced by Young [275] and Dahan [50], and the goal of this pattern is to separate actions that change data (commands) from requests that ask for data (queries). Although event sourcing and CQRS can be used separately, the common application of the two patterns is worth exploring. Based on literature and the interviews an example architecture combining event sourcing with CQRS is discussed. This architecture is shown in Figure 3.1. As illustrated, the event store schema θ is part of the ESS: the event store conforms to it, and the command and query system encode it in their application logic.

In the command system $aggregates$ (as introduced by Evans [74]) are used to process incoming commands (1). Commands are routed by the $commandhandler$ to the correct aggregate. Aggregates will process the commands using the $accept$ and $append$ operations. First the existing events are read (2), a projection is built (3), and then the $accept$ function is called. When the command is accepted, the resulting event will be appended to the event stream (4).

An aggregate reads a specific event stream, to which the new event is also appended. Often the aggregate will be the owner of the event stream it reads and appends to. As a benefit, commands sent to different aggregates can be processed concurrently without interfering. E6 describes a solution where multiple aggregates use the same stream. This variation is used to share generic behaviour among aggregates; it is mixed with more specific logic.

In the query system, $projectors$ are used to build $projections$ that can be used to return information to the sender. Queries are routed by the $queryhandler$ to the correct projector (5), depending on the specific purpose of the projector (such as browsing or searching). The projector will retrieve the requested information from its projection. First the events from the event streams will be read (6), then the projection will be built (7).

While queries can be handled by building the projection on-demand, most ESSs based on CQRS will update the projection as soon as new events are appended. In that scenario, step (6) and (7) will be executed before (5), and the projector can immediately use the projection to handle the query. This decision is based on the ratio between events and queries. When there are few queries, and many events, prebuilding the projection takes up resources (such as storage). If the workload consists of more queries, building the projection ahead of time results in faster response times. E24 describes a flexible approach that merges the two approaches in an on-demand fashion. The sequence numbers of events are used as checkpoints and allow the projectors to track which events are already processed. The immutability of the event store is crucial for these projectors. If events or their ordering are mutated, the checkpoint has no value and the projector needs to re-read the event streams and rebuilt the projection.

Most pre-built projectors are *eventually consistent*. As Vogels [261] explains, the ESS guarantees that if no new commands are processed, eventually all queries will return the last updated value. However, because there is time between the acceptance of a command and the updating of a projection, a query might return an older value. The

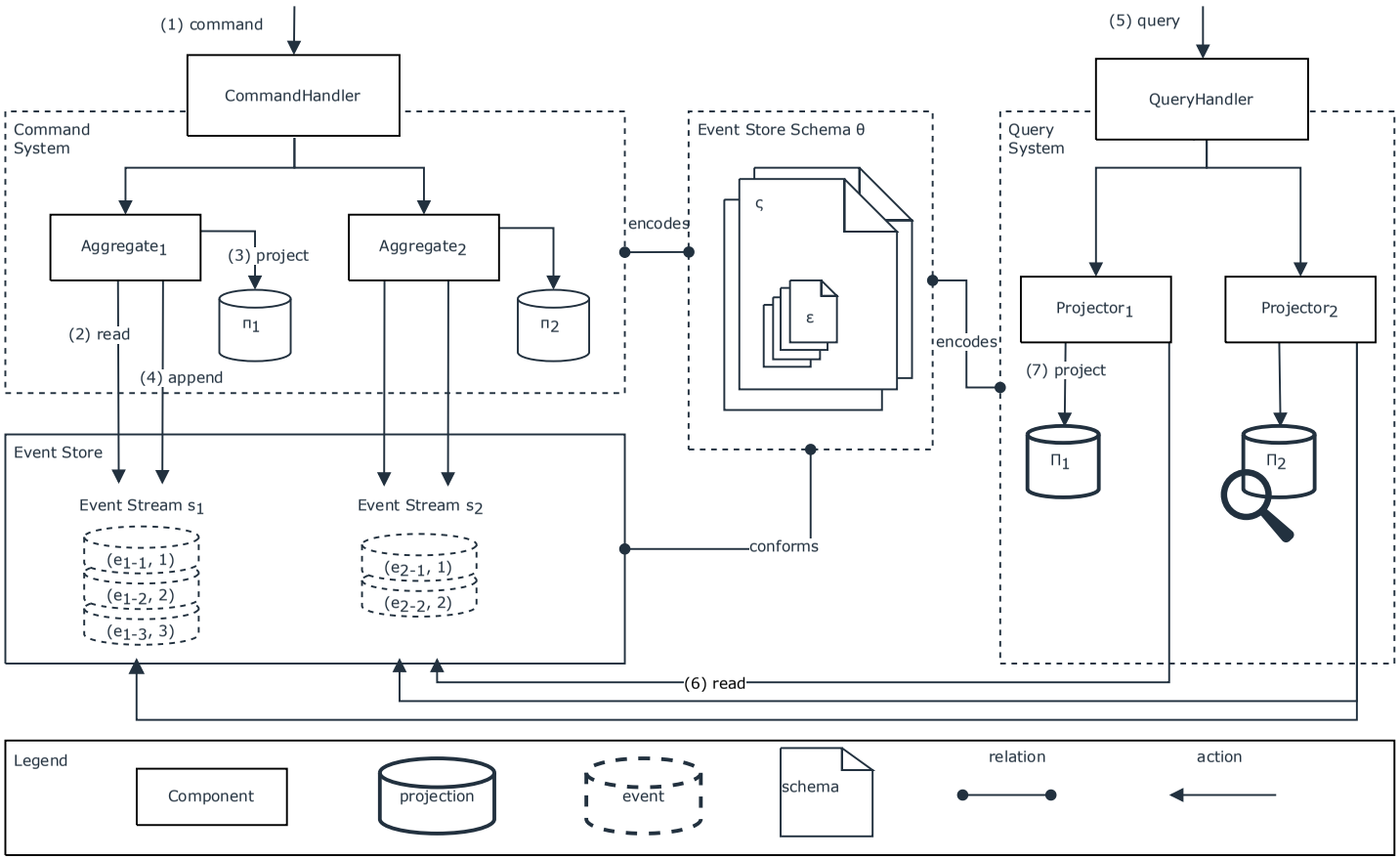

Figure 3.1: An event sourced system based on CQRS. The event store conforms to the schema θ, which is encoded by the command and query systems. The command system validates commands using the events. These same events are read by the query system to build the projections, which are used to respond to queries.

Concepts	Codes
Event Store	Business events (E5, E11); State propagation (E12, E14, E16, E17, E22); Monotonically increasing sequence number (E3); Append only (E1, E2, E16, E17, E20); Optimistic concurrency control (E24); Internal versus external (E14, E25)
Event Sourced System	Projector variations (E6, E8, E12, E23); Optimization of projecting (E11, E12, E13, E14)
Schema	Domain Specific Languages (E11, E14); Strongly typed classes (E2, E3, E4, E5, E17)
CQRS: Projections	Synchronous (E2, E20, E21, E23); Opportunistic (E24); Independent (E16, E17)
CQRS: Aggregates	Multiple on one stream (E6); Snapshots (E2, E20, E21); Instance versus type (E14, E25)

Table 3.5: The concepts and codes extracted from the interviews related to the implementation of CQRS based ESSs.

duration between (4) and (7) is the so-called inconsistency window: the command system and the query system do not share a consistent state. Eventual consistency was also listed as one of the challenges in ESSs and is discussed in Section 3.6.

Four engineers explain how their projectors share a database transaction with the aggregates. This allows them to achieve immediate consistency, because both the event as the projections are committed in a single projection. In those systems scalability is sacrificed for immediate consistency. This implementation technique results in *synchronous* projections.

Table 3.5 summarizes the different concepts and codes that were extracted from the interviews. While the definitions are mainly based on our experience in building an ESS, we have used the data extracted from the interviews to scope our description. The concepts and codes discussed by the engineers determined what specifics were described.

3.6 Challenges Faced in Applying Event Sourcing

A pattern description without discussing the consequences is incomplete, and would lead engineers astray. While Section 3.4 discusses the positive consequences that engineers experienced, they also discussed the negatives in the interviews. In this Section we discuss five challenges experienced by the engineers with two goals in mind: (1) to indicate to practitioners what the limitations of the pattern are and (2) to formulate novel research topics for future research around the pattern. The first two challenges are addressed in more detail by two of our contributions in Section 3.5 and 3.7. The summary of mentioned challenges by engineers is listed in Table 3.6.

Challenge	Codes
How can Engineers better be Supported in Learning how to Apply the Event Sourcing Pattern?	Eventual Consistency (E1, E2, E14, E24); Events versus state (E1, E2, E4, E5, E6, E12, E13, E15, E16, E17, E20, E21, E23, E25); Lack of knowledge sharing (E1, E3, E9, E10); Start is slow (E4, E5)
How can Tools, Frameworks, and Platforms be Provided to Make the Pattern even More Successful?	Immature tools (E2, E9, E10, E17, E20, E21, E25); Frameworks not properly maintained (E6, E19, E22); Pattern versus framework (E6, E24); Tools not accepted by operations (E14); Frameworks hide details from developers (E24); Frameworks help beginners (E6, E7, E13, E14, E17, E24)
How can Projections be Optimized?	Rebuilding is slow (E4, E5, E8, E9, E10, E13, E20, E21, E22, E25); First in-memory (E8, E24); Targeted rebuilds (E8, E9, E10, E16, E17, E20, E21, E23); Rebuild versus developer time (E1, E2, E6, E7, E9, E10, E11, E13, E14, E15, E20, E21, E22)
How can a System that Uses Event Sourcing Protect User Privacy?	Separate events from personal information (E20, E21); Remove (E11, E23); Anonymization (E11, E25)
How can Event Stores be Evolved?	See Table 3.7

Table 3.6: The challenges faced by the practitioners while implementing ESSs.

How Can Engineers Better Be Supported in Learning How To Apply the Event Sourcing Pattern?

The most prominent category of challenges mentioned by the engineers is in the area of designing software. Designing ESSs is more difficult than other systems, because of two characteristics. In the experience of 13 of the 25 engineers, thinking in events and state transfers is completely different from thinking in current state and database transactions. Section 3.5 proposes a description that improves the understanding, and supports the teaching of event sourcing and event sourced systems (ESSs).

An ESS introduces not only events and state transfers. Eventual consistency forces developers to let go of guarantees that they would have in a system using current state and synchronous processing. In a CQRS system, an update sent through a command will not immediately be reflected in the result of a query. The system first needs to process the event into one or more projections. Engineer E12 states that *"a lot of developers had to get used to information not being in place"*, and E2 adds that *"getting people to understand eventual consistency is the biggest hurdle."* Eventual consistency forces developers to rethink the basic interactions of the user with the system.

We give two examples of interactions that force developers to rethink system design. The first example is that of the expectation of users to retrieve data that they previously submitted into the system. In a CQRS system, the query system might not directly return the data that was submitted through a command. The user interface of the system should make it clear to the user what is going on, or even try to hide the fact that the system is eventually consistent. The second example is that of developers that more or less have the same expectation. Often developers try to use the result of the

query to make decisions in an aggregate. However, the query system might not have processed all events and missed recent updates. If developers overlook this principle, the decisions lead to bugs in the system.

How Can Event Stores Be Evolved?

Both E13, *"we dreaded the upgrading, we had some fear in advance"*, and E22, *"versioning in event sourced systems is a big problem"*, point out the perceived difficulty of upgrading ESSs. This challenge did not come as a surprise, our earlier work [186] and the work of Young [277] underline this. During the interviews we identified five fundamental techniques for schema evolution in ESSs, which are described in Section 3.7.

How Can Tools, Frameworks, and Platforms Be Provided To Make the Pattern Even More Successful?

Eight engineers discuss the lack of standardized tools, such as frameworks, platforms, and databases. A commonly stated opinion within the community is that you do not need frameworks to implement an ESS. However, engineers E9, E10, E17, E20, E21 and E25 state that they wish to see more mature libraries and frameworks. Engineers E6, E17, E19, and E22 mention that infrastructure and tooling for ESSs is immature. Either the tooling does not support a broad enough set of scenarios, or the quality is lacking. How large the market is for specialized event sourcing tools is difficult to say. Recently AxonIQ [10] has started to offer commercial support for ESSs, similar to what Event Store [76] does.

How Can Projections Be Optimized?

Projections, as discussed in Section 3.5, are used to retrieve information from the system. Rebuilding projections, however, can become a bottleneck for ESSs.

Engineers E11, E13, and E14 discuss their search for improved implementations of projectors. Quick improvements can often be found in faster database technology, or better use of hardware. Although rebuilding projections needs planning, engineer E14 discusses how they rather plan the rebuilding in weekends, instead of investing developer effort for optimization.

Engineer E16 explains how the domain can show an optimization: not reading all the events on a rebuild. Often the older events are no longer reflected in the projection, because the specific data (such as a classified advertisement) is no longer active.

Another important implementation detail that lifts some of the burden is that projectors can (and must) be implemented as independent, autonomous processes. This gives the system the possibility to only rebuild those projections that need to be rebuilt, instead of all the projections at once.

How Can a System That Uses Event Sourcing Protect User Privacy?

Privacy regulations, such as the GDPR, are designed to protect users from being taken advantage of. Personal information should not be kept in a system for all eternity, but the system should delete it whenever someone requests that. However, such a requirement conflicts with the nature of event sourcing: retaining all the data. Engineers E20, E21, E23, E25 mention that they designed their systems to comply with these regulations. Systems HealthSys and P-PaySys use some form of anonymization and removal of information to comply. Obviously, this requires them to rewrite events.

System IdentitySys takes a completely different approach. The system separates the events and the personal information in two different stores. When events are read, they are supplemented with personal information. If that information is no longer present (because of removal requests), default values are supplied.

3.7 Schema Evolution in Event Sourced Systems

A challenge discussed by multiple engineers is the evolution of event sourced systems (ESSs) (as stated in Section 3.6). From the transcripts, we identified five fundamental techniques for schema evolution. These event schema evolution (ESE) techniques are described using the definitions given in Section 3.5.

We encountered two reasons why event schema evolution in ESSs is difficult. First of all, the implicit schema (as described by Fowler [86]) makes evolution in ESSs difficult. Solutions as proposed by Meurice, Nagy & Cleve [168] and Maule, Emmerich & Rosenblum [161] to analyze the impact of schema changes are not usable, because there is no explicit schema. In contrast to their solution, the change originates in the application and impacts the data in the event store. This makes the direction of the impact different from theirs.

The second difficulty in event schema evolution is the immutability of the event store. Traditional solutions to transform or rewrite the store are not always possible. However, the benefits of immutability in event stores (as listed in Section 3.4) are not always requirements. The different degrees of immutability, as shown in Table 3.2, allows for different evolution techniques.

Teams that apply event sourcing without a clear understanding of the business domain introduce risk, according to E14, E16, and E22. E22 explains that the challenge of evolution is exactly why it is preferred to start a new system without event sourcing, and only introduce event sourcing when the domain knowledge is stable: *"once we have enough trust in our model we will transform to event sourcing."* As E16 confirms, events based on a sufficiently clear domain knowledge will decrease schema evolution.

Another prevention technique is the cleaning up of events in the event store, of which we encountered two possibilities. First of all, older events that no longer represent active information can be moved into *cold* storage. These events can still be read and processed, but are no longer processed by the ESS itself. Therefore, they do not have to conform to the implicit schema of the ESS. Second, sometimes these events can be kept in the event store itself, but the ESS will never read them. Again, this makes it possible to ignore those events on upgrades.

Event schema evolution that cannot be prevented can be solved by the following five evolution techniques. Although in our work [186] we also discuss five techniques, during the interviews a different set of techniques was encountered. The technique *lazy transformations* was not mentioned by any of the engineers, while *weak schema* was mentioned as a new technique. Which techniques are used by which engineers, and the benefits and liabilities per technique given by the engineers during the interviews are classified in Table 3.7. In some cases the liabilities are also from engineers that do not apply the particular technique: they stated the liability as a reason for not

Technique	Engineers	Benefits	Liabilities
Versioned Events	**2**: E7, E19	Simplicity of implementation (E19)	Application logic pollution (E7, E9, E16)
Weak Schema	**11**: E2, E7, E8, E11, E14, E15, E16, E17, E20, E21, E22	Simplicity of implementation (E2, E8, E11, E15, E17, E22)	Application logic pollution (E9) Feature incomplete (E8, E15, E17)
Upcasters	**12**: E1, E4, E5, E7, E11, E12, E13, E14, E16, E19, E23, E24	No application logic pollution (E19) Strict immutability (E24) Simplicity of implementation (E14)	Decrease of run time performance (E11, E23) Multiple schemas (E23) Complexity of implementation (E23)
In-Place Transformation	**5**: E8, E9, E10, E13, E23	Ad-hoc evolution (E8, E9, E10, E13, E23) Single schema (E13)	Mutability of events (E22) Complexity of implementation (E13) Decrease of evolution performance (E24) Risk of data-loss (E8)
Copy-Transform	**14**: E3, E6, E7, E8, E9, E10, E11, E13, E14, E15, E17, E19, E22, E23	Simplicity of implementation (E6, E13) Strict immutability (E15, E17, E19) Ad-hoc evolution (E3, E6, E17, E23)	Mutability of events (E11, E16, E22) Decrease of evolution performance (E6, E24)

Table 3.7: Benefits and liabilities of event sourcing evolution techniques.

using the technique.

ESE Technique 1: Versioned Events

Given an event store es conforming to a schema θ, the technique *versioned events* transforms the schema into θ' such that

$$\mathit{conforms}(es, \theta') \wedge \forall \varsigma \in \theta : \exists \varsigma' \in \theta' : \varsigma \subseteq \varsigma'$$

This technique introduces only new types of events, and does it in such a way that the event store es conforms to θ' without transformation. The $project$ functions that process the involved streams are required to handle these new events.

FINDINGS This technique is applied by engineers E7 and E19, with the sole benefit that it is a simple technique that does not require specific changes to the ESS. The liability of this technique is the pollution of application logic, as stated by E16: *"I try to keep my domain abstraction pure. My v1 and v2 version of the event do not enter the model together."*

ESE Technique 2: Weak Schema

With this technique the events are described in a minimalistic manner. Similar to technique 1, the event store es or the schema θ are not transformed during evolution.

Evolution operations that are allowed with this technique are limited to transforming the event e into e' such that it still conforms to the event schema ϵ. This requires the $project$ operation to handle this variability.

RELATED WORK This technique is described by Daigneau [51] as the *tolerant reader* pattern. Serialization formats such as Protobuf by Google Inc. [95] and AVRO by The Apache Software Foundation [245] support this technique by reading the existing binary data into the new version of the objects.

FINDINGS Eleven engineers apply this technique, because of the simplicity. The limitations of this technique are stated as a liability, together with the pollution of the $project$ operation that is required (E9 explains: *"you want to assume a certain event schema"*).

ESE Technique 3: Upcasting

This technique is well known to event sourcing practitioners and described by Betts et al. [15]. The event streams are transformed into streams conforming to the latest schema by a new function: the $upcast$ function. This function is called before the streams are passed into existing $project$ functions. The transformation is centralized in this new function, which improves the maintainability of the system.

For the $project$ functions it appears that little has changed, it appears that the relation *conforms*(es, θ') holds. However, events already stored in es still conform to θ, while newly appended events conform to θ'. After appending new events to es, the store itself will neither conform to θ or θ'.

RELATED WORK The technique is similar the pattern *message translators* as described by Hohpe & Woolf [111].

FINDINGS Twelve engineers use upcasters, claiming benefits such as no domain pollution, the immutability of events, and simplicity of implementation. One of the stated liabilities is a decrease in performance: *"If you have been running upcasters for a long time, you will have quite a stack of them in place, which slows down the entire loading."* Other liabilities are added complexity in analyzing the event store, because it contains events that conform to different schemas.

ESE Technique 4: In-Place Transformation

This technique updates events to resemble the new schema, and thus forces ESSs to forgo immutability. New operations that alter event streams need to be introduced, such as $insert$ (insert an event at a specific position) and $update$ (update the event at a specific position). These operations break the immutability of the event store, with the consequence that cached projection need to be rebuilt. Therefore, two available event stores, EventStore Event Store [76] and AxonDB AxonIQ [10], deliberately do not offer these operations.

RELATED WORK This technique is similar to migration scripts for relational databases. Scherzinger, Klettke & Störl [224] and Saur, Dumitraş & Hicks [223] both propose a similar approach to evolve data in a NoSQL store. The lazy migration (on data access) is similar to *incremental migration* as described by Sadalage & Fowler [215].

FINDINGS Four systems, HealthSys, PaymentSys, ApproveSys, and Advert1Sys, apply this technique. Benefits are the possibility of ad-hoc fixes, and improved reasoning because the store will only contain events conforming to a single schema. However,

the risk of making errors, the loss of immutability, and the performance are stated as liabilities. E22 explicitly prevented this technique from being used: *"to prevent this technique we first zipped the events, and then encoded the result before storing them."*

ESE Technique 5: Copy-And-Transform

During the execution of this technique, existing streams are processed and new streams are created from transformed events that conform to the new schema. This does not violate the immutability of the source events, but creates new events instead. Existing projections are still valid, although they do need to process new streams to receive new events.

RELATED WORK Young [277] describes this technique as *copy and replace*. The *parallel universe* of IMAGO, as described by Dumitraş & Narasimhan [64], is similar to this technique. QuantumDB, created by Jong & Deursen [130], uses *ghost* tables to apply this technique in relational databases. Copy-and-transform of a complete event store could be seen as an ETL process that creates a new store.

FINDINGS Fourteen engineers have used this technique, either to transform specific streams or a complete event store. As E6 states, this technique is relatively simple to implement, because *"we can do literally anything we want."* The data preservation is stated as a benefit, as well as the fact that this is a one-time operation. The performance of this operation is a liability, transforming a large store takes a considerable amount of time.

The data discussed in Table 3.7 does not allow us to discuss how techniques are combined within a single system. It does allow us to discuss how engineers have experienced and applied different techniques over the course of working on multiple systems. We can observe the following from the discussed engineering experiences:

- No engineer has solely applied *versioned events* or *in-place transformation*, those techniques are clearly used in combination with others.
- Five engineers have solely applied *upcasters*, which corresponds with the general advice we found in the grey literature and community.
- The *copy-transform* technique is mostly used in combination with other techniques, only two out of the fourteen engineers have solely applied this technique.
- Four engineers have considered techniques, but opted not to apply them: E9 considered *versioned events* and *weak schema*, E16 considered *versioned events* and *copy-transform*, E22 considered *in-place transformation*, and E24 considered *copy-transform* and *in-place transformation*.

We conclude that the techniques are not exclusive: almost all engineers have used multiple techniques and applied multiple techniques in a single system. Example combinations mentioned in the interviews are

- The application of *upcasters*, with *copy-transform* to clean up the upcasters when there are too many.
- The application of *in-place transformation* for quick patches, while a different technique is used for planned evolution.
- The application of *weak schema* for simple evolution steps, while a different technique is used for more complex evolution.

From the study we formulate the following advice:

1. *Versioned events* and *weak schema* are the simplest techniques to implement. Systems should start out with those techniques.
2. When evolution operations cannot be handled by the first two techniques, systems can apply *upcasting*. This retains the immutability of the event store.
3. Only when a decrease of performance or maintainability is experienced should systems apply *copy-and-transform*.
4. *In-place transformation* should only be used by those systems that do not require immutability or an audit log.

The techniques form a range of possibilities to evolve the event store of an ESS. All techniques, with one exception *in-place transformation*, can be applied in an ESS that follows the definition given in Section 3.5.

3.8 Discussion

One could wonder whether another research approach would have been equally successful in extracting architecture knowledge about the event sourcing pattern. We have looked at open-source systems such as AxonIQ [11], Event Store [76], NEventStore Dev team [174], Prooph Components [199], and observed that these follow the pattern and guidelines as discussed in this article. However, aspects such as the rationale and consequences of using the pattern are impossible to extract this way. This research is also similar to a study with multiple cases (Flyvbjerg [81]), although one would expect a more extensive extraction of information about the case (i.e., system) and its context in a multiple case study. We would have had to use more research resources, but perhaps we would have also been able to provide more code examples of how the pattern was implemented. Finally, design research (Sein et al. [229]) could have also been used to extract the pattern description. While the description would perhaps have been less extensive, there would have been more focus on the evaluation and validation of the pattern and its description. We consider this last aspect as future work, even though we are convinced that the incremental nature of this research has led to a pattern description that is reusable and useful for architects.

Our pattern description itself does not follow a specific format. We decided to structure our presentation according to the concepts that emerged from the GT, and not according to a specific pattern description format. We did, however, use the examples of Gamma et al. [90] to evaluate the completeness of our pattern description.

Gamma et al. state three essential elements besides the **the pattern name**: the **problem**, the **solution**, and the **consequences**. The problem describes what the context is of the pattern, and when to apply it, which we have summarized in Section 3.4. The description of the pattern, the solution, is covered in Section 3.5. Finally, the consequences are split into two sections: Section 3.4 covers the positive consequences by linking them to the problems that are solved. Section 3.6 covers the negative consequences by stating several research challenges for future work.

The format that Gamma et al. use to describe patterns consists of thirteen different sections. While these sections cover the four essential elements, the *related pattern*

section should be discussed on its own. The design of a software system is never the application of a single pattern, but rather the combination of different patterns that together form the design. This is no different in ESSs. Section 3.5 recognizes this, and explains the combination of event sourcing in CQRS in great detail. The relation to other patterns to solve the specific challenges of schema evolution are covered in Section 3.7.

A second question that must be asked is whether academic fora are the optimal place to publish patterns. As whole books have been written about particular patterns and as patterns appear to have a certain shelf life, one could wonder whether patterns should be published in academia at all. We argue, with this article, that some patterns are too important to ignore (SOA, Client-Server, Event Sourcing, etc.) and that these deserve specific detailed attention from academics. We find the strongest proof for this in the provided research challenges (Section 3.6) and in the challenge discussion about evolving event sourced systems (Section 3.7).

The number of interviews does not allow us to generalize the results. It is not possible to prove that, because 14 engineers use the technique *weak schema*, it is the recommended technique. However, practitioners can integrate the reported experience into their decision-making. They can weigh the context of the interviewed engineers and match that with their own context. Although our research does not result in hard recommendations, we believe that practitioners can benefit from the reported experiences.

3.9 Threats to Validity

Both Golfashani & Nahid [94] and Onwuegbuzie & Leech [178] discuss the challenges of assessing validity in qualitative research. We identify several biases for both internal and external validity. First, we regard the objects of study, i.e., the engineers and their uses of and experience with the pattern. The contributions of our research are based on the 25 interviews that were conducted. The engineers were not hand selected, but volunteered. Therefore, it is possible that we only interviewed a particular subset of practitioners, who were willing and able to discuss the pattern at length. It is for instance remarkable that they all combine CQRS with event sourcing. Table 3.1 shows a diverse variety of experiences, and Table 3.2 shows an equally diverse variety of systems. We have interviewed consultants (E14 and E16), and full-time employees, with a wide range of years of experience. From small systems to multi-million user systems, the interviewed engineers have been exposed to all. These characteristics indicate a broad range of opinions and experiences. Within the group of 25 engineers, 16 engineers have three years or less of experience working on ESSs. This could be due to the relative novelty of the pattern. However, these engineers were involved full-time in the development of the ESS. The exploratory questions (Section 3.11) focus on topics that can be sufficiently answered by engineers with one or two years of experience.

Internal validity, which is strengthened by the way in which the research is conducted, has been defended in several ways. First, an interview and analysis protocol (Section 3.11) had been applied to each interview. The interview protocol was

created from extensive literature study and discussion in the research team, in which two members have no experience with the pattern itself, thereby reducing bias. The first two authors have extensive experience in developing a large ESS. This experience has led to many interactions with practitioners in gatherings, conferences, and online. These interactions have served as an informal triangulation that support the findings presented in this article.

As a constructivist GT approach [33] was followed, we conducted relatively open interviews. The exploratory nature of the interviews enabled interviewees to comment on all aspects of the subject under study, independent of the experience of the engineer with the pattern. Many engineers work on closed-source, commercial systems, which makes it hard to use documentation or source code in the research. Every interview was closed with the question if anything important was left unasked, and if they knew other engineers that we should interview. Often the engineers came with stories and anecdotes that amplified the discussed topics. The engineers that were referred to us were all invited to cooperate.

External validity, i.e. generalizability to other cases, can be defended by the multitudes of systems that the engineers have observed and worked on.

As already discussed in Section 3.2, we do not claim to have reached saturation. Not reaching saturation could leave us open to missing crucial information, or even using incorrect information. Seven of the interviewed engineers have five or more years of experience, and we did not find conflicts between their statements and the other interviews. Together with the experience of the first two authors in developing ESSs, we believe that our findings are supported by the data.

We have not covered all niches in the software world, so we can not generalize to all types of systems. However, we do believe that in the domain of business information systems, we have sufficient coverage to claim generalizability to other systems in this domain. Furthermore, we do believe that other domains can be inspired by our findings in designing event sourced systems. Also, the common occurrence of all event sourcing evolution techniques in Table 3.7, illustrates that we observed a broad cross-section of systems in use. Finally, the use of GT has provided us with a reliable manner of extracting concepts and definitions from the interviews. While this study's findings can be generalized to describe event sourced patterns, the research work is not finished.

3.10 Conclusion

In this article we present a conceptualization of the event sourcing pattern, grounded in interviews with 25 event sourcing engineers. Event sourcing is a pattern that solves the three problems that modern systems face. The flexibility that the combination of event sourcing and CQRS gives decreases the complexity in large systems. The decrease of complexity enables the development of larger systems that remain maintainable. The reliability of the system improves when every state change is stored in a durable store. It allows engineers to undo state changes that were incorrect, or replay those state changes after system failures. An improved reliability is essential for systems that provide increasingly critical processes. Finally, systems that serve increasing

numbers of end-users benefit from the improved scalability that ESSs systems provide.

These benefits give enough reasons to incorporate event sourcing in modern systems. This article presents a thorough description of the pattern, including the context in which it is applied and the consequences that are encountered. The description itself is grounded in the experience of 25 engineers, making it a reliable source for both new practitioners and scientists. We answer the following four research questions in this work.

What types of systems apply event sourcing, and why? The overview of 19 systems, given in Section 3.4 and especially in Tables 3.2 and 3.4, show that event sourcing can be applied in systems of any size: both smaller and larger systems benefit from the pattern. We studied systems with thousands of events up to and including systems with billions of events, and according to their engineers all of these systems have benefited from event sourcing. As E14 states *"I have never seen an event sourced system that was rewritten to a system with traditional current state storage."* The event sourcing pattern is not tied to a specific type of application, but is applied in many different domains, such as marketing, micro-lending, content management and classified advertising. The systems under study show a strong relation to DDD as a software development approach. This is partially explained by the fact that event sourcing and CQRS were invented in the community that grew around DDD. The microservice architectural style has a weaker relation (8 out of 19 systems apply it), while CQRS is used in all these systems. We identify four reasons for event sourcing: audit, flexibility, complexity, and trending. While a common characteristic of event sourcing is the immutability of the events, we show that there are three levels of immutability that can be found in ESSs. The characteristics summarized in 3.2 substantiate that event sourcing can be applied in a diversity of domains and technologies.

How can event sourced systems be defined? Section 3.5 gives definitions of the different concepts in event sourcing and event sourced systems. These definitions are based on our five years of experience in building an ESS, and they are augmented with the interviews. The experiences of the interviewed engineers add nuance and variation options to the different concepts, making them reflect the view of practitioners. Concepts and codes extracted from the interviews scoped our definition: the engineers provided us with topics to define through the interviews.

How can event sourced data structures be evolved? Five event schema evolution techniques are discussed in Section 3.7: *versioned events*, *weak schema*, *upcasters*, *in-place transformation*, and *copy-transform*. For every technique the benefits and liabilities as discussed with the interviewed engineers are summarized in Table 3.7. Almost all engineers have experience with multiple techniques, often combining them in a single system. As all techniques have their benefits and their liabilities we did not find a single technique that would be applicable in all scenarios. We conclude the section with general advice on when to apply specific techniques, and how to combine the techniques.

What are the challenges faced in applying event sourcing? Five challenges that the interviewed engineers experienced are discussed in Section 3.6 and summarized in Table 3.6. We address the steep learning curve in Section 3.5 by giving definitions and operations that can be used in discussing and teaching ESSs. Evolution is discussed in detail in Section 3.7, again using the concepts and operations to explain and characterize the different techniques. The other three challenges, lack of technology, rebuilding projections, and privacy, are presented as a start for a research roadmap. We call for researchers to further explore these challenges.

The main scientific contributions are found in Sections 3.2 and 3.6. In the research approach, we aim to inspire future architecture researchers to use similar qualitative techniques, such as GT, for the explication of architecture knowledge from practitioners. Secondly, a set of research challenges is provided for software engineering researchers to challenge the knowledge around event sourcing in large software systems. Additionally, we are excited to define and document such an important software pattern for the scientific community.

3.11 Interview Protocol

Context related questions

1. Please introduce yourself, the company, the product, and your role in the development.
   (a) How many years is the system in production?
   (b) How many installations are there of the system (single on-premise custom-made, single cloud SaaS, multiple on-premise customers, ...)?
   (c) What is the load on the system in terms of users/ traffic (events?)? Can you give a rough estimate?
2. Why is event sourcing applied in this software system?
   (a) If this decision is already a few years old, is event sourcing still applicable of would the team decide otherwise with the current knowledge?
3. What is the technology stack?
4. Could you give a summary of the size of the system in terms of event sourcing? For instance in terms of different stream types, stream instances and number of events.

Versioning related questions

5. What strategy do you use for event versioning? (Elaborate on the why)
   (a) When using weak serialization: How do you deal with not being able to perform certain operations? Does it bother you, or not?
   (b) When using upcasters: How many upcasters are there? What is the longest chain of upcasters? How do you manage them?
   (c) When using in-place scripts: How do you validate the correctness? What about the audit log, how do you deal with re-writing?

   (d) When using conversion: How long does it take? What about the audit log, how do you deal with re-writing?
6. Do you need/ want the audit features? (What is the level of immutability?)
7. What is your strategy for the query-side? How do you keep this in sync?
8. How often are new versions released, and who performs the upgrade?
9. What kind of upgrade strategy is used? How do you deploy an upgrade?
   (a) Do you have any SLAs based on the domain/product? (such as 24/7, 9 to 5)

Other topics

10. Do you use ProcessManagers/Sagas? Anything special for those?
11. Are you satisfied with the current upgrade and versioning strategy? If not, what would you like to see differently?
12. What do you see as future challenges of ESSs?
13. Can you apply event sourcing without DDD?
14. What would your approach be to building a huge system?

Closing

15. What did we miss? What should we have asked?
16. With whom should we talk?

Acknowledgements

The authors thank all the engineers for sharing their valuable experience and their willingness to contribute to this study. Furthermore, we would like to thank Paris Avgeriou, Fabiano Dalpiaz, Jurriaan Hage, André van der Hoek, John Mylopoulos, Alexander Serebrenik, Jan Martijn van der Werf, Greg Young, Uwe Zdun, and all the anonymous reviewers for their constructive feedback on earlier drafts.

Data Package: Accompanying Anonymized Transcripts

For the research in Chapter 3 we conducted 22 distinct interviews with 25 engineers. The interviewed engineers are event sourcing practitioners in the roles of developers, architects, and product owners.

The interviews were transcribed and interpreted using constructivist grounded theory. The anonymized transcripts of the interviews with 25 engineers on their experience applying Event Sourcing, with the accompanying classifications are made available [188].

Part III

API Management in Software Ecosystems

4

API-m-FAMM: a Focus Area Maturity Model for API Management

Context: *Organizations are increasingly connecting software applications using Application Programming Interfaces (APIs) to share data, services, functionality, and even complete business processes. However, the creation and management of APIs is non-trivial. Aspects such as traffic management, community engagement, documentation, and version management are often rushed afterthoughts.*
Objective: *In this research, we present and evaluate a focus area maturity model for API Management (API-m-FAMM). A focus area maturity model can be used to establish the maturity level of an organization in a specific functional domain described through a number of areas. The API-m-FAMM addresses the areas Lifecycle Management, Security, Performance, Observability, Community, and Commercial.*
Method: *The model is constructed using established methods for the design of a focus area maturity model. It is grounded in literature and practice, and was developed and evaluated through a systematic literature review, eleven expert interviews, and five case studies at software producing organizations.*
Result: *The model is described in detail, and its application is illustrated by six case studies.*
Conclusions: *The evaluations are reported on, and show that the API-m-FAMM is an efficient tool for aiding organizations in gaining a better understanding of their current implementation of API management practices, and provides them with guidance towards higher levels of maturity. The detailed description of the construction of the API-m-FAMM gives researchers an example to further support the available methodologies, specifically how to combine design science research with these methodologies. Additionally, this studys unique case study design shows that maturity models can be successfully deployed in practice with minimal involvement of researchers. The focus area maturity model for API Management is maintained on* `www.maturitymodels.org`*, allowing practitioners to benefit from its useful insights.*

This work was originally published in *Information and Software Technology, volume 147 (2022)*, titled 'API-m-FAMM: a Focus Area Maturity Model for API Management'. It was co-authored by Max Mathijssen and Slinger Jansen.

4.1 Introduction

In recent years, there has been an increasing demand among organizations to have access to enterprise data through a multitude of digital devices and channels. This demand is also increased by the transformation from software product towards a platform, called 'platformisation' [193]. Platforms are a vehicle for software ecosystems and are defined as a set of organizations collaboratively serving a market for software and services [126]. In order to meet these demands, enterprises need to open up and provide access to their assets in an agile, flexible, secure and scalable manner [53]. These assets include matters such as raw and cleansed data or functionality that perform complex calculations or data processing based on inputs [266]. Access to these assets may be provided by utilizing Application Programming Interfaces (APIs). De [53] defines an API as a software-to-software interface that defines a contract for applications to communicate with one another over a network, without the need for any user interaction.

As shown by an analysis conducted by ProgrammableWeb [221], known as the largest directory of APIs, the usage and offering of APIs has evolved from a curiosity to a trend since 2005. This observation is further supported by a survey conducted by Coleman Parkes Research [41], showing that 88% of global enterprises have some form of an API program. Furthermore, the survey found that respondents experience a wide variety of benefits from their API programs, including an average increase in speed-to-market of around 18%. These statistics signal the emergence of the API Economy, in which organizations are offering access and the ability to recombine their digital services and products for novel value creation [12]. As a result, by making their APIs accessible to external or partner consumers, these organizations are able to reach new markets, enable their business strategy, and drive the creation of new innovative solutions [26]. After an API has been created it needs to be managed so that developers may easily integrate it into their applications. API management is done by performing activities such as providing helpful documentation, controlling access to the API, as well as monitoring and analysing its usage.

Medjaoui et al. [162] list three reasons that make it hard for organizations to improve their API management activities. First, organizations that are performing well in terms of their API management programs often do not have the time, resources or personnel to share their experience and expertise with third parties. Secondly, organizations that are careful with regard to the amount of knowledge they share on their API management expertise might consider their know-how to be a competitive advantage, and will as such not feel urged to make their findings public. Finally, even in the event where organizations share their experience at public conferences, articles or blog posts, the information shared is usually company-specific and difficult to translate to a wider range of organizations' API programs.

Focus area maturity models (FAMMs) [235, 236] are an established method to communicate extensive domain knowledge. Not only do they contain this knowledge, they also offer a clear path for organizations to improve their maturity in a certain domain. In this article we present the API-m-FAMM, a focus area maturity model for API management. We show that this model improves on existing API management assessment

frameworks and tools in terms of transparency and availability, and that it can be used by organizations that expose their API(s) to third-party developers to assess and evaluate their degree of maturity with regards to API management. We also extensively describe the methods that are applied in constructing the FAMM, and improve the available design methods by providing a concrete and detailed example. Additionally, explicit attention is paid to support organizations in performing their own assessment by using a *do-it-yourself* kit that we created and supplied to organizations.

Section 4.2 provides an overview of existing assessment models for API management and discusses their strong and weak points. A description of our research approach is given in Section 4.3, including the detailed steps that were taken to develop the FAMM. Section 4.4 describes the API-m-FAMM and Section 4.5 discusses how it is applied in four different companies. The results of these case studies are discussed in Section 4.6, while Section 4.7 discusses focus area maturity models in general. The threats to validity are discussed in Section 4.8. Section 4.9 summarizes our findings and contributions, among which are the previously undefined framework for API management, a detailed description of the construction of a focus area maturity model using both an existing methodology as well as tools from design science research, and finally an example of how we can make focus area maturity models more accessible by investing in their usability.

4.2 Related Work

In an effort to guide organizations in successfully managing their API programs, a number of commercial frameworks and tools exist with which organizations may evaluate and assess their API management approach and capabilities. In this section these existing frameworks, tools, models, reports and case studies are summarized and discussed. The existing frameworks and tools are assessed on several attributes. First of all we discuss availability, some frameworks are only available commercially and require extra costs. Secondly we discuss the grounding of the framework, some frameworks are grounded in scientific literature, others only in experience from an industry setting, and one framework is grounded in both. Finally we discuss the transparency of the framework: can we find details on how this framework is constructed. As becomes apparent, these frameworks are either not publicly available, transparent or grounded in academic literature.

Accenture API Management Suite - Based on their experience with implementing API programs, Accenture Technology Labs has developed a Maturity Model for APIs [251]. The Accenture model consists of 5 maturity levels, and is aimed towards helping organizations identify the maturity of their API management capabilities. These maturity levels are mapped onto five distinct dimensions, which detail the processes an organization should implement in its journey from API enablement to industrialization. However, this maturity model fails to address certain core API management-related processes and capabilities such as versioning, threat protection and lifecycle management. The model is also quite outdated, and has since been deprecated, as it is no longer available on Accenture's official website. Additionally, in part due to its industrial foundation and commercial nature, it is unclear as to how

the contents of this model have been populated.

Endjin Maturity Matrix - This maturity matrix [68] is a tool that was developed to aid business decision-makers in assessing their organization's ability to evolve towards an API-driven business model. The assessment is performed by having the organization fill out their perceived degree of maturity related to a set of categories, based on which practical suggestions for improvement are then provided.

While this maturity matrix comprises a selection of categories that are relevant in the scope of API management such as governance, documentation and support, it mainly focuses on strategies and commercial aspects. As a result, many API management-related aspects such as traffic management and community engagement are missing from the matrix.

WSO2 Platform Evaluation - In 2015, WSO2 published a whitepaper that describes digital business goals, outlines API-oriented IT initiatives, and presents API management platform requirement categories [271]. Alongside this whitepaper, an evaluation matrix spreadsheet is provided that details a set of evaluation criteria, which may be used to evaluate API management platform vendors [102]. In illustrating the discipline of API management, two types of APIs are discerned: naked and managed APIs. A naked API is considered to be not monitored, managed, secured, documented or accessible through a self-service subscription portal, a managed API on the other hand is thought to be actively advertised and subscribable, available alongside a published service-level agreement (SLA), secured, authenticated, authorized, protected, as well as being monitored and monetized by using analytics. The whitepaper argues that to move from naked to managed APIs, the *API façade pattern* should be implemented, which enables teams to layer network-addressable endpoints, monitor usage, enforce usage limits, manage traffic, and authorize consumers. According to WSO2, an API management infrastructure should guide teams towards best practices with regard to six main focus areas. However, it is unclear as to whether organizations are supposed to assign themselves scores, or whether they are assisted by WSO2 in this process. Furthermore, while a 'weighted score' column is included in the matrix, it is unclear what these weights are based on, or what formula is used to calculate the weighted scores. Due to the fact that this whitepaper and matrix were written by WSO2, which is a commercial API management platform provider, organizations are steered towards selecting the WSO2 platform.

Gámez Gateway Comparison - As part of their work, which seeks to analyze the API Gateway paradigm and propose a SLA-Driven solution in an API Gateway design, Gámez Díaz, Fernández Montes & Ruiz Cortés [89] have compared API management features offered by various API gateway providers. This collection of providers consists of 13 Gateways including: *3Scale, Akana API Gateway, API Umbrella, Apiaxle, Apigee Edge, Axway API Gateway, Azure API Management, CA API Gateway, Mashape, Mashery API Gateway, Monarch API Manager, Repose* and *WSO2 API Management*. The aforementioned gateways were compared as based on a set of features such as *Security*, *Pricing plans support*, and *Lifecycle Control*. Confusingly, in their work, Gámez Díaz, Fernández Montes & Ruiz Cortés [89] use the terms 'API gateway' and 'platform' interchangeably. As a result, it is unclear whether the intention of the authors was to analyze features offered by the API gateway component, which is one of the main ar-

chitectural components offered by the listed API management platform providers, or features provided by the platforms as a whole. Moreover, the set of features the aforementioned API gateways are compared on is very limited when compared to work on API management by authors such as De [53].

Broadcom Playbook - In order to promote their API management platform, called Layer7, Broadcom has employed CA Technologies to compose an 'API Management Playbook' [27]. This playbook is targeted towards helping its readers comprehend the various reasons for API's importance in business, the API lifecycle and its relation to API management, the essential capabilities of an API management solution, and the features offered by the Layer7 platform. An evaluation method is presented which considers API management capabilities based on a collection of 13 use cases, or 'plays'. These use cases are broadly classified as having the goal of: API integration and creation, security, mobile and internet of things (IoT) development acceleration, and unlocking the value of data.

It is clear that even though this work presents an useful overview of API management capabilities, this overview directly matches the features offered by the Layer7 solution. As such, it may be concluded that the main purpose of this work is to convince and attract potential customers to Broadcom's platform. Furthermore, due to the commercial nature of this document, it cannot be considered transparent in the sense that the source of the presented information is not known.

Accenture Advisory Report - In addition to their maturity model, Accenture has also published a consultancy report in 2019, advising banks on how to implement API management [150]. Even though this report is specifically focused on the banking sector, and as a whole may thus be difficult to generalize and apply to other sectors, it contains several frameworks, figures and models that are related to this study.

The presented capability overview may be utilized by organizations seeking to implement API management processes, or may aid them in identifying API management platform capabilities that cater towards their needs. Furthermore, considering the aforementioned pillars are described in great detail and provide clear-cut guidelines for organizations to follow, this advisory report may be beneficial to organizations wishing to implement API management. However, when compared to the earlier described maturity models and frameworks, it may be difficult for organizations to self-assess their degree of maturity concerning API management.

Gartner Guidance Framework - Published by Gartner in 2019 [93], this report is similar to those published by WSO2 and CA Technologies. It is aimed towards assisting technical professionals in selecting an appropriate API management platform. The general outline and structure of the framework are visible by reviewing the table of contents. Judging from this outline, it may be concluded that the framework is grounded in literature, using De [53]'s work on API management as a foundation.

Devoteam Case Study - On their website, a Dutch company called Devoteam summarizes a case study in which the implementation of an API management platform at a large organization, *Liberty Global*, is described [57]. As part of this case study, the case organization is described to have initially implemented an API gateway, which is argued to be an incomplete solution when compared to an API management platform. Furthermore, the organization required API management capabilities such as devel-

oper and partner onboarding, lifecycle management, documentation and testing, and analytics, which Devoteam [57] does not consider to be capabilities that are typically provided by an API gateway.

In order to recommend an appropriate API management platform to the case company as based on their needs, an 'integration cookbook' was created, documenting the principles and guidelines for the usage of the API management platform and the different integration patterns. Unfortunately, similarly to the earlier described framework by Gartner, this cookbook is not publicly accessible. However, Devoteam [57] mentions that it comprises policies that range from security policies, such as OAuth 2.0, OpenID Connect, basic authentication and IP whitelisting, to operational policies and monitoring and auditing policies. Judging from this information, it seems as though this cookbook primarily focuses on strategy, governance, and API management vendor evaluation and comparison. Due to its commercial nature however, it is unknown as to how this cookbook was created, whether it is able to be used to assess an organization's degree of maturity with regards to API management, or how complete it is.

A summarizing comparison matrix is presented in Table 4.1, which compares the artifacts. From the discussions of the different frameworks it becomes clear that the frameworks and tools either have a different goal (support consulting or platform selection), are not complete (missing capabilities), are not publicly available, or are not transparent in their construction. Our model combines the goal of evaluation and knowledge sharing, is publicly available, is transparent in the construction of the model, and describes the complete set of capabilities and practices for API management.

4.3 Research Approach

As discussed in the previous section, organizations that employ API management activities have no tools or frameworks at their disposal with which they may evaluate and improve upon their business processes regarding the topic of API management, that are publicly available, transparent, and grounded in both literature and industry. Despite growing interest in the topic of API Management in industry, more research is needed in order to fill knowledge gaps and identify best practices regarding the subject. Based on this problem statement the following research question is formulated, ensuring this research succeeds in achieving its goals. *How can organizations that expose their APIs to third parties evaluate their API management practices?*

Maturity models have been developed for organizations to use as an evaluative and comparative basis for improvement, in order to derive an informed approach for increasing the capability of a specific area within an organization [25]. Moreover, maturity models have been designed to assess the maturity of a specific domain based on a set of criteria, and are a proven tool in the creation of collections of knowledge of practices and processes about a particular domain [14]. Maturity models consist of a sequence of maturity levels for a class of objects, which typically include organizations or processes. The aforementioned sequence represents an anticipated, desired, or typical evolution path of these objects as discrete stages [196]. In order for an orga-

Framework	Type	Missing content	Goal	Availability	Grounding	Published	Transparent
Accenture API Management Suite	Maturity Model	Versioning, threat protection, authorization	Evaluation	Public	Industry	2014	No
Endjin Maturity Matrix	Maturity Matrix	Lifecycle management, dev. onboarding, traffic management, interface translation, monitoring	Evaluation	Public	Industry	2017	No
WSO2 Platform Evaluation	Whitepaper & Evaluation matrix	Dev. support	Platform selection	Public	Industry	2015	No
Gámez Gateway Comparison	Comparison Matrix	Dev. onboarding & support, authorization, traffic management, monitoring	Platform selection	Public	Literature	2015	Yes
Broadcom Playbook	Whitepaper	Version management, interface translation, logging	Consulting support	Public	Industry	2019	No
Accenture Advisory Report	Whitepaper	Dev. support, interface translation, logging	Knowledge sharing	Public	Industry	2019	No
Gartner Guidance Framework	Evaluation framework	Unknown	Platform selection	Commercial	Industry & Literature	2019	No
Devoteam Case Study	Case Study & Cookbook	Unknown	Platform selection	Commercial	Industry	2016	No

Table 4.1: Comparison of existing frameworks and tools for API management assessment. It must be noted that the descriptions of the practices in each of these models typically had a low level of detail.

nization to progress along this path, criteria and characteristics relating to capabilities or process performance are included which need to be fulfilled to reach a particular maturity level. The lowest stage of this path represents an initial state that may be characterized by an organization having little capabilities with regard to the domain under investigation. Organizations whose capabilities are of the highest maturity are located at the highest stage. Maturity refers to being well equipped to fulfill a purpose, i.e. having a higher level of sophistication, capability, or availability of specific characteristics [167]. To appraise an organization's maturity, maturity models are commonly applied to assess the as-is-situation regarding the given criteria, so that improvement measures may be derived and prioritized [119].

As De Bruin et al. [54] and Steenbergen et al. [236] argue, the most well-known maturity model within the field of Information Systems is the Capability Maturity Model (CMM) from the Software Engineering Institute [190]. Studies have shown that since its inception, the CMM has inspired the development of hundreds of subsequent maturity models. These maturity models are aimed at a wide variety of domains and topics, including, for example, project management [45], corporate data quality management [114], service integration [8], and offshore sourcing [30]. Another example, which has been discussed in Section 4.2, is Accenture's API management maturity model [251].

An improvement over the maturity model is the Focus Area Maturity Model (FAMM) [235, 236]. A FAMM allows a flexible number of maturity levels and assesses the maturity per focus area which results in a more detailed evaluation. We follow the meta-model of FAMMs as presented by Jansen [125] (shown in Figure 4.1), his work describes the development of a FAMM for the functional domain of software ecosystem governance. The functional domain is described by the set of focus areas that constitute it. Each focus area is composed out of a set of capabilities, which in the case of the API-m-FAMM are defined as the ability to achieve a goal related to API Management through the execution of two or more interrelated practices. Together, these practices and capabilities form the focus areas which describe the functional domain of which API management is composed. A practice is defined as an action that has the express goal to improve, encourage, and manage the usage of APIs. Each individual practice is assigned to a maturity level within its respective capability. In order to establish an organization's degree of maturity with regard to the functional domain, the organization is asked to answer assessment questions linked to the capabilities the maturity matrix consists of. Based on the results of this maturity assessment, the organization is then guided towards incremental development of the domain, through a set of improvement actions with regard to the (missing) capabilities.

We apply the design methodology of Steenbergen et al. [235] and De Bruin et al. [54] in constructing our FAMM. The development of the FAMM is done in five phases: *Scope*, *Design*, *Populate*, *Test*, and *Deploy*. These phases are executed through a Systematic Literature Review (SLR), expert interviews, case studies, and numerous discussions among the authors. Every phase concluded with the authors discussing the state of the model until consensus was reached on its contents and structure. This was done using online *Card Sorting* [175], with *Google Drawings* as a tool. In the *Scope*, *Design*, and *Populate* phases, the input for these discussions primarily consisted of the

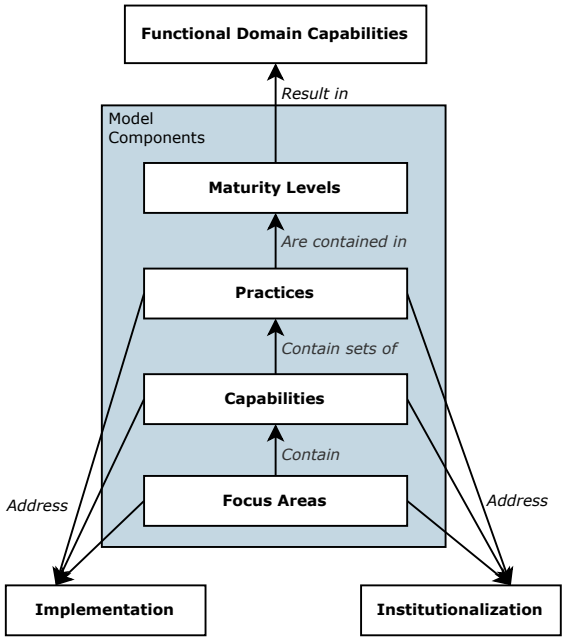

Figure 4.1: The meta-model of focus area maturity models as used in this research.

practices found in the different sources. The authors used card sorting to categorize these practices into capabilities, and to categorize the capabilities in areas. The various sources were used as inspiration for the discussions. During the *Test* phase the experts were also asked to comment on these categorizations. Comments that were deemed to be correct were then integrated into the model. Practices were assigned to a maturity level based on the ordering of practices within a capability and identified dependencies. In every phase we identified dependencies between the different practices. When a practice depends on a practice with maturity level l, the practice itself should have at least maturity level $l + 1$.

Figure 4.2 shows which methods were used in each phase, linked to the different intermediate versions of the API-m-FAMM. We use these versions to reference a certain point in the construction of the model. A full description of the changes made during the different phases is available through the source data [159]. The source data also details the dependencies between the different practices. This document was published at different points: *v1 of the source data* corresponds with *v0.2*, *v2 of the source data* with *v0.3*, *v3 of the source data* with *v0.4*, and *v7 of the source data* with *v1.0*.

The initial model (*v0.1*) used the work of De [53] as a starting point. Next a SLR [158], based on the methodology developed by Okoli [176] and guidelines composed by Kitchenham & Charters [138], was used to further design and populate the model (resulting in version *v0.2*). In this SLR a comprehensive overview of literature related to API management was collected. This was accomplished by entering a series

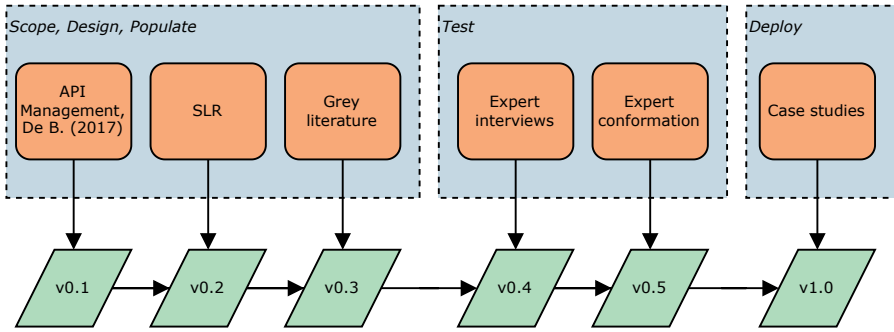

Figure 4.2: The steps executed in construction and the different intermediate versions of the API-m-FAMM.

of relevant keywords in a list of scientific libraries, resulting in the extraction of an initial collection of 5152 books, research papers, theses and white papers. After having applied a set of inclusion and exclusion criteria, as well as removing duplicates, this collection was narrowed down to 43 published works. Next, the features API Management consists of were identified and extracted in the form of practices and capabilities from the complete body of literature. As a result of scanning and coding the body of included literature, 39 capabilities were identified and extracted. Among the 32 papers that were found to contain at least one practice or capability, 114 practices were identified and extracted.

To ground the model in both academic literature and industry experience two extra sources were used to construct the model. The collection of practices and capabilities resulting from the SLR are verified by using information gathered from grey literature, which includes white papers, online blog posts, websites, commercial API management platform documentation and third-party tooling (resulting in version *v0.3*).

Furthermore, experts on this topic are consulted to verify that the contents of the API-m-FAMM are complete and correct. To ensure that the selected experts are experienced and knowledgeable regarding the subject a purposive sampling technique is used, which refers to the deliberate choice of a participant due to the qualities the participant possesses [73]. The process of expert selection and interviews is visualized in Figure 4.3. Potential participants were identified and contacted through the usage of the partner network of AFAS Software (the company where the first and second authors are employed at). Additional potential participants were identified and contacted through the usage of the professional network of the authors. Furthermore, potential participants on both the API provider and consumer side are contacted. This ensures that knowledge stemming from both perspectives of API management is incorporated into the API-m-FAMM.

In order for the expert interviews to produce useful results, participants should be experienced and knowledgeable with regard to the topic of API management. As such, participants should adhere to the following requirements:

1. potential participants must indicate to be knowledgeable on a minimum of two out of the six focus areas of the API-m-FAMM;

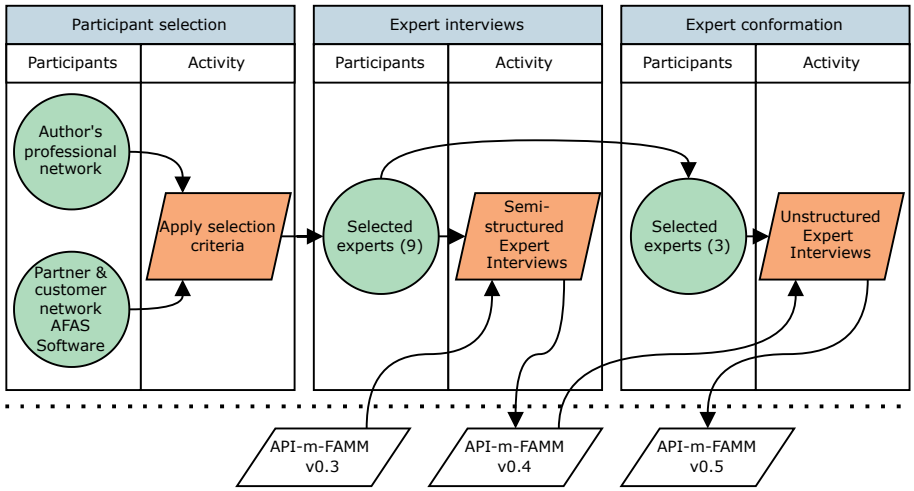

Figure 4.3: The process of expert selection explained. First we selected possible experts based on the author's professional network and the case company's partner and customer network. While 50 experts were invited, eleven experts responded to the survey. We selected nine experts by applying our selection criteria and interviewed them using *v0.3* of the model. The results of these interviews were integrated in *v0.4*. From the nine experts we selected three experts to conduct extra conformation interviews. The results of these interviews were integrated in *v0.5*.

2. potential participants must have a minimum of 3 years of experience with either consuming, developing, integrating, providing, versioning, monitoring or managing APIs;
3. potential participants must be working at an organization as an architect, developer, engineer, or product owner as part of a team working with APIs, or as a CTO, IT consultant or any comparable role.

In order to establish whether the potential participant satisfies the first requirement, a short preliminary survey is sent out, requesting the potential participant to indicate the degree of knowledge they possess with regard to the topic. 50 potential participants were contacted through e-mail. The survey was sent together with a shortlist describing all the main elements that are contained in the API-m-FAMM, accompanied by a description (based on version v0.3). After having read these descriptions, potential participants are asked to denote their knowledge of the individual subjects through the use of 5-point Likert scale questions. In order to verify the second and third requirement, potential participants are asked to fill out their experience and current role within their organization. Then, if the potential participant passed all imposed requirements for participation, participants were asked to read an information sheet and sign an informed consent form.

The sampling process resulted in the selection of nine experts, who are presented in Table 4.2 (two respondents were excluded based on the knowledge assessment). This ratio may seem high at first glance, but considering only one or two focus areas were discussed during an expert interview, it was deemed to be unnecessary for participants

Interviewee	Experience	Hours	Community	Security	Lifecycle	Monitoring	Performance	Commercial
$API\ Evangelist$	10+	1	✓					✓
CEO_A	10+	1.5	✓					
CEO_B	10+	5.5		✓	✓	✓		
$Engineer$	10+	1				✓	✓	
$IT\ Consultant$	10+	3.5	✓	✓	✓			✓
$Product\ Manager$	6	3		✓				
$Lead\ Engineer_A$	10+	1				✓	✓	
$Lead\ Engineer_B$	10+	1					✓	✓
$Lead\ Engineer_C$	5	1.5			✓			
Total	N/A	19	3	3	3	3	3	3

Table 4.2: Interviewees and the current role they fulfill within their organization, as well as the years of experience they have with either consuming, developing, integrating, providing,versioning, monitoring or managing APIs. Additionally, it is shown which focus areas were discussed with interviewees, as well as the duration of the interviews.

to be knowledgeable on more than two focus areas as only those particular focus areas were included for discussion during interviews. Based on an interviewee's knowledge regarding the six focus areas the API-m-FAMM consists of, one or more focus areas are selected for evaluation. Focus areas were evaluated through eleven interviews, which resulted in each focus area being evaluated three times. These interviews were subsequently transcribed and processed to incorporate all comments into the FAMM.

During the interviews, which are semi-structured in nature, the API-m-FAMM in its entirety is first presented to the expert. Next, the focus area that is selected for evaluation and each capability it comprises of is described. Then, all practices these capabilities consist of are elaborated upon. For each capability, experts are asked whether they are familiar with it and whether they believe it is assigned to the correct focus area. Similarly, experts are asked whether they are familiar with each practice, and whether they believe it is assigned to the correct capability. Additionally, they are asked whether they can identify any dependencies with regard to the implementation of other practices. After having answered these questions for a capability and the practices it comprises, experts are asked to rank practices in terms of their perceived maturity and complexity for each capability. This ranking exercise is performed by using the same card sorting technique that was used to initially structure the API-m-FAMM (via *Google Drawings*). Figure 4.4 shows an example ranking result of a single capability. These results are analyzed and combined with any implementation dependencies to create a final ranking of the practices (resulting in *v0.4* of the FAMM).

The resulting version of the FAMM was evaluated through a second cycle of unstructured interviews with three experts originating from the same sample of experts that were interviewed during the first evaluation cycle: $Product\ Manager$, $IT\ Consultant$, and $Lead\ Engineer_A$. These experts are knowledgeable on a large number of areas

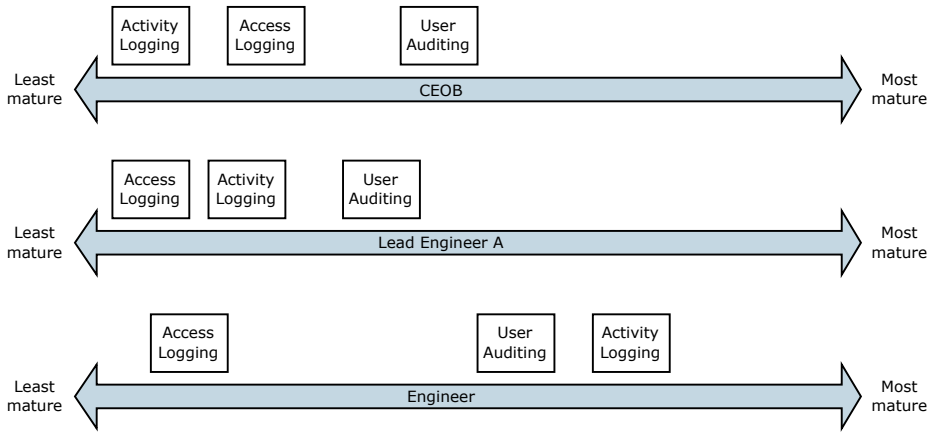

Figure 4.4: Ranking of practices in the capability *Logging* (intermediate practices are shown as an intermediate version was used during the interviews) as done by the experts.

and based on that criteria were asked to participate in the second evaluation cycle. During these interviews the changes made to the FAMM were discussed and confirmed (resulting in *v0.5* of the FAMM).

Finally the model was deployed in four companies, by evaluating five different software products. As a result of the case studies the model was changed by removing one practice and improving the descriptions of three other practices. This resulted in *v1.0* of the API-m-FAMM. The contents of the API-m-FAMM are discussed in Section 4.4, and the case studies are described in Section 4.5.

4.4 The API Management Focus Area Maturity Model

The previous section motivates the usage of the focus area maturity model as an artifact to capture the functional domain of API management, and describes the methods used in constructing it. This section introduces the **API M**anagement **F**ocus **A**rea **M**aturity Model and its contents: the **API-m-FAMM**.

The scope of the API-m-FAMM is the domain of API management: the API-m-FAMM aims to support organizations that expose their API(s) to third-party developers in their API management activities. Based on the SLR API management is defined as *an activity that enables organizations to design, publish and deploy their APIs for (external) developers to consume*. However, three adjustments are made to the scope of the API-m-FAMM with respect to this definition.

First, considering that a prerequisite for performing API management as an activity, is for an organization to already have designed and created an API, the actual design and creation process of APIs itself is excluded from the API-m-FAMM. These processes overlap with software engineering in general and would require the inclusion of capabilities such as agile software development and test-driven development. In contrast however, the publication, maintenance, and deprecation (which are contained in the

API lifecycle) of APIs are included in the scope of the API-m-FAMM, considering that these are concerned with capabilities such as lifecycle management, developer enablement, security, and analytics.

Second, the management of internal APIs is not explicitly targeted with the API-m-FAMM, in contrast to the management of partner and public APIs. Capabilities such as developer enablement, security and analytics are of lesser importance with regard to managing internal APIs, considering that these APIs are exclusively used for internal app integration and development. However, the case studies (discussed in Section 4.5) show that the API-m-FAMM still proves to be useful for such organizations wishing to incrementally improve upon capabilities such as lifecycle management, and performance of their API(s).

Finally, the API-m-FAMM seeks to provide practitioners with incremental capability improvements that are tool-, technology-, and platform-independent. For example, there are many tools that organizations may utilize to implement monitoring, communication, and support capabilities. In the same vein, capabilities regarding traffic management or security, such as authentication, authorization, and threat protection, may be relatively straightforward to implement through the use of commercially available gateway or management platform solutions. However, some organizations may opt to develop these solutions in-house, or may not wish to use such solutions for a variety of reasons. To ensure that the API-m-FAMM is generalized for both these types of organizations, comparisons between specific tools or management platform solutions are excluded from the scope of the model. An example of practices that were removed because of this are *Visual Data Mapping*, which is exclusively provided by the *Axway* API management platform [1] and *Error Handling*, which is implementable through the use of the *Apigee* platform [2]. Another example is the practice *Standardized Authorization Protocol*, which was initially included as *OAuth 2.0 Authorization*, but was renamed with a referral to the OAuth 2.0 protocol being included in its description instead.

As discussed by De Bruin et al. [54], the design phase explains how the needs of the intended audience are met through answering the *why*, *how*, *who*, and *what* questions. The **'why'** for the API-m-FAMM is that its main goal is to assist organizations that (plan to) expose their API(s) to third-party developers to assess and evaluate their degree of maturity with regard to their API management capabilities. The **'how'** is that organizations can utilize the API-m-FAMM to assess their as-is situation with regards to their API management capabilities, and then subsequently incrementally improve upon these capabilities by implementing practices that are of a higher maturity. The **'who'** involved in applying the API-m-FAMM may vary across organizations, depending on characteristics such as their size in terms of employees involved in the API program, number of exposed APIs, and the degree of incoming traffic. For example, for a small organization that exposes one mildly popular API, it is likely that a small number of employees are familiar with the API program and the activities involved in managing it. These employees may then utilize the API-m-FAMM to assess the as-is situation, and then use the model as a road map to incrementally implement capabilities

[1] `https://www.axway.com/en/products/api-management`
[2] `https://cloud.google.com/apigee/api-management?hl=nl`

and practices to reach a higher level of maturity. In contrast, a large organization that exposes multiple, popular APIs that generate large loads of traffic, is likely to employ multiple development teams, product owners and designers who are involved with the API program. In this case, it is unlikely that a solitary employee or a small group of employees will be able to assess the current as-is situation of the API program. Instead, information for this assessment will have to be extracted through meetings with employees from varying teams and backgrounds who are involved in the various aspects of API management, such as community engagement, security measures, monitoring capabilities, and lifecycle management. Alternatively, consultants with a thorough understanding of API management and the API-m-FAMM may be able to apply the model by conducting interviews with all relevant stakeholders involved in the organization's API program. The **'what'** that can be achieved through the application of the API-m-FAMM is an insight into the current maturity of an organization with regards to its API management capabilities, as well as a path to incremental implementation and improvement of more mature, specific practices.

The API-m-FAMM, based on the SLR, the expert interviews, and case studies, is presented in Figure 4.5. Detailed descriptions, preconditions, and reference literature can be found in the source data [159]. The FAMM consists of six focus areas, 20 capabilities, and 80 practices. The highest maturity level is 10.

During the expert interviews, part of the *Test phase* (see Figure 4.2), an interim version of the API-m-FAMM, was measured on four criteria using Likert scale questions:

- **Operational Feasibility**: How likely do you think it would be that an organization would actually use the API-m-FAMM in practice to evaluate and improve upon their API management related processes?
- **Ease of Use**: How easy do you think it would be to understand the API-m-FAMM's content and use it to self-assess and evaluate your organization's maturity in API management?
- **Usefulness**: How useful do you think the API-m-FAMM would be in providing you and your organization with valuable and interesting insights in your organization's API management related processes?
- **Effectiveness**: How effective do you think the API-m-FAMM would be in helping you and your organization improve on their API management related processes?

The results of this interim evaluation are listed in Table 4.3. These results should be prefaced by a few initial remarks. Firstly, it should be noted that during the interviews, an intermediate and unfinished version of the API-m-FAMM was presented to experts. One of the implications of this is that maturity levels were absent from this version of the model. Because of this, experts often found it difficult to envision what the final version would look like, as well as to properly judge its ultimate capabilities and potential for helping an organization in improving their API management maturity. Additionally, only one or two of the six focus areas the model consists of were selected for discussion, which, in some instances, impaired experts ability to grasp the API-m-FAMMs scope as a whole on a conceptual level. Furthermore, the descriptions given for the focus areas, capabilities, and practices during the interview were summarized and shortened. Lastly, the experts had not familiarized themselves with the contents of the API-m-FAMM prior to the interview taking place.

Components \ Maturity Levels		0	1	2	3	4	5	6	7	8	9	10
1.	**Lifecycle Management**											
1.1.	Version Management			Implement Evolutionary API Strategy			Implement Multiple API Versions Strategy	Implement API Deprecation Protocol	Check Backwards Compatibility			
1.2.	Decoupling API & Application		Decouple API & Software Versioning			Decouple Internal & External Data Model	Decouple Internal & External Data Format	Decouple Internal & External Transport Protocol				
1.3.	Update Notification			Distribute Changelogs	Distribute Versioning Notification through Channel(s)		Extend API with Versioning Information				Announce API Versioning Roadmap	
2.	**Security**											
2.1.	Authentication		Implement Basic Authentication			Implement Authentication Protocol			Implement Single Sign-on			
2.2.	Authorization			Implement Access Control		Implement Token Management		Implement Standardized Authorization Protocol	Implement Authorization Scopes			
2.3.	Threat Detection & Protection		Implement Allow & Deny IP Address Lists	Implement Injection Threat Protection Policies			Implement DoS Protection		Implement Security Breach Protocol		Conduct Security Review	Implement Zero Trust Network Access
2.4.	Encryption		Implement Transport Layer Encryption		Implement Certificate Management							
3.	**Performance**											
3.1.	Resource management			Implement Load Balancing			Implement Scaling	Implement Failover Policies				Implement Predictive Scaling
3.2.	Traffic Management		Set Timeout Policies	Implement Request Caching	Perform Request Rate Limiting	Perform Request Rate Throttling	Manage Quota	Apply Data Volume Limits			Prioritize Traffic	

Figure 4.5: The API-m-FAMM: a focus area maturity model for API management. Consisting of six focus areas, 20 capabilities, and 80 practices.

4.	**Observability**											
4.1.	Monitoring		Monitor API Health		Monitor API Performance		Monitor Resource Usage					
4.2.	Logging		Log Errors	Log Access Attempts	Log Activity		Audit User Activity					
4.3	Analytics			Report Errors	Broadcast API Status			Generate Custom Analysis Reports	Set Alerts		Enable Predictive Analytics	
5.	**Community**											
5.1.	Developer Onboarding		Facilitate Developer Registration			Provide SDK Support	Implement Interactive API Console			Provide Sandbox Environment		
5.2.	Support		Establish Communication Channel			Manage Support Issues		Dedicate Developer Support Team				
5.3.	Documentation		Use Standard for Reference Documentation		Provide Start-up Documentation Including Samples		Create Video Tutorials					
5.4.	Community Engagement		Maintain Social Media Presence		Provide Community Forum	Provide Developer Portal			Organize Events		Dedicate API Evangelist	
5.5.	Portfolio Management		Enable API Discovery			Provide API Catalog	Bundle APIs					
6.	**Commercial**											
6.1.	Service-Level Agreements		Publish Informal SLA		Provide SLA			Monitor SLA Proactively	Customize Personalized SLA			
6.2.	Monetization Strategy							Adopt Subscription-Based Monetization Model		Adopt Tier-Based Monetization Model	Adopt Freemium Monetization Model	Adopt Metering-Based Monetization Model
6.3..	Account Management			Implement Subscription Management System					Report on API Program Business Value	Provide Subscription Report to Customer	Proactively Suggest Optimizations to Customer	

Figure 4.6: The API-m-FAMM, continued.

Expert	Operational Feasibility	Ease of Use	Usefulness	Effectiveness
$API\ Evangelist$	3	2	4	3
CEO_A	4	3	4	4
CEO_B	3	2	3	4
$Engineer$	4	4	4	5
$IT\ Consultant$	2	2	3	4
$Product\ Manager$	4	4	3	3
$Lead\ Engineer_A$	4	3	5	4
$Lead\ Engineer_B$	3	2	4	3
$Lead\ Engineer_C$	4	3	5	4
Average	3.4	2.8	3.9	3.8
Std. Dev.	0.68	0.62	0.74	0.63

Table 4.3: The rankings given by the experts in response to the questions corresponding to the four evaluation criteria, as well as their averages and standard deviation.

Still the experts generally responded positively to the model, and expressed interest in using it in practice to evaluate their organization's API management related processes and assess their API management maturity. For example, $Product\ Manager$ concluded the interview with the following remark: *"I think this - the API-m-FAMM - is a very thorough analysis. You have made a very nice overview that can help organizations with deciding what and when they have to do if they start with an API. If they want to bring something to the market quickly, this helps them realize they must first have implemented the processes on the lower levels, and have to start small. Actually, for many organizations that already have an API or want to start building one, this is the roadmap they should follow for a good API strategy."* The evaluation criteria as stated earlier should be interpreted as how the expert thinks the final model will score. These scores are no proof of the actual effectiveness or usefulness of the model, but an indication of how the model will be received by industry in the future. In this phase the criteria are used as *marketing research*: how likely will the API-m-FAMM succeed in industry.

4.5 Case Studies

The case studies, as described in Section 4.3, are evaluative in nature and are aimed at determining to what degree the API-m-FAMM succeeds in aiding an organization in evaluating and improving upon their API management related business processes in practice. This corresponds with the *Deploy* phase of the design of the maturity model. The API-m-FAMM is evaluated with an embedded case study at the company the first

and second author are employed at, and through case studies at four different companies. Data resulting from the application of the API-m-FAMM is collected through an Excel spreadsheet. Finally, participants evaluated the API-m-FAMM on the same criteria that were employed as part of the first evaluation cycle. These processes are part of the case study protocol that is used in conducting the case studies.

The main objective of the case study is to determine to what degree the API-m-FAMM succeeds in aiding an organization in evaluating and improving upon their API management related business processes in practice. The selected organizations were provided with a visual copy of the API-m-FAMM, as well as a copy of the source document describing the focus areas, capabilities, and practices the model consists of in detail. Considering that the API-m-FAMM was previously presented to selected participants as part of the first evaluation cycle, they were already informed with regards to the focus of this research as well as the purpose and structure of the API-m-FAMM. After having familiarized themselves with the contents of the API-m-FAMM, participants are asked to assess their maturity by filling out whether each practice has either been or is:

- **Implemented**: the practice has currently been implemented in the organization.
- **Implementable**: the practice has not been implemented, but in theory is implementable. Depending on the organizations needs and plans, the practice will either be implemented in the short-term, long-term, or not at all.
- **Not applicable**: the practice has not been implemented, and is not applicable as it will most likely never be implemented. This may be due to a number of reasons. For example, the practice may not be of added value, or not desirable for the organization to implement because it does not align with the organizations goals, vision, or needs.

Lastly, participants are asked to fill out a short survey consisting of a series of questions that are similar to those that were asked as part of the expert interviews during the first evaluation cycle. The main difference is that during the expert interviews, the questions were phrased future tense due to the API-m-FAMM not being completed yet, while during the case studies the questions were posed in past tense. The purpose of these questions is for the participants of the case study to evaluate the API-m-FAMM with regard to its *operational feasibility*, *ease of use*, *usefulness*, and *effectiveness*. These questions are aimed at determining the degree to which the the API-m-FAMM has succeeded in aiding the participating organization in evaluating and improving upon their API management related business processes in practice.

For the embedded single-case study, the API-m-FAMM is applied to two software products that are developed by AFAS Software. Within this case company, the API-m-FAMM was deployed and evaluated with two development teams that are working on two separate software products. Afterwards the evaluation was discussed with the teams as well. We discuss these two products and the highlights from the assessment.

AFAS Software is a Dutch vendor of ERP software based in Leusden, the Netherlands. Additionally, AFAS has offices in Belgium and Curaçao. The privately held company currently employs over 500 people and generated 191 million of revenue last year (2020).

AFAS Profit - AFAS' main software product is Profit, which is an ERP package consisting of different modules such as Fiscal, Financial, HRM, Logistics, Payroll, and CRM. Currently, this product has over 2 million users across over 11.000 small, medium and large organizations. AFAS Profit provides customers with two APIs: a REST API and a SOAP API. Both of these APIs offer the same functionalities, and customers may decide on using either of these depending on their preferences. These APIs are called about 500 million times a month. Furthermore, standard connections with external software products and applications that utilize these connectors, from organizations with which AFAS is partnered, are offered through AFAS' partner portal.

With regard to *Lifecycle Management* the interviewees have marked *Implement Multiple API Versioning Strategy* as not applicable. During the discussion they explained that this is due to the fact that the version of the connectors is directly tied to the version of the application itself. However, in the event of changes, the consumers had to be notified and plan the required changes. In order to notify consumers of updates, Profit publishes release notes describing general updates made to the product as well as specific updates made to the connectors.

Only a few practices from the *Commercial* focus area have been implemented. Consumers are to adhere to a SLA, which contains agreements on fair use, time-out policies, and uptime guarantees. Furthermore, no strategy for monetizing the APIs is employed considering that this is already done indirectly through the product's licensing.

AFAS Focus - Currently, AFAS is developing a new version of its ERP software, which is called Focus. This product shares nothing in terms of the technology and codebase used with AFAS Profit, and is cloud-based as well as generated by using the ontological model of an enterprise as input. Considering that since the time development first commenced some modules such as Financial have been developed, AFAS is currently in the process of transferring customers from the current Profit product to Focus. AFAS Focus only supports REST APIs, which support both the XML and JSON data formats. The endpoints are described using the OpenAPI specification and make use of the OAuth application token flow for authentication. The current size and state of AFAS Focus encompasses about twenty available endpoints, which are directed at a few specific integrations. In the future this number will increase as Focus continues to grow and branch out to more partners.

As AFAS Focus is not yet mature, a number of areas are not yet relevant. While groundwork for *Community* is done, there is not yet a community that needs to be supported. This also results in the underdevelopment of the *Commercial* area. The number of customers do not yet pose strong demands on *Performance*, resulting in a number of advanced practices not being implemented.

The remaining case studies are conducted with multiple organizations. These organizations were partly selected by contacting the experts that were previously interviewed as part of the first evaluation cycle. Furthermore, other organizations were selected through the utilization of the network of the authors. Please note that the names of some of these organizations are anonymized. Explicit consent to include the name of the organization was obtained from those that are not.

ConsultComp - $ConsultComp$ is a multinational professional services network, is one of the Big Four accounting organisations, and is among the largest professional services networks in the world by revenue and number of professionals. $ConsultComp$ provides audit, consulting, financial advisory, risk advisory, tax, and legal services with over 300.000 professionals globally. Aside from these services, the organization also develops software products for customers, which are developed in-house in their office in the Netherlands. The team involved in developing these products utilizes internal APIs, third-party service integrations to access data from service providers, and is also in the process of starting to expose (partner) APIs to customers.

In the scope of the *Lifecycle Management* focus area, the internal API is versioned using the *evolutionary versioning strategy*. Considering that the organization's API is exclusively used internally, practices corresponding to *Update Notification* are not implemented. Noticeably, a large part of the practices corresponding to the *Security* focus area have been implemented. In order to authenticate internal consumers of the API, an authentication protocol along with the SSO method is used. Furthermore, all practices belonging to *Threat Detection & Protection* and *Encryption* have been implemented to further secure the product. Virtually all practices belonging to the *Community* and *Commercial* focus areas have been marked as not implemented or not applicable. Similarly to capabilities such as *Update Notification* and parts of the *Traffic Management* and *Analytics*, this is logical considering that in the current case of an internal API, there simply is no community surrounding it to manage as of yet.

Exact - Exact is a multinational organization that provides ERP software and was founded in the Netherlands. Apart from their headquarters in the Dutch city Delft, Exact also has offices in 20 other countries, and currently employs over 1850 people and annually generates 209 million of revenue. Exact provides customers with various products, such as an integrated ERP package, and a package that incorporates modules that are targeted towards CRM, HRM, and workflow management. Exact's main software product that is offered in the Netherlands and Belgium is called Exact Online, which is a package consisting of modules such as accountancy, CRM, and project management. This SaaS product currently has over 500.000 users and is fully internet-based. Exact Online provides customers with two main API types: a REST API and an XML API. These APIs comprise a range of endpoints that combined are called about 700 million times a month.

All practices belonging to the *Lifecycle Management* focus area have been marked as *Implemented*. Both versioning strategies are marked as implemented, as well as the deprecation protocol and backward compatibility checking practices. The product is also mature with regard to update notification, with the exception of announcing an API versioning schedule. Similarly, most practices in the *Security* focus area have been marked as implemented, including conducting security reviews: over the course of the past few years, Exact has conducted security audits at 2000 organizations. Furthermore, Exact's Zero Trust Network Architecture is implemented through a third party platform. Consumers of Exact Online are provided with an extensive SLA, which contains elements such as uptime guarantees, fair use policies, and agreements on rate and data limiting. Furthermore, while access to the product as a whole is monetized through monthly licensing fees, no monetization model that specifically and

exclusively applies to the APIs is used.

Uber - Uber is a large-scale multinational technology organization that was founded in the United States. It provides multiple services, such as ride-hailing, food delivery (Uber Eats), and package delivery. Uber is estimated to have over 93 million monthly active users worldwide. In 2012, Uber established its international headquarters in Amsterdam, the Netherlands. Here, among other things, development teams are responsible for optimizing the performance and scalability of the public API that is provided by the organization, as well as developing new functionalities.

In terms of the *Lifecycle Management* focus area, nearly all practices have been implemented. The organization is able to version their APIs using the evolutionary strategy, as well as the multiple API versioning strategy. To this end, Uber also has a deprecation protocol and backward compatibility checking methods in place. Furthermore, mechanisms to provide consumers with information regarding updates to the API are in place, with the *announce versioning schedule* practice being planned for implementation in the foreseeable future. Most practices have been implemented for the *Performance* focus area. Considering that Uber is a large-scale multinational organization, this is to be expected. Regarding *Resource Management*, the organization has implemented advanced scaling methods, considering that automatic scaling takes place when certain resource thresholds are exceeded. Furthermore, Uber has runbooks in place that detail what course of action should be taken when the scaling strategy fails. Additionally, considering that the organization has various data centers that are located in different continents and countries, there are *failover policies* in place which allow the organization to mitigate outages.

The overall results of the case studies are presented in Table 4.4, showing the number of practices that are *implemented* for each capability. Consequently, this means that the other implementation categories (*implementable*, and *not applicable*) are not taken into account in this overview. Furthermore, before discussing these results, it should be noted that the percentages representing the average amount of practices that are implemented in each focus area should be interpreted in an indicative manner, as some practices may encompass a much larger amount of work than others. Moreover, some case companies vary heavily in terms of their vision, size, usage of APIs, and goals. For example: Uber, AFAS Profit and Exact may be roughly classified as large and mature organizations that have been exposing public or partner APIs for a long period of time, which is reflected by the observation that these organizations have implemented the majority of practices for most focus areas. $ConsultComp$ is also a large organization in size, but the product that was evaluated has only been utilizing internal APIs for a short period of time. Hence, some focus areas are largely irrelevant within the scope of this product. Similarly, the AFAS Focus product has been utilizing partner APIs for a relatively short period of time and is currently in the process of expanding the services it provides as well as the assets it exposes. Due to the current scale of the product and the stage of development it is in, some practices are currently not yet necessary to be implemented.

First, it was observed that some practices are mutually exclusive. This phenomenon may be observed in the *Authentication* capability, where in some cases the *Implement*

	Focus Area	AFAS Focus	AFAS Profit	*ConsultComp*	Exact Online	Uber	Avg	Stdev
1	**Lifecycle Management**	58%	50%	42%	92%	83%	65%	
1.1	Version Management	2	2	1	4	4	2.6	1.20
1.2	Decoupling API & Application	4	2	4	4	3	3.4	0.80
1.3	Update Notification	1	2	0	3	3	1.8	1.17
2	**Security**	53%	73%	80%	87%	100%	79%	
2.1	Authentication	1	1	2	2	3	1.8	0.75
2.2	Authorization	2	2	2	4	4	2.6	0.98
2.3	Threat Detection & Protection	3	6	6	6	6	5.4	1.20
2.4	Encryption	2	2	2	1	2	1.8	0.40
3	**Performance**	36%	45%	55%	64%	82%	56%	
3.1	Resource Management	3	3	3	2	3	2.8	0.40
3.2	Traffic Management	1	2	3	5	6	3.4	1.86
4	**Observability**	75%	67%	67%	92%	83%	77%	
4.1	Monitoring	2	3	3	3	3	2.8	0.40
4.2	Logging	4	3	4	4	4	3.8	0.40
4.3	Analytics	3	2	1	4	3	2.6	1.02
5	**Community**	28%	78%	6%	94%	78%	57%	
5.1	Developer Onboarding	2	1	0	4	3	2.0	1.41
5.2	Support	2	3	1	3	3	2.4	0.80
5.3	Documentation	1	2	0	3	2	1.6	1.02
5.4	Community Engagement	0	5	0	5	3	2.6	2.24
5.5	Portfolio Management	0	3	0	2	3	1.6	1.36
6	**Commercial**	0%	42%	0%	33%	50%	25%	
6.1	Service-Level Agreements	0	2	0	2	4	1.6	1.50
6.2	Monetization Strategy	0	0	0	0	0	0.0	0.0
6.3	Account Management	0	3	0	2	2	1.4	1.20

Table 4.4: The results of the deployment of the API-m-FAMM at the case companies. For each capability, the amount of implemented practices is shown. The percentages indicate the share of practices that have been implemented out of the total amount of practices that are assigned to the focus area. Please note that these percentages should be interpreted in an indicative manner, as the practices are not weighted, and some practices are more extensive than others.

Basic Authentication has been marked as *not applicable* when the *Implement Authentication Protocol* had been marked as *implemented*. This is indeed a logical result, considering that only one method of authentication is necessary. Similarly, such choices are also observed between the *Implement Interactive API Console* and *Provide Sandbox Environment Support* practices.

Secondly, it may be observed that the mature organizations (Uber, Exact and AFAS Profit) as a whole have implemented the majority of practices across most focus areas. This is particularly the case for *Lifecycle Management*, *Security*, *Performance*, *Observability* and *Community*. However, the AFAS Profit product forms an exception to this observation, considering the relatively high number of practices that are *not applicable* and *implementable*. The practitioners noted that practices that have been marked as such have been discussed internally in the past, and were concluded to not be desirable to be implemented due to them not aligning with the prevalent vision and goal of the product.

Thirdly, we observe that within the earlier mentioned focus areas, some practices on the higher end in terms of maturity are often not implemented across the board, such as *Announce Versioning Roadmap*, *Implement Predictive Scaling*, and *Prioritize Traffic*. This is supportive evidence for these practices having high maturity levels. Furthermore, in most cases such practices were marked as *implementable*, signalling that organizations that have done so are interested in implementing these practices in the future. It should be noted however, that the time-span in which organizations plan on implementing such practices may vary greatly.

Lastly, a common trend that may be observed across all organizations is that the *Commercial* focus area is underdeveloped. In particular, this is the case for the *Monetization Strategy* capability, considering that no organization has marked any of these practices as *implemented*. However, the practitioners at Exact have expressed interest in implementing a monetization strategy in the future. Specifically, the practitioner at Exact mentioned that he expects that monetization for APIs will become increasingly more common in the future, and that he suspects that large-scale organizations that expose high-traffic public APIs already monetize their APIs. However, no concrete evidence of this has been found as part of this case study and its participants.

The practitioners experienced the usage of the API-m-FAMM in a positive manner. As can be seen in Table 4.5, the *ease of use*, *usefulness*, and *effectiveness*, were all ranked with an average score of 4 or higher. This supports the usefulness and effectiveness of the API-m-FAMM in achieving its goal of aiding organizations in assessing their current maturity in API management and guiding them towards achieving higher levels of maturity. However, the average score attributed to the API-m-FAMM's operational feasibility is notably lower when compared to the other criteria. In particular, the scores given by practitioners corresponding to the AFAS Focus and Profit products are among the lowest. This may be explained by the fact that the AFAS Profit product is almost fully matured and developed, which reduces the incentive among its practitioners to repeatedly consult the API-m-FAMM for guidance. This is further compounded due to the practitioners already being aware of most practices that have not yet been implemented, as well as having previously discussed them internally in the past. For the case of AFAS Focus, this may be explained by the observation that a large portion of fo-

Product	Operational Feasibility	Ease of Use	Usefulness	Effectiveness
AFAS Profit	2	4	4	3.5
AFAS Focus	2	3	4	3
ConsultComp	4	4	3	4
Exact Online	4	5	5	4
Uber	3	4	4	5
Average	3.2	4.0	4.0	3.9
Std. Dev.	0.89	0.63	0.63	0.6
Expert Interviews Average	*3.4*	*2.8*	*3.9*	*3.8*
Expert Interviews Std. Dev.	*0.68*	*0.62*	*0.74*	*0.63*

Table 4.5: The rankings given by the case companies' practitioners in response to the questions corresponding to the four evaluation criteria as well as their averages. Included are the average and standard deviation from Table 4.3 for reference.

cus areas and capabilities are irrelevant for the product given its current development stage, which in turn reduces the effectiveness of the API-m-FAMM in this case.

Interestingly, the API-m-FAMM was ranked fairly high by *ConsultComp*'s practitioner, regardless of the fact that, similarly to the AFAS Focus product, some focus areas and capabilities do not apply to the product given its utilization of internal APIs. However, *ConsultComp*'s intent of developing partner APIs in the future may provide an explanation for the practitioner's appreciation of the API-m-FAMM. Regarding the average score given to the *operational feasibility* criterion (Table 4.5), the researchers suspect that this is on the low end due to a lack of incentive for practitioners to re-use the API-m-FAMM after the initial assessment.

4.6 Discussion

The API-m-FAMM has shown to be an effective tool to measure and assess the maturity of an organization with regard to API management. The case studies' participants rank the model with an average of 4 out of 5 for ease of use, usefulness, and effectiveness. In comparison with the ranking of the experts, made during the interviews, only operational feasibility is ranked lower with a 3.2 out of 5 (shown in Table 4.5). This last one is explainable, as two out of five participants remarked that there is no added value of a continuous evaluation with the API-m-FAMM. The rankings given by the experts during the development of the model were a bit lower for usefulness and effectiveness, while being more than a point lower for ease of use. This last criterion is important for the general adaption of the API-m-FAMM and maturity models in general.

Experts occasionally suggested during interviews that they are of the opinion that a bigger range of practitioners could be reached if the accessibility of the API-m-FAMM

as a tool would be improved. One way in which this could be achieved, is by developing a web-app through which practitioners may easily navigate, as well as read focus area, capability, and practice descriptions, and then mark which practices are or are not implemented within their organization. In doing so, the Excel spreadsheet and source document that were used as part of the case studies could be combined into a single easily usable tool. Currently, the API-m-FAMM is maintained on `maturitymodels.org`. While this website offers a visual alternative to the source document considering that descriptions are included, it does suffer from some limitations. Most notably practitioners are unable to use the website to perform an evaluation and denote whether practices have been implemented. Therefore, we published a *do-it-yourself* kit that was used during the case studies. Improving this would allow practitioners to share the current as-is situation of their organization's API management maturity with management and stakeholders. These results could be used as a benchmark for other practitioners through an opt-in consent. The opportunity for improved visualization of results extracted from FAMMs was also previously identified by Spruit & Röling [233]. In this work, it is suggested that effective result visualization may be accomplished through spider charts, since most managers are familiar with such a type of representation.

Another opportunity lies in the potential to customize and adapt the API-m-FAMM depending on certain organizational characteristics and goals. For example, the case studies have shown that certain focus areas are irrelevant for organizations that exclusively utilize internal APIs. For instance, this is visible in the results corresponding to the *Community* and *Commercial* focus areas (Table 4.4). The organizations that scored less on these areas stated that these areas are of less importance for them, because the APIs are either targeted at internal users or the software platform itself is not yet mature enough. The area *Performance* also shows a wide range of results, indicating that the assessed platforms operate in different contexts. Such information could be preemptively collected through a checklist or survey, based on which the contents of the API-m-FAMM may be adapted. Other information that could be used to perform this adaptation may include characteristics such as the size of the organization, whether a third-party API management platform is used, or what type of product or services the organization provides.

The aforementioned opportunity for customization and adaptation of FAMMs can be linked to a general limitation of maturity models that has been identified in previously conducted research. In their work, Proença & Borbinha [198] argue that "*maturity assessments can be used to measure the current maturity level of a certain aspect of an organization in a meaningful way, enabling stakeholders to clearly identify strengths and improvement points, and accordingly prioritize what to do in order to reach higher maturity levels*". However, as a prerequisite to performing such a maturity assessment, evidence needs to be manually collected to substantiate the maturity level calculation, which makes maturity assessment an expensive and burdensome activity for organizations to perform. Hence, Proença & Borbinha [198] argue that future research should focus on developing methods and techniques to automate maturity assessment. The researchers second this observation: even though the maturity assessments that were done as part of the case studies were performed with minimal intervention from the researcher, there are opportunities for automation, customization, and adaptation of

maturity models and FAMMs to reduce the amount of effort needed to perform maturity assessments and provide tailor-made advice for maturity improvement. Moreover, the researchers hypothesize that the aforementioned opportunities for automation, customization, and adaptation could be key in creating incentives for practitioners to re-use maturity models and FAMMs over a longer period of time. We consider the maturity model's low degree of re-usability to be a point of concern due to our findings as part of the case study, which have shown that practitioners are relatively unlikely to revisit the API-m-FAMM to track their progress over a longer period of time.

The researchers are of the opinion that useful insights could be gained by conducting research into the differences in terms of advantages and disadvantages that occur with regards to API management for organizations that actively utilize third-party, commercial platforms to manage their APIs when compared to those who do not. This suspicion is supported by the observation that currently, the largest part of available (grey) literature on the topic of API management is either written by authors that are either directly or indirectly affiliated to commercial management platform providers. Examples of such authors include Weir [266] (director at Oracle) and De [53] (former Apigee consultant). This literature is, more often than not, exclusively focused on the benefits that organizations attain as a result of using API management platforms. However, when asked, some experts that were interviewed during the evaluation cycles noted that their organization does not use a management platform and does not wish to do so. For future work, it should be investigated whether significant differences exist in terms of API management maturity between organizations that do use commercial platforms and those that do not.

4.7 Focus Area Maturity Models

Throughout this work, the design, population, evaluation, and deployment of a focus area maturity model targeted towards the topic of API management has been described. Aside from the main practical contributions the API-m-FAMM offers organizations in maturing their API management practices, this work also provides various scientific contributions in the field of focus area maturity models.

This work provides researchers that seek to develop a focus area maturity model for a different functional domain with an improved description of the existing framework on how to do so. Publications on the development of FAMMs such as that of Jansen [125] and Spruit & Röling [233] offer a high-level overview of this process, which adhere to the FAMM meta-model presented by Steenbergen et al. [236] and De Bruin et al. [54]'s methodology for developing maturity models. The accompanying source data of this article discusses the intermediate versions of the API-m-FAMM and the changes between them in detail [159]. The steps that this methodology consists of were also followed in this work: conducting an SLR, population, evaluation through expert interviews, and deployment as part of case studies. The added details contribute towards alleviating concerns mentioned by Jansen [125]: *"Interestingly enough, while there is a rapid increase of publications of new maturity models, there is little literature that particularly discusses the development of maturity models"*. Furthermore, work conducted by Proença & Borbinha [198] on the state of the art of maturity

models has found that one of the main limitations to maturity models is that there is a general *"lack of information regarding the maturity model development method"*. This article could form an exemplar for other works describing FAMMs.

Because the underlying thought process and specific design choices that were made in developing other maturity models and FAMMs in particular were also largely inaccessible and unknown to the researchers, some novel approaches were used and described throughout this work. A card sorting technique was used and described, which uses digital tools to rank and assign practices to maturity levels. Additionally, the way these exercises were interpreted and the manner in which practices were ultimately assigned to maturity levels is elaborated upon. Another novel approach includes the utilization of criteria used for Design Science Research (DSR) artifact evaluation introduced by Prat, Comyn-Wattiau & Akoka [197] to evaluate the API-m-FAMM's usefulness, completeness, ease of use, effectiveness, and operational feasibility. Doing so has shown that using these criteria is an adequate strategy for evaluating FAMMs during expert interviews or through surveys. In this work, the criteria were used to gather feedback and foster discussion with experts as well as among the researchers themselves, in order to subsequently guide further improvements to the API-m-FAMM. While in this case the criteria evaluation was conducted during the first evaluation cycle and as part of the case studies, the criteria could also be used to evaluate a prototype version of a FAMM to gauge interest among practitioners, or as part of the second evaluation cycle.

This work has shown that De Bruin et al. [54]'s methodology for maturity model development may be incorporated with DSR, by using it to construct a design science artifact during the *solution design* phase. Furthermore, this study demonstrates which techniques and tools may be utilized during the *test* phase of De Bruin et al. [54]'s methodology. This includes some of the approaches listed above, such as usage of Prat, Comyn-Wattiau & Akoka [197]'s evaluation criteria, conducting multiple evaluation cycles through expert interviews to evaluate the maturity model, as well as the usage of Nielsen [175]'s card sorting technique to perform maturity level assignments. As such, this work provides researchers that utilize De Bruin et al. [54]'s methodology and are involved with maturity model development with suggestions for the usage of novel approaches so that they may incorporate them in the testing phase of their maturity models.

Furthermore, the API-m-FAMM was successfully deployed in practice with minimal involvement of the researchers. As is described in Section 4.5, practitioners were provided with the API-m-FAMM, along with a set of instructions and a spreadsheet that was used to denote which practices had been implemented in the case company[3]. To the best of the researchers' knowledge, this study is the first among work that has previously been done with regard to the design of FAMMs where practitioners are enabled to self-assess their organization's maturity in a functional domain such as API management. In comparison, Jansen [125]'s SEG- M2 and Spruit & Röling [233]'s ISFAM were deployed in practice by gathering input through in-person collaboration between the practitioners and the researchers themselves, as well as desk studies. Moreover,

[3]These instructions and spreadsheet are available through the *do-it-yourself* kit, published on `https://www.movereem.nl/api-m-famm.html`.

as is discussed in Section 4.5 and shown in Table 4.5, the majority of practitioners' experience with using the API-m-FAMM was positive, considering that its *ease of use*, *usefulness*, and *effectiveness* was ranked with a score of 4 out of 5 on average.

4.8 Threats to Validity

Like all empirical research, this work is vulnerable to threats to validity. In this section we discuss four categories of validity: *construct validity*, *internal validity*, *external validity*, and *reliability* [4, 212].

Construct validity refers to the degree to which this study actually measures what it is intended to. The API-m-FAMM sets out to evaluate the maturity of organizations with respect to API management practices. In order for this model to be successful and actually measure what we intended to, we need to trust in the application of focus area maturity models in general and the content of the API-m-FAMM specifically. This threat was mitigated through a number of actions. First, the SLR that was conducted as a means of initially populating the first versions of the API-m-FAMM adhered to several constraints. This includes a search string that was constructed through snowballing in an iterative manner. Furthermore, multiple databases were searched and strict inclusion and exclusion criteria were adhered to as to mitigate study inclusion and exclusion bias. Publication bias was further mitigated as a result of the inclusion of grey literature. Additionally, multiple discussion sessions were held among the authors at various stages of the population process in order to mitigate data extraction bias and researcher bias [4]. Moreover, construct validity is mitigated through this work's adherence to design science research (DSR) guidelines and De Bruin et al. [54]'s methodology for developing maturity models. Lastly, the robustness of the initial classification that was used by De [53] was ensured by evaluating this decision through multiple expert interviews. Through these actions the API-m-FAMM builds on existing and proven research from SLRs, expert interviews, and maturity models.

Internal validity is concerned with the extent to which there is evidence that the artifact makes a difference in terms of cause and effect in the context of this study. The API-m-FAMM is intended to achieve both a realistic maturity evaluation as well as an actionable path for improvement. In order to investigate whether the model is able to achieve its intended goal, which is to assist organizations in assessing and evaluating their degree of maturity in API management in order to improve upon their API management related processes, the researchers suspect that multiple desk studies that last substantial periods of time (multiple years) should be conducted. However, verifying the implementation of practices contained in the API-m-FAMM through desk studies is largely infeasible since practices are often placed on long-term roadmaps that are susceptible to change, reducing the chance of actually observing the implementation of such practices during the desk study. Instead, the effects of the API-m-FAMM were investigated through evaluation during the experts' interviews (Table 4.3) and case studies (Table 4.5). Considering the difference and increase in terms of the scores that were assigned to these criteria by practitioners when comparing the first and second versions of the API-m-FAMM, in addition to the positive results and feedback received as a result of the case studies, we believe that the model is able to achieve its intended

goal. In addition to this long-term goal, short-term benefits may also be attained by using the API-m-FAMM due to practitioners being able to immediately identify practices that are not currently implemented.

External validity revolves around the degree to which the results of this study may be generalized and applied to other contexts and situations. We conducted five case studies at four different companies. Based on these studies it is hard to claim that the API-m-FAMM will add value to all companies that expose public APIs. Although we are inclined to believe that our API-m-FAMM is an effective tool, based on the results presented, more studies need to be conducted over a longer time period. We do believe that the research design enables us to deploy the API-m-FAMM in a wide range of organizations. The API-m-FAMM is mainly targeted towards organizations that expose one or multiple public or partner APIs. However, two experts that are employed at organizations that currently exclusively utilize a set of internal APIs were involved in the first evaluation cycle. Considering that these experts expressed that focus areas such as *Security*, *Observability*, *Performance* and, to a lesser extent, some capabilities belonging to the *Lifecycle* focus area, are relevant and applicable to them, we suspect that the API-m-FAMM is also useful to such organizations. This is further supported by the results of the case study conducted at $ConsultComp$. Organizations of varying sizes and backgrounds were involved in the evaluation cycles and case studies. While some experts that are employed at large organizations noted that most practices have already been implemented at their organization, they also commented that some practices that are assigned to high maturity levels have not yet been implemented. Furthermore, experts working for small organizations indicated that the API-m-FAMM could be a useful tool for them to use as a road map or checklist to use in discussions with management and stakeholders so that future implementation of practices that are currently not implemented may be discussed and planned. Additionally, due to the decision to exclude practices that are solely tied to the usage of API management platforms, the API-m-FAMM is generalizable to both organizations that do not use such platforms, and those that do not. However, as this was not the case with organizations involved in the evaluation cycles and case studies, more case studies should be conducted with organizations that heavily utilize API management platforms to fully determine whether this exclusion in terms of scoping has negatively impacted the usability of the API-m-FAMM for such organizations.

The **reliability** aspect is concerned with the extent to which the data and analyses that were conducted as part of this study are dependent on the specific researchers. A large part of the construction of the API-m-FAMM is the result of discussions among the authors. After every phase, as visualized in Figure 4.2, we integrated the new knowledge into the API-m-FAMM through discussions supported by the card sorting technique. However, we designed the research to be as transparent as possible, and published all intermediate versions and source data. Due to adherence to DSR guidelines, the use of De Bruin et al. [54]'s methodology, and the SLR, some degree of researcher bias has been mitigated. Furthermore, the protocols that were used for experts interviews and case studies are included and were reviewed through peer reviewing among all involved researchers. Additionally, all design decisions and processes that resulted in the increments of the API-m-FAMM are extensively documented, along with separate source documents detailing each major version of the model.

Another threat to reliability is that, considering the maturity level assignments described in Section 4.3 were partially done in a pragmatic and subjective matter, they may result in a different outcome if replicated. This is due to the fact that the experts that ranked practices as part of the maturity ranking exercises each have varying backgrounds, experiences, and are employed at organizations with different characteristics. However, this threat was mitigated by using the average of these maturity assessments, as well as analyzing and cross-evaluating focus areas and maturity assignments during the second evaluation cycle. Furthermore, the deployment of the API-m-FAMM as part of the case studies has not produced any criticism among practitioners regarding the maturity levels that practices are assigned to.

4.9 Conclusion

Throughout this work, the design, population, evaluation, and deployment of a focus area maturity model targeted towards the topic of API management has been described. The goal of this model as well as this work in general was to improve the transparency and availability of API management assessment frameworks and tools by constructing, evaluating and validating a publicly available, industry and academically grounded framework or tool that can be used by organizations that expose their API(s) to third-party developers to assess and evaluate their degree of maturity with regards to API management in order to improve upon their API management-related business processes. By constructing, evaluating, and publishing the API-m-FAMM we answered the research question posed in Section 4.3: *How can organizations that expose their APIs to third parties evaluate their API management practices?*. Aside from the main practical contributions the API-m-FAMM offers organizations in maturing their API management practices, this work provides the following scientific contributions.

Firstly, this work offers researchers a previously undefined framework that captures the topics and processes API management consists of. By decoupling API management processes and topics from commercial platforms, the API-m-FAMM offers the academic community a de-commercialized overview of the topic that was developed by using insights from both literature as well as the industry. Secondly, this work provides a detailed description of the construction of a focus area maturity model through the published source data [159], which researchers can use as an example in future work. Third, this work has shown that De Bruin et al. [54]'s methodology for maturity model development may be incorporated with DSR, utilizing Nielsen [175]'s card sorting technique to perform maturity level assignments, conducting multiple evaluation cycles, and utilizing criteria for DSR artifact evaluation introduced by Prat, Comyn-Wattiau & Akoka [197]. Lastly, The API-m-FAMM was successfully deployed in practice with minimal involvement of the researchers using the constructed *do-it-yourself* kit. This shows that we as researchers can make maturity models more relevant for industry by investing in the usability of these assessment and improvement tools.

Data Packages: Systematic Literature Review and Source Data

We describe a Systematic Literature Review (SLR) that has the goal of collecting API Management practices and capabilities related to API Management, as well as proposing a comprehensive definition of the topic. In the scope of this work, a practice is defined as any practice that has the express goal to improve, encourage and manage the usage of APIs. Capabilities are defined as the ability to achieve a certain goal related to API Management, through the execution of two or more interrelated practices. A standard method for SLRs in software engineering is followed, through which we collected 24 unique definitions for the topic, 114 practices and 39 capabilities. A detailed description and the results are made available [158].

The collected practices and capabilities were categorized into six focus areas. Next, the practices and capabilities were analyzed and verified through inter-rater agreement and four validation sessions with all involved researchers. Then, the collection of practices and capabilities was verified by using information gathered from supplemental literature, online blog posts, websites, commercial API management platform documentation and third-party tooling. As a result, the initial body of practices and capabilities was narrowed down to 87 practices and 23 capabilities. These practices are described by a practice code, name, description, conditions for implementation, the role responsible for the practice, and the associated literature in which the practice was originally identified. Capabilities and focus areas are described by a code, description and, optionally, the associated literature in which it was originally identified. The complete data set is made available [159].

5

API Management Maturity of LCDPs

Low-code development platforms are environments that enable citizen developers without software engineering knowledge to create software products. These software products range from small business applications to large business platforms, around which software ecosystems increasingly form. In these software ecosystems, different organizations want to extend the created software products with services and software, with the goal of creating active enterprise networks that create value collaboratively. Well designed and maintained application programming interfaces are crucial for these organizations.
In this paper we evaluate the application programming interface management maturity of four low-code development platforms. We show that these platform providers are not yet concerned with helping their customers build software ecosystems around the software platforms that citizen developers create. Furthermore, we identify the software engineering research challenges that these platform providers face. For instance, low-code development platforms should create abstractions that let citizen developers design, develop, and manage application programming interfaces. If low-code development platform providers follow our advice and act on it, they will become able to provide customers with complete ecosystem-enabled platforms instead of providing only simple throwaway business applications.

This work was originally published in *Enterprise, Business-Process and Information Systems Modeling. BPMDS 2021, EMMSAD 2021. Lecture Notes in Business Information Processing, vol 421. Springer, Cham.*, titled 'API Management Maturity of Low-Code Development Platforms'. It was co-authored by Max Mathijssen and Slinger Jansen.

5.1 Introduction

Increasingly, traditional Software Producing Organizations, i.e., organizations whose main activities include the production of software, such as software vendors and open source organizations, are discovering platforms as a vehicle to increase the value of their software for their customers through collaboration with third parties. This transformation from a product towards a platform is called 'platformization' [193]. Platforms are a vehicle for software ecosystems and are defined as a set of organizations collaboratively serving a market for software and services [126]. These ecosystems form around software platforms, which in turn are managed by software platform orchestrators.

We find that not all software products can easily transform into software platforms. One particular category of software products are products that are created using no-code/low-code development platforms (LCDPs). LCDPs apply model-driven development to raise the abstraction level of software development, increasing productivity and decreasing complexity as a result [28]. They target a wide variety of users, from professional software developers to non-technical business experts [182]. The latter category is commonly referred to as the 'citizen developer': people without software development education who nonetheless build software applications. Customers of these LCDPs utilize these platforms to increase their agility, it enables them to develop business applications more efficiently without suffering from the lack of professionally trained software developers. While LCDPs are traditionally used to create agile business software applications, they are increasingly becoming part of the core IT landscape [219]. As long as the LCDPs prove their value, customer companies will utilize them in more and more diverse projects. The customer companies will be interested in evolving their application into a platform to enable complementors, which are organizations that build applications that extend the core system, to create more value [126].

Evolving into an ecosystem is done by offering parts of the developed application through Application Programming Interfaces (APIs). Research shows that concerns such as stability, security, and scalability in ecosystems are related to API management capabilities [6]. API management is the activity that enables organizations to design, publish, and deploy their APIs for (external) developers to consume. This is one of the enabling practices for the creation of an ecosystem. Traditionally the activities within API management are executed by technical staff members. The strength of LCDPs, however, is that activities that used to be executed by highly technical staff, such as software engineers, are now executed by citizen developers. To support these API management activities LCDPs have to provide the means to the citizen developers to integrate applications developed on the LCDP with other applications.

We believe that LCDP providers must mature their API management capabilities to remain relevant for citizen developers. After all, software ecosystems are increasingly seen as the way to remain strategically relevant in the software industry [125]. In this paper, we distinguish between two software ecosystems that form around LCDPs. First, the LCDP ecosystem is the ecosystem that forms around the LCDP and the provider of the platform. The second is the LCDP Application Ecosystem, which forms around the

application that is created by one of the customers of the LCDP. Our contribution is an evaluation of the maturity of API management capabilities support among LCDPs, along with a set of challenges that we identify. We evaluate the capabilities of four LCDPs through descriptive case studies. In these case studies we measure the maturity of their API management capabilities through a Focus Area Maturity Model (FAMM). A FAMM is a model that groups capabilities and practices in focus areas and aligns them with maturity levels, which can be used to evaluate organizational practices around particular focus areas [25, 125].

Section 5.2 describes our research method and gives a short description of the four LCDP providers investigated in our evaluating case studies. The API management FAMM (API-m-FAMM) is described in Section 5.3. In Section 5.4 we evaluate the LCDPs to uncover their level of support for a citizen developer in API management activities. The results of the case studies are analyzed in Section 5.5. We contribute a set of engineering research challenges for software engineering researchers in Section 5.6. Threats to validity are discussed in Section 5.7. In Section 5.8 we observe, through the four case studies, that some LCDPs are slowly maturing their API management capabilities while others do not act on these opportunities. Furthermore, we observe that the API-m-FAMM, while mostly aimed at organizations with a proprietary API infrastructure, is also applicable to LCDPs. Finally, we conclude that these platform providers need to increase the level of abstraction of the technical complexity of managing APIs, without losing the strategic strengths of it.

5.2 Research Method

Our research focuses on the question: *How mature are the API management capabilities that LCDPs offer?* We use an established framework for evaluation of API management maturity, to evaluate how well applications, created with four LCDPs, enable API capabilities for the LCDP customer.

In our research we apply a FAMM to measure the maturity of API management in LCDPs. The API-m-FAMM [159] captures API management in 81 practices, grouped in 20 capabilities, which are in turn assigned to six focus areas. The model is intended for organizations that develop their own proprietary API infrastructure. However, in our case studies we evaluate the LCDPs to uncover how they support the API management activities of their customers. We apply a maturity model to measure the current state and provide a roadmap to advance this state, similar to the work of Feijter et al. [79]. Please note that more details are provided on the API-m-FAMM in the next section.

The evaluations are performed in four descriptive case studies, which were conducted with the ACM SIG Empirical Research standard in mind [204]. To be able to extrapolate our findings we selected platforms that represent the current state of LCDPs. Our selection was made based on the Quadrant for Enterprise LCDPs of the advisory company Gartner [256]. This report surveys 18 platforms and categorizes them into leaders, visionaries, challengers, and niche players. We selected two leaders and two visionaries that were willing to cooperate in our evaluation. From Gartner's report we can conclude that leaders and visionaries will show the state of the art in

Enterprise LCDPs and represent the most advanced platforms.

The case studies were conducted with the following steps. First, public sources such as product documentation and company blogs were studied to create a first impression of the API management capabilities of the LCDP. Next, an interview with a company expert (either a product manager, architect, or chief technology officer) was conducted. During the interview the API-m-FAMM [159] was described to the interviewee, including all of the practices and their terminology. Together with the interviewee the API-m-FAMM was used to assess the LCDP. Finally, the interviewees discussed their LCDP with respect to API management, and their opinion on creating platforms on top of the LCDP. Subsequently the interviews were processed and analyzed. Based on this analysis and together with the company documentation the evaluation of the LCDP using the API-m-FAMM was completed. Afterwards the evaluation was shared with the interviewee to correct mistakes and oversights. Finally, our general findings were discussed with the interviewees, to establish how they perceive the role of APIs in their generated applications and whether they equally acknowledge the trend of 'ecosystemification'.

We shortly describe the case study organizations here. *Mendix* is a worldwide operating low-code platform provider, founded in the early 2000s. The company employs 1,000 employees, serving thousands of customer companies and an ecosystem of almost 150,000 developers. *OutSystems*, as the second largest and oldest provider, has been active for almost 20 years. With well over 1,000 employees worldwide, they serve a large range of companies. Their LCDP originated as a rapid application development platform. *Betty Blocks*, founded almost 10 years ago, employs around 200 people. Operating worldwide, they serve customers in all business domains. This LCDP has a strong focus on the citizen developer in enterprises, as reflected in their vision: 'anyone should be able to build an application.' *Pega* is the oldest (40 years) and biggest (6,000 employees) company of the four. Its LCDP evolved from a business process modeling suite.

5.3 Introduction of the API-m-FAMM

The goal of the API-m-FAMM[1] is to support organizations that expose their APIs to third-party developers in their API management activities. Using the API-m-FAMM, organizations may evaluate, improve upon and assess the degree of maturity their API management processes have.

A focus area maturity model [235] consists of focus areas, and an area consists of capabilities, which are defined as *the ability to achieve a certain goal related to API management, through the execution of two or more interrelated practices.* A practice in turn is defined as *an activity that has the express goal to improve, encourage and manage the usage of APIs.* The API management maturity model is created following the steps described by Steenbergen et al. [235] and Bruin et al. [25]. The scope, design, and populate phase are based on a systematic literature review [158]. The model was further refined through two rounds of interviews with experts: nine interviews in the

[1] A detailed description and the source data are published [159]. The model is also available on the `https://MaturityModels.org` web site.

first round and three interviews in the second round. Finally the model was used to asses five different software products.

The API-m-FAMM consists of six focus areas that we briefly summarize here:

- **Lifecycle Management**: An API undergoes several stages over the course of its lifetime [162]. Version management is particularly challenging: complementors in the ecosystem benefit from stable APIs, but at the same time demand new functionality to further their own product.
- **Security**: APIs provide access to valuable and protected data and assets. Therefore, mature APIs implement the latest security standards, such as the *OAuth 2.0* authorization protocol, and protection against threats such as Denial of Service attacks.
- **Performance**: APIs deliver data and services to complementors in the ecosystem. This increases the demand for APIs to perform well under load: the application itself as well as the complementors are negatively affected by a decrease in performance.
- **Observability**: An organization benefits from insight into the API's usage. Through various monitoring techniques, the organization is able to collect metrics which can shed light on the API's health and performance, as well as its usage by complementors. A performant and healthy API is crucial, because an interrupted service of the APIs will also most likely interrupt the complementors application.
- **Community**: It is desirable for organizations to foster, engage, and support the community that exists around the API. This entails offering developers the ability to register for API access and offering them access to test environments, code samples, and documentation.
- **Commercial**: Exposing and consuming APIs can have a commercial aspect tied to it [53]. On the one hand, APIs can require a subscription fee from the complementors, on the other hand complementors might demand Service Level Agreements from the provider.

These focus areas are composed of 20 capabilities, which in turn comprise 81 practices. Within their corresponding capabilities, which may be regarded as sub-topics, practices are ranked based on the perceived complexity of their implementation. In order to verify whether an organization has implemented a practice, a set of conditions for implementation has been defined for each practice. By examining the fulfillment of the aforementioned implementation conditions, it may be determined whether an organization has implemented a practice. When this is done for each practice a capability consists of, an organization's *maturity level* for that capability may be determined.

We provide a description of the practice *Implement Multiple API Versioning Strategy* here, to clarify how the practices are evaluated. The description of this practice is *"The organization has a versioning strategy in place which entails the process of versioning from one API to a newer version. In order to do so, the organization must be able to maintain multiple versions of (one of) their API(s) for a period of time. Possible strategies include URI/URL Versioning (possibly in combination with adherence to the Semantic Versioning specification), Query Parameter versioning, (Custom) Header versioning, Accept Header versioning or Content Negotiation."* Each practice has an *Implemented when*

text, that describes one or more conditions to evaluate whether a practice has been implemented or not. In this case the condition is self-explanatory: “The organization utilizes one of the following versioning strategies: URI/URL Versioning, Query Parameter versioning, (Custom) Header versioning, Accept Header versioning or Content Negotiation.” For the LCDPs, we discussed whether it is possible to maintain different versions of the API, or whether an API always co-evolves with the model and does not have any kind of evolution mechanisms implemented to enable different API versions.

5.4 Case Studies

This Section describes the four evaluations of the LCDPs that were done with the API-m-FAMM. This assessment was done based on the available platform documentation and the interview. The interviewees were able to point out mistakes in the evaluation, comments were incorporated accordingly. First we describe the LCDPs in general, then we discuss the six focus areas and how the LCDPs support these.

Mendix - The road map of *Mendix* shows a focus towards enabling citizen developers to create increasingly complex applications, which is motivated by two developments. First of all, applications developed on the LCDP are growing and becoming increasingly complex. Second, an increase in demand from citizen developers to build integrated applications independent from professional developers is observed. Strong API management capabilities enable customers to split their large applications into smaller integrated applications. The envisioned central API catalog will bring together applications within an enterprise, enabling citizen developers to develop integrated solutions.

OutSystems - The focus of *OutSystems* is ensuring that the co-development between the citizen developer and the professional developer is made as efficient as possible. Their vision is ‘fast and agile development of enterprise applications’. API management is not hidden behind abstractions, but rather placed in the hands of professional developers. The gap between technical API management and the citizen developers is not actively bridged, instead the co-development between citizen developers and professional developers is promoted.

Betty Blocks - This LCDP is focused on consuming APIs, instead of publishing them. *Betty Blocks* states that their LCDP is not used to develop core systems, but to develop supporting applications. Applications mostly complement existing systems. The runtime of the LCDP consists of a web-based server and browser-based client application. Developers do not have to explicitly design APIs, as every application feature is an API by default through this architecture.

Pega - Ease of change and rapid application development by collaborating departments in enterprises are the focus of *Pega*. The developer tool of the LCDP supports multiple personas, both the citizen developer as well as the professional developer. *Pega* supports a myriad of integration options, besides REST and SOAP APIs it also supports integration through database connection or e-mail. Despite plentiful options to integrate with other applications, *Pega* has observed that a large portion of their customers does not use these capabilities to integrate with complementors outside of the organization.

Through the API-m-FAMM evaluation we measure the state of API management support that the LCDPs offer. Considering that the API-m-FAMM is targeted towards organizations that expose their APIs to third-party developers, the evaluation of LCDPs differs from this original intention. A LCDP is both an application (run-time) platform and a development platform. API management practices can be implemented in different ways in an LCDP:

- A practice can be **statically implemented** by the LCDP, meaning that the customer cannot influence it (an example is *Load balancing*).
- **Variable** implemented practices are those practices that can be influenced by citizen developers or professional developers, such as the *Multiple API Versions Strategy*.
- Some practices can only be implemented by using products from **third-party** vendors. An example is the *Adopt Subscription-Based Monetization Model* practice.
- The LCDP is a development environment and in that capacity the LCDP can be used to implement a number of API management practices. Examples of these **build-your-own** practices are *Community Forum* and *Broadcast API Status*.

The last two categories, third-party and build-your-own, result in more work for the customers of the LCDP. They become responsible for developing and maintaining these specific practices, while statically and variable implemented practices do not have these liabilities. Therefore, we evaluate the four LCDPs by scoring the practices in two categories: *supported* (by the LCDP) and *custom* (developed) practices. The first category consists of all statically and variably implemented practices, third-party and build-your-own practices are grouped in the second category. Table 5.1 shows the results per API-m-FAMM capability (details are made available [184]). Every score consists of two numbers: first the number of practices that are supported by the LCDP, then the number of practices that need to be custom implemented.

Focus Area: Lifecycle Management - Generally speaking, the LCDPs support both the consumption and publication of modern APIs. Standard protocols such as SOAP and REST are supported by all LCDPs. *Mendix*, *OutSystems*, and *Pega* also support the API protocol OData. The decision of *Betty Blocks* to create APIs automatically shows a strong opinion on APIs. Some practices are not implemented, because their choice for GraphQL based APIs enforces APIs without versions. Their LCDP customers cannot implement a versioning strategy, considering that API consumers always use the latest version, and that customers need to take care of backward compatibility. The other capabilities, *Decoupling API & Application* and *Update Notification*, show great resemblance between the four LCDPs. In the first capability all practices are supported by the LCDPs, while customers are expected to *custom* implement most practices in the second capability.

	Focus Area	M	O	B	P
1	**Lifecycle Management**	**7/5**	**8/4**	**6/4**	**8/4**
1.1	Version Management (*4 practices*)	2/2	3/1	2/1	3/1
1.2	Decoupling API & Application (*4 practices*)	4/0	4/0	3/0	4/0
1.3	Update Notification (*4 practices*)	1/3	1/3	1/3	1/3
2	**Security**	**12/4**	**12/4**	**10/4**	**12/4**
2.1	Authentication (*3 practices*)	3/0	3/0	2/0	3/0
2.2	Authorization (*4 practices*)	4/0	4/0	4/0	4/0
2.3	Threat Detection & Protection (*6 practices*)	3/3	3/3	2/3	3/3
2.4	Encryption (*3 practices*)	2/1	2/1	2/1	2/1
3	**Performance**	**6/5**	**6/5**	**8/2**	**7/4**
3.1	Resource Management (*4 practices*)	3/1	3/1	3/1	3/1
3.2	Traffic Management (*7 practices*)	3/4	3/4	5/1	4/3
4	**Observability**	**5/7**	**5/7**	**5/7**	**5/7**
4.1	Monitoring (*3 practices*)	0/3	0/3	0/3	0/3
4.2	Logging (*4 practices*)	4/0	4/0	4/0	4/0
4.3	Analytics (*5 practices*)	1/4	1/4	1/4	1/4
5	**Community**	**10/8**	**9/9**	**10/8**	**8/10**
5.1	Developer Onboarding (*4 practices*)	4/0	4/0	4/0	4/0
5.2	Support (*3 practices*)	1/2	1/2	1/2	0/3
5.3	Documentation (*3 practices*)	2/1	2/1	2/1	2/1
5.4	Community Engagement (*5 practices*)	0/5	0/5	0/5	0/5
5.5	Portfolio Management (*3 practices*)	3/0	2/1	3/0	2/1
6	**Commercial**	**2/10**	**2/10**	**2/10**	**2/10**
6.1	Service-Level Agreements (*4 practices*)	2/2	2/2	2/2	2/2
6.2	Monetization Strategy (*4 practices*)	0/4	0/4	0/4	0/4
6.3	Account Management (*4 practices*)	0/4	0/4	0/4	0/4
	Total	**42/39**	**42/39**	**41/35**	**42/39**

Table 5.1: Evaluation of the API management maturity of the four LCDPs according to the API-m-FAMM: *Mendix* (M), *OutSystems* (O), *Betty Blocks* (B) and *Pega* (P). For every API-m-FAMM capability we show the total number of practices, and the LCDP evaluation. The two numbers per LCDP stand for practices *supported* by the LCDP and practices that need to be *custom* developed respectively.

Focus Area: Security - In the area Security there is almost no differentiation between the LCDPs. All of the LCDPs follow modern security standards such as *Implement Transport Layer Encryption*, *Implement Authentication Protocol*, and *Implement Access Protocol*. The platforms support their customers in most practices.

Only the capability *Threat Detection & Protection* has a number of advanced practices that need to be implemented by the LCDP customers: *Security Breach Protocol*, *Conduct Security Review*, and *Implement Zero Trust Network Access*. While the providers have implemented these practices for their own hosted services, customers are responsible for their own protocols and are thus required to implement these practices as well.

Focus Area: Performance - Once again the LCDPs are similar in their support of

the practices in this area. In the *Resource Management* capability we observe that the LCDPs implement most of the practices. *Load Balancing*, *Scaling*, and *Failover* are all supported by the providers. Advanced practices in *Traffic Management* are mostly left to be *custom* implemented by the LCDP customers. The LCDP customers are required to configure third-party applications that implement practices such as *Manage Quota* and *Prioritize Traffic*. Implementing these practices requires the expertise of professional developers and thus extra investment from the customers.

Focus Area: Observability - In this area there is no difference between the LCDPs. The practices in the capability *Logging* are supported by all four platforms. Monitoring the health, performance and resource consumption of APIs is left to the customers. *Custom Analysis Reports*, *Status Broadcasting*, and *Alerts* are also left to be custom implemented.

Focus Area: Community - Practices from the area *Community* that focus on technical capabilities, such as *Software Development Kit Support* and *API Catalog* are supported by the LCDPs. All of the LCDPs implement the API specification language OpenAPI, which supports practices such as *Use Standard for Reference Documentation* and *Provide SDK Support*. Less technical practices, such as *Social Media Presence* and *Communication Channel* are left to the customers to implement. Some of these practices, such as *Community Form*, can be built on top of the LCDP.

Focus Area: Commercial - The area *Commercial* is underdeveloped in all four LCDPs. Developers are not able to monetize the APIs developed on top of the LCDP, nor are they able to construct custom Service Level Agreements towards their API consumers. In order for customers to implement these practices they are required to integrate with third-party API management solutions.

5.5 Analysis of the Results

The four LCDPs under study show a great resemblance when evaluating their API management maturity. The fact that they are all either leaders or visionaries in the Quadrant for Enterprise LCDPs makes this no surprise, considering that they are ranked similarly. However, what is surprising is the general lack of support for advanced API management practices.

Mendix supports 42 practices, leaving 39 practices to be implemented by their consumers, making it an almost 50-50 split. The roadmap, as discussed during the interview, shows a focus on supporting their customers in the API management activities. This support is aimed at making it easier for citizen developers to build and publish APIs of higher quality. This roadmap focuses on the practices in the *Community* focus area. The area *Commercial* is not on the roadmap, making it harder for customers to monetize their APIs.

OutSystems generally supports the same practices as *Mendix*, but made it clear during the interview that their ambitions differ. They have a strong focus on creating a platform that enables the development of core enterprise systems by teams consisting of both citizen and professional developers. In this vision, there is no need to support all practices, because the provider recognizes that their customers already have several enterprise API platform solutions in place. Therefore, although there is a mature plat-

form, many of the API management practices are left to their customer to implement themselves.

Within *Betty Blocks* (supporting 41 practices), publishing APIs is possible, but the capabilities are not mature enough to build a platform. In agreement with their vision, the LCDP can be used to complement other applications, but is not as suitable for creating core systems. This is caused by two main reasons. First of all, their opinionated implementation of API versioning through GraphQL limits customers in how they want to expose APIs to their complementors. Second, *Betty Blocks* focuses less on the implementation of practices with third-party vendors. This makes it hard to implement practices such as *Prioritize Traffic*.

Pega (supporting 42 practices) is focused on letting their consumers build richer applications with their platform. These richer applications require integration with other applications. However, the focus of *Pega* is on in-house company projects that integrate within the company, or are complemented by selected organizations and partners. Companies are not supported in building an open platform that attracts complementors: capabilities for advanced community engagement or monetization strategies are not supported.

Overall the focus of the LCDPs appears to be on building enterprise applications, and less on platforms or even ecosystems. All LCDPs show that they have developed mature platforms, with support for modern standards in security and resource management. Through their implemented practices they enable their customers to develop and publish modern APIs that can be consumed by complementors. However, looking at the areas *Community* and *Commercial*, which contain less technical practices, we observe a gap. Many of the more advanced capabilities that customers can use to build platforms and attract complementors, such as *Monitoring*, *Analytics*, *Community Engagement*, and *Monetization Strategy*, are left to their customers to implement. The LCDPs support around 50% of the practices, leaving the other 50% to be implemented by their customers. In the evaluation of the API-m-FAMM with non-LCDP ecosystems we encountered five products that implemented respectively 42%, 59%, 42%, 77%, and 79% of these practices. Three out of five of these products are more mature than the support offered by the LCDPs, meaning that customers would have to implement a number of practices themselves to build comparable products with one of the LCDPs.

Not all providers agree with our belief that they should support API management activities to enable their customers to create platforms. By not implementing these practices, and leaving them to be custom implemented, their customers have to invest more effort in building a platform on top of their LCDP. The providers miss the opportunity to support better API management for citizen developers. Instead they obligate citizen developers to seek help from professional developers to complete these tasks. This creates a dependency from citizen developers on these professional developers, and misses the opportunity to put more power in the hands of the citizen developers, democratizing software development even further. As claimed in the Gartner report [256] LCDPs improve productivity and reduce the time to market, and because of the shortage of developers, democratizing development would offer a possible solution. However, the current state of these LCDPs does not enable citizen developers to create platforms, without requiring the support of professional developers. Raising

the abstraction of API management practices could and should be the next step for the LCDPs.

5.6 Engineering Research Challenges for LCDPs

The previous section discussed the current state of API management support among LCDPs, measured with the API-m-FAMM. Even though we provide LCDP providers with engineering and product planning direction through this evaluation with the API-m-FAMM, there are still several research challenges that hamper further progress in this domain. These challenges are based on our observations made during the case studies and on the authors' experience with software ecosystems and LCDPs. They are based on the capabilities that show the highest number of practices that require a *custom* implementation, and common remarks extracted from the interviews. We outline these engineering research challenges here and provide several solution directions.

Life Cycle Management - Citizen developers will be constructing new application extensions and releasing them to customers, probably without regard for software and data complementors who use a previous version of the application. Citizen developers need to be made more aware of the effects of data model and interface changes on the software ecosystem surrounding the application. While typically these problems would be solved through abstraction, it is practically impossible for citizen developers to remain ignorant of the effects of software evolution on interfaces with third parties in the ecosystem. This can be accomplished through, often complementary practices such as versioning policies, backward compatibility, publishing road maps, and change notifications [53, 112, 162].

The LCDPs support impact analysis within the platform, knowing the relations between different components. However, novel solutions to support analysis of the impact on applications outside of the LCDP are necessary to support the citizen developer.

Performance - Considering that the created applications will be approached through different channels than the traditional user interface, novel architectures are required that can handle large volumes of traffic through other channels, such as APIs. Architecture styles such as Microservices [121] offer a possible solution to these scalability challenges. Of course, an important requirement is the abstraction that citizen developers should be offered.

Observability - LCDPs should be able to handle an increase in users, while still providing the citizen developer with control over who uses the API, how much the API is used, and how the API is used. API gateways [53] traditionally provide these controls to professional developers and operational staff, but now need to be supported by the LCDP itself and usable by non-technical users. The citizen developers need to have access to API usage metrics and statistics to ensure that they too can identify misuse and monitor traffic from the citizen developer's partners [266].

Community - As the community around a product starts growing, complementors need to be supported as much as possible. Such capabilities are for example enabling citizen developers to generate API access credentials for complementors, infrastructures for communicating with complementors, as well as providing application stores around a generated product. All studied LCDPs leave the development of these com-

munity practices to the citizen developers.

The abstractions provided by LCDPs should give citizen developers enough control over API documentation and usage, while automatically adding technical documentation such as SDKs and source code examples. The studied LCDPs offered this through the use of standardized specification languages, such as the OpenAPI specification.

In the past, research has been conducted on the generation of APIs [194]. However, a number of related practices, such as *Provide FAQ and Code Samples* and *Provide Start-up Documentation*, are only offered through consumer built solutions. These practices are challenging to support due to an ever-evolving generated object model. Without support from the LCDP the developer is responsible for evolving the manual written documentation together with the model of the API. This will lead to mistakes that hurt the community.

While the term 'citizen developer' indicates that it has been the goal to open up software engineering to people without formal software engineering education, we can hardly claim that this has been successfully accomplished for API management. The complexity of modern software solutions and the inherent simplification required to create LCDPs are constantly in direct conflict with each other. The platformization trend lays this bare and shows that new models and perspectives are required to truly make software engineering accessible to any citizen developer. We see it as future work to design new abstractions that make LCDP solutions simpler and more powerful in supporting API management practices.

5.7 Threats to Validity

In this paper we present four descriptive case studies that we conducted based on interviews and documentation. Through these case studies we evaluate the current state of API management support offered by LCDPs.

Our conclusions are threatened by concerns regarding the generalizability of these four LCPDs when compared to the LCDP industry as a whole with respect to API management maturity. We cannot deny that there could be an LCDP that we did not study that supports more, or even all, API management practices. However, given that we studied four major platforms that are recognized as such in the Quadrant for Enterprise LCDPs report [256] confirms that we have studied a representable group. While the API maturity evaluation might not be generalizable to other LCDPs, these four platforms are recognized as the most innovative in the industry. Given that there might be a provider that has a more mature support of API management practices only confirms that these leaders and visionaries are missing out on opportunities to further support their customers. Providers with less mature support of API management practices make our call to action only more pressing.

Another threat to validity of this research are the evaluations of the LCDPs based on the API-m-FAMM. Wrong or imprecise evaluations based on documentation and interpretation could distort the conclusions. The fact that the interviewees reviewed and corrected the evaluation mitigates this risk. We believe that the general evaluation of the LCDPs with respect to API management, combined with the vision of the LCDP provider gives a truthful representation of the current state of API management matu-

rity. Our findings and conclusions are based on the global state of API management support of the LCDPs, and do not depend on specific practice support.

In our research we focused on the LCDPs and their API management capabilities. Although we discussed a number of organizations that built an internal platform on top of the LCDP, we did not discuss specific example platforms. We did not specifically search for an example, but rather focused on the general state of API management maturity. Future work should study existing platforms built on LCDPs to further understand what opportunities LCDPs have.

5.8 Conclusion

Our case studies, as presented in Section 5.4, evaluate four LCDPs using the maturity model API-m-FAMM. Our research was guided by the research question: *How mature are the API management capabilities that LCDPs offer?* We conclude that these LCDPs support around 50% of the practices described in the API-m-FAMM. The other practices are left to be implemented by the customers of the LCDPs. We conclude that only *Mendix* places API management firmly on its road map. Both *Betty Blocks* and *Pega* do not observe a demand for API management capabilities among their customers, and neither are they promoting these capabilities. *OutSystems* recognized the demand, but has not yet focused on providing more of these capabilities to their customers. Instead they defer much of the work to either third-party vendors or the LCDP customers. By not supporting these practices we believe that LCDP providers miss out on the opportunity to further democratize software development. They instead require citizen developers to solicit the support of professional developers to develop platforms that are open for other companies to extend.

We draw the following conclusions from this work. First, we suspect that LCDP providers will soon be challenged in providing capabilities that enable citizen developers to transform their applications into platforms. Our research shows that LCDP providers are currently unable to support such capabilities for citizen developers and require technical staff to implement such architectures and mechanisms through either third-party solutions or custom solutions built on top of the LCDP. Second, we conclude that as LCDPs are becoming more powerful, they can use the API-m-FAMM to evaluate and update their road maps. Finally, we identify five engineering challenges that, if solved, will create a next generation of citizen developers who can independently create complete software platforms and software ecosystems, and subsequently manage them without the requirement for highly specialized technical knowledge.

Data Package: Evaluations of Four LCDPs

We evaluated the API management maturity of four low-code development platforms using the focus area maturity model API-m-FAMM. Based on these evaluations we discuss the state of API management capabilities in low-code platforms. The evaluations of the four platforms are made available [184].

Part IV

Evolution Supporting Architecture

6

Generative versus Interpretive MDD: Moving Past 'It Depends'

Model-driven development practices are used to improve software quality and developer productivity. However, the design and implementation of an environment with which software can be produced from models is not an easy task. One part of such an environment is the model execution approach: how is the model processed and translated into running software? Experts state that code generation and model interpretation are functionally equivalent. However, a survey that we conducted among several organizations shows that there is a lack of knowledge and guidance in designing the execution approach. In this article we present the results of a literature study on the advantages of both interpretation and generation. We also show, using a case study, how these results can be utilized in the design decisions. Finally, a decision support framework is proposed that can provide the guidance and knowledge for the development of a model-driven engineering environment.

This work was originally published in *Pires L., Hammoudi S., Selic B. (eds) Model-Driven Engineering and Software Development. MODELSWARD 2017. Communications in Computer and Information Science, vol 880. Springer, Cham.*, titled 'Generative versus Interpretive Model-Driven Development: Moving Past 'It Depends". It was co-authored by Slinger Jansen and Sven Fortuin.

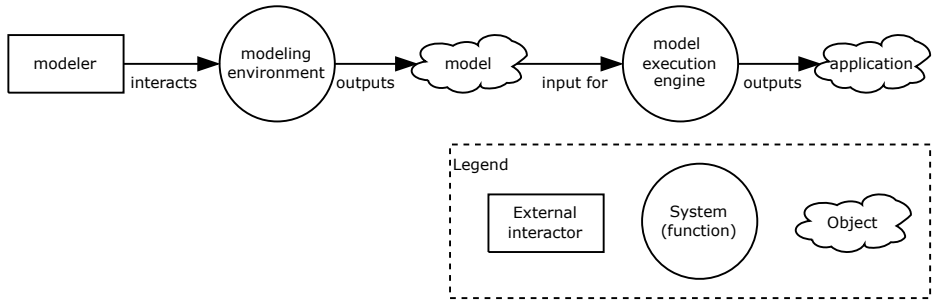

Figure 6.1: A model-driven engineering environment enables a modeler to create a model in a *modeling environment*. The model is subsequently translated by the *model execution engine* (using a *model execution approach*) into an application.

6.1 Introduction

Model-driven development (MDD) is used by software producing organizations (SPOs) to improve software quality and developer productivity. According to Díaz et al. [59] these improvements in quality and productivity are achieved because a well designed model raises the abstraction level of the software development process. The abstracted model allows for an expressiveness that can be more concise than general-purpose programming languages. Domain-specific modeling improves that even further by catering the model to a certain domain. The expressiveness causes both the increase of productivity (more can be done with less) and the quality (there will be fewer mistakes, because there is a smaller model). The models can be used in different manners, Brown [24] shows a modeling spectrum with, among others, roundtrip engineering, model-centric, and model only. We are especially interested in the model-centric approach: the model is the source of truth and the application follows from the model. The model-centric approach is implemented in Model Driven Engineering Environments (MDEE), an environment that is similar to an Integrated Development Environment (IDE) used for software development. Modelers create models using modeling languages in a specific modeling environment, just as developers write software in their IDE. These models are translated according to well-defined semantics, into an application. Together these components (from the modeling environment up to and including the application) form the MDEE (visualized in Figure 6.1). The translation process that reads the model and produces an application is defined as the *model execution approach*, and implemented in the model execution engine.

Our experiences are that the development of an MDEE is by no means an easy task. The initial investment is large, because there are many technical challenges. One of these technical challenges that is of particular interest to us, is the design and development of the *model execution approach*. SPOs can choose for code generation, run-time interpretation, or a hybrid form that combines both approaches (Figure 6.2). As in every design challenge, there are numerous decisions to make (with their specific trade-offs) that influence the overall quality of the MDEE.

Just like any other (architectural) design question, the design questions for the model execution approach can be answered with “it depends". In this article we show that the design depends on desired quality characteristics and the context of the MDEE. Moreover, we show how SPOs can take these characteristics into account. It might be regarded as an implementation detail, but the model execution approach, like any other component in the system, has its influence on the quality characteristics (such as run-time behavior and maintainability) of the whole system. As in any system that consists of multiple components working together, the model execution approach should not be designed individually (i.e., not out of the context of the MDEE). The influence of the model execution approach is similar to, for example, the influence of a specific database on the quality of a data-intensive system. While users may not see a difference in functionality between two different databases, the quality of the system is affected by it, for example, in terms of performance, stability, and availability. SPOs can deliver the same functionality, whether they choose code generation or run-time interpretation, but the quality of the MDEE will differ significantly.

The main research question of this article is *How can SPOs make an informed decision between a generative or interpretive model execution approach?* In Section 6.2 we explain the different model execution approaches in more depth, and discuss the work already done in this area. We motivate our research question in Section 6.3 by presenting the results of a survey among SPOs applying model-driven development. This survey shows that there is no “one size fits all” solution. It also shows that many SPOs do not have a clear rationale for the model execution approach that is used. Therefore, decision support and clear guidance are necessary to improve the overall design and implementation of MDEEs. Section 6.4 discusses the results of the literature study that we have performed on the advantages and disadvantages of the generative and interpretive approach. There are many hybrid model execution approaches that combine the generative and interpretive approach. We show the preference for the two pure approaches in terms of percentages. These percentages can be used by the SPO to find the right balance in designing their own hybrid model execution approach. Section 6.5 describes a case study, in which we observe the design of a fitting model execution approach. We conclude that the design of a fitting model execution approach is not detached from the overall design of the MDEE. We reflect on the case study and our observations in Section 6.6. We observed three general areas of design decisions that influence the model execution approach, and we present a design support framework based on the case study. Finally, Section 6.7 and Section 6.8 evaluate and discuss the study, and present our conclusion respectively.

6.2 Context and Related Work

There are several model execution approaches, many of which are a hybrid form of the two pure approaches. We discuss the two pure approaches, and describe two groups of hybrid approaches, shown in Figure 6.2. The first pure model execution approach is *code generation*. During code generation a model is parsed, interpreted and transformed into source code. The generated source code generally results in running software. This approach is not exclusive to MDD, and is formalized and defined by

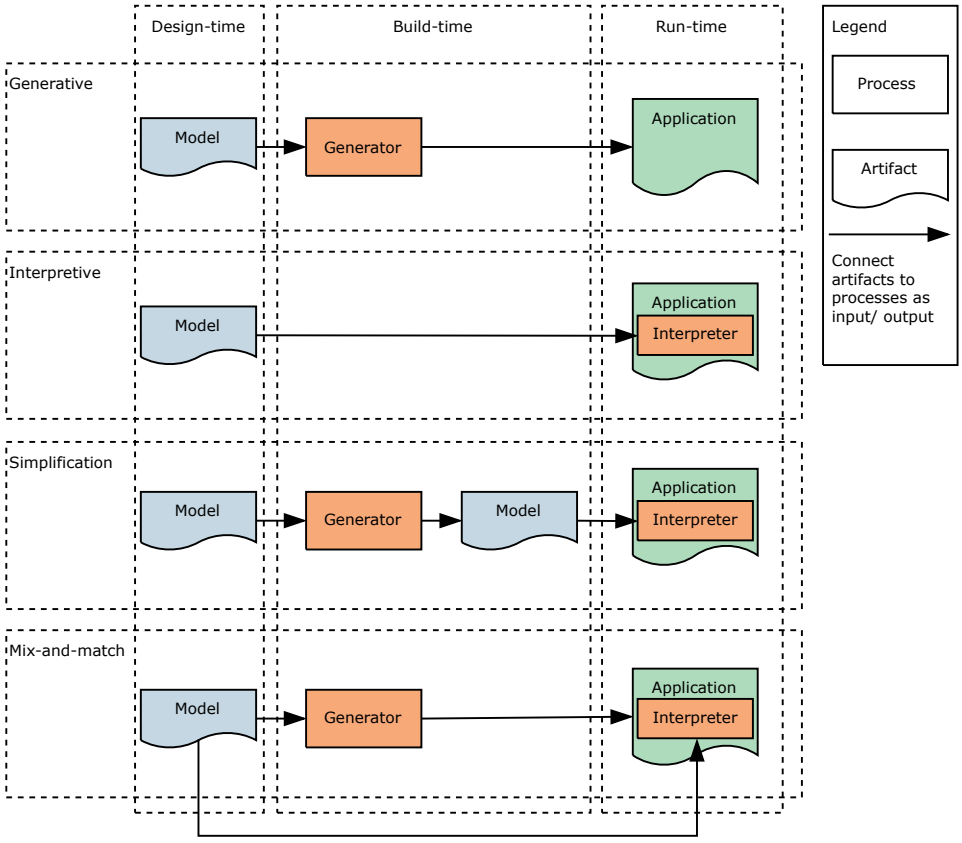

Figure 6.2: The four main types of execution approaches are generation, interpretation, simplification, and mix-and-match. The darker boxes show the execution process. With the two hybrid approaches, the execution process can be split up and divided between build-time and run-time. The model is created at design-time, but is used at build-time and/or run-time.

Czarnecki & Eisenecker [49] as *Generative Programming*. According to their definition, it is a paradigm based on modeling facilities used to automatically manufacture customized and optimized intermediate and/or end-products. Applying generative programming within MDD results in generative MDD. Although nothing in the definition states that the output cannot be changed manually before the final software product is delivered, we only regard *full* code generation that does not need manual changing of the generated code. This does not imply that every part should be generated; the generated code can be combined with frameworks or base libraries, as pointed out by Kelly & Tolvanen [136].

The second pure model execution approach is *run-time interpretation*, or interpretive MDD. The idea is similar to code generation, but the timing is different: the parsing and interpretation of the model are done at run-time. There is no need to first generate source code, the running software executes its functions directly based on

the model. In this case the model execution approach becomes part of the application, the application interprets the model before offering functionality based on the model. Further manual coding is not possible with this approach, because there is no time to intervene in the execution of the software. However, as we see in Section 6.3 it is possible to combine custom code with an interpreter. The model needs to be deployed along with the running software, while in the generative approach, the model is not part of the running software.

These two approaches form the extremes of the execution spectrum, and many hybrid forms are possible. We see two groups of hybrid approaches. The first group is *simplification*: a model is transformed into a second model before deploying it for run-time interpretation. In this approach there is both a generation step and an interpretation step, instead of generating source code. The generation step transforms the model into a second model that can be interpreted at run-time. This can be achieved by transforming high-level concepts into low-level concepts, or by transforming into a model with fewer constructs. The results of this approach are manifold: **1)** The interpreter is easier to develop and better maintainable, because it has to support fewer constructs. **2)** The translation is less complex, so the interpreter is faster. And finally, **3)** the interpreter becomes more reusable, because there can be many different models that can be transformed into the intermediate model. This approach is also used by programming languages that compile into an intermediate language that is in turn interpreted by a runtime environment, such as the approach Meijler et al. [163] discuss. They generate Java source code, but use a customized class loader that acts as a run-time interpreter.

The second group of hybrid approaches is a *match-and-mix* approach: some parts of the platform use code generation, while others use interpretation. This approach can be used both from an architectural perspective as well as from a model perspective. The MDEE could use a different approach in different components, for instance, the user interface could be interpreted, while the database access layer is generated. Different approaches could also be chosen based on model dynamics, where the more stable parts can be generated into source code and the more dynamic parts are interpreted.

Figure 6.2 shows the four described approaches, marking out the time at which the execution takes place. In the generative approach, the execution is done at build-time, as opposed to the interpretive approach in which the execution takes place at run-time. Both the simplification and mix-and-match approaches show that they have part of the execution at build-time, and part at run-time. This makes them flexible, because SPOs can decide how much happens at what time. These hybrid approaches can also be combined; the mix-and-match approach can combine the interpretive, generative, and simplification approach into a single encompassing model execution approach.

There is already some work done on the challenge of designing a fitting model execution approach, which is discussed in this article. A multi-criteria analysis of the different approaches is performed by Batouta et al. [13] with as goal to support of the decision-making. Their analysis results in a decisive statement about the best approach (based on their list of ten criteria). However, they do not take the context of the MDEE into account. Fabry et al. [77] address a number of advantages regarding

the different model execution approaches, but they do not give any support for the decision-making. Zhu et al. [279] research the decision-making within MDD applied to game development, however, they only look at other architectural decisions rather than the model execution approach within a MDEE. Code generators and the interaction with developers is researched by Guana & Stroulia [101], only without making a comparison with the interpretive approach. All of the mentioned work is incorporated in the literature study in Section 6.4.

The design of software and its architectures is a thoroughly researched topic. Kruchten, Capilla & Duenas [147] show how design decisions play a role in software architecture, and that it is important to capture them. Jansen & Bosch [122] define "software architecture as the composition of a set of architectural design decisions", and formalize this in the Archium approach which is further extended by Ven et al. [254]. Svahnberg et al. [239] present a decision process that, based on desired quality attributes, supports an SPO in finding the architecture variant that shows the most potential. We combine the definition of software architecture as a set of design decisions with the approach to support a decision with quality attributes, and apply this to MDD. Because of this we are able to uncover the rationale of either a generative or interpretive approach, and support SPOs in their design process.

6.3 How SPOs Design and Develop MDEEs

We interviewed 22 product experts of 16 different SPOs that develop MDEEs. All of the experts had either five or more years experience with the product or were working with the product since its start. They served in different roles at the time: 12 of them as chief executive, the others in different roles such as lead developer, business developer, and sales manager. These experts were asked questions on the design and implementation of their company's MDEE. The SPOs were identified by an Internet search, exploiting our network, and asking interviewed product experts.

We identified 36 qualifiable case companies with representatives in Belgium, The Netherlands, or Luxembourg. For 16 companies we found experts that were willing and able to cooperate in our research[1]. The companies differ in size (ranging from ten employees to thousands of employees), in market (some operate only in The Netherlands, while others operate worldwide), and maturity (some MDEEs are almost twenty years old, while others only two years). After we processed the answers, every expert had the opportunity to correct any mistaken interpretations. The answers are summarized in Table 6.1.

The first topic of interest is the target users of the MDEE and its modeling language. We asked the experts what the target group of users for the MDEE is, and what kind of expertise they expect from them. Their answers resulted in four categories of users:

- **Laymen** are people without any technical knowledge.
- **Technical business users** are those that have some knowledge of software development, but are no developers. They are expected to have knowledge about software concepts such as data models, and data types. An informal description

[1]Some needed to be excluded due to confidentially issues or the lack of (technical) knowledge.

	SPO_1	SPO_2	SPO_3	SPO_4	SPO_5	SPO_6	SPO_7	SPO_8	SPO_9	SPO_{10}	SPO_{11}	SPO_{12}	SPO_{13}	SPO_{14}	SPO_{15}	SPO_{16}	
Company size																	
0-50 employees	✓	✓	✓	✓			✓					✓		✓	✓	✓	56%
100-500 employees						✓		✓	✓	✓			✓				31%
+500 employees					✓						✓						13%
Development years																	
0-5			✓						✓			✓					19%
6-15	✓	✓		✓						✓					✓	✓	37%
+15					✓	✓	✓	✓			✓		✓	✓			44%
Target platforms																	
Web	✓	✓	✓	✓	✓	✓*	✓	✓*	✓	✓	✓	✓	✓	✓*	✓	✓	100%
Desktop				✓		✓							✓				19%
Mobile													✓				6%
Target users																	
Laymen	✓			✓					✓			✓			✓		32%
Technical business users		✓	✓		✓			✓		✓	✓		✓	✓		✓	56%
SQL experts				✓													6%
Developers	✓	✓				✓	✓			✓						✓	37%
Model execution approach																	
Interpretation	✓	✓	✓	✓		✓	✓			✓			✓	✓	✓	✓	69%
Generation	✓	✓				✓		✓	✓		✓	✓	✓			✓	50%
Simplification					✓				✓								13%
Match-and-mix	✓	✓				✓			✓				✓			✓	38%

Table 6.1: Anonymized results of the survey among SPOs. The target users and the model execution approach are shown, along with the company size (in terms of number of employees) and maturity (in terms of number of development years). The cells marked with an * identify MDEEs that support two distinct web platforms.

would be people more knowledgeable than layman, but less knowledgeable than developers.

- **SQL experts** are a specific set of users that are able to write SQL queries. They are not able to write software in other programming languages. This specific category was added after the review with SPO_4, because the category **developers** did not match their target description.
- **Developers** are those users that are able to write software in a programming language. MDEEs that target developers expect them to be familiar with IDEs and other programming concepts.

A third of the SPOs specifically target laymen, while the others require some form of technical knowledge of their users. There is no correlation found between the model execution approach and the targeting of laymen. In the case study we conducted (see Section 6.5) we also observed the design of an MDEE that targets laymen. A third of the SPOs that target technical business users also target developers, their MDEEs support custom programming, because the model is not able to express all required functionality. The six SPOs that target developers all use an interpretive approach, four of them also use code generation.

Five SPOs (SPO_2, SPO_6, SPO_9, SPO_{13}, and SPO_{16}) state that a reason for their model execution approach is a certain required build-time behavior. As an example, the expert of SPO_{16} states "*You can't generate code again in an end application that is already generated. To allow workflow modeling in the end application, we were forced to make use of an interpretative solution.*". All of the five mentioned SPOs explain that users are able to change the model, and expect that their changes are (near) instantly applied and visible in the application. Four of them use run-time interpretation, while the other one uses a simplification approach. The SPOs that use a generative approach did not mention such a requirement for build-time behavior.

All of the SPOs target a web platform, meaning that they support at least back-end and front-end applications. Three of the SPOs, however, support two different back-end platforms, one also supports mobile applications, and two others also support native desktop applications. Effectively, we can conclude that all SPOs support multiple platforms. The interpretive approach is motivated three times by the advantage of platform independence, or portability.

We have found little reasoning behind the implemented approaches, one expert even stated "*We just had to go with one of the two.*". An expert of SPO_6 refers to an advantage in portability for interpretation, a correlation that we will see again in Section 6.4: "*By interpreting the UI and generating the remaining parts of the application, we are able to share models between different platforms.*". A reference to resource utilization is made by an expert of SPO_{14}: "*We don't want to regenerate an entire database every time the model changes, because this can potentially cause a lot of problems with data migration.*". The interviews show that all approaches are used, and nearly half of them use a hybrid form. This supports our claim that the model execution approach depends on many factors and is context-specific. We cannot give a simple answer such as "web platforms should use an interpreter", Table 6.1 shows that other approaches are used for web platforms as well. Like Kruchten, Capilla & Duenas [147] we believe that it is important for SPOs to document the rationale behind important architectural

decisions. In the next section we will show that the model execution approach influences the quality of the MDEE, and that it is important to capture the rationale of the design.

6.4 Quality Characteristics of Model Execution Approaches

We started the literature study by executing a literature review on the advantages and disadvantages of both code generation and run-time interpretation. The literature review was done with the snowballing approach as described by Wohlin [270]. The snowballing approach uses references between articles as a means to discover other relevant literature. The first step is to select a start set from which the references can be followed. This approach was chosen because the research areas to be covered in this review are broad. We expected literature from the MDD field as well as Domain-Specific Language engineering and compiler design. The second reason was that the literature that we had found in earlier explorations never mentioned the advantages or disadvantages directly; these were often hidden in implementation details.

Our start set was created by earlier informal explorations with the Google Scholar engine, using "interpretation versus code generation" and "interpretation vs. code generation" as keywords. We selected five articles as the start set [56, 163, 166, 243, 259]. These papers represent the different research areas and have a broad research question, resulting in many references (both backward and forwards). With this start set we executed several steps, following both backward and forward references. The literature found was included when it mentioned advantages or disadvantages of model execution approaches; we ended up with 35 studies.

The literature was classified using the ISO standard 25010:2011 for software product quality [118]. This standard is used to assess the quality of software systems, and matches our intent to assess the quality of MDEEs. The ISO standard consists of eight categories with 31 characteristics. We found evidence for differences in quality fulfillment for five out of these eight categories, summarized in Table 6.2. The summary of all the evidence found is presented in Table 6.3. There was no evidence found for the categories *functional suitability*, *usability*, and *reliability*. The first two categories match the statement of Stahl et al. [234]: "code generation and model interpretation are functionally equivalent". For the category *reliability* no evidence was found as well, which was expected. Reliability is the degree to which the system performs its functions under certain conditions. We assume the generative and interpretive approach to be functionally equivalent, and in both approaches it is possible to build a reliable functioning system.

Table 6.3 presents every mention of an advantage or disadvantage in relation to its source and quality characteristics. A *G* stands for a preference of generation over interpretation, while an *I* stands for the opposite. When we encountered statements on the two approaches without a preference, we marked the corresponding cell with both *G* and *I*. The total number of preferences is used to calculate the percentage of the two alternatives with respect to the quality characteristic. The evidence we found is presented in relation to the generative and interpretive approach. This does not

Category	Characteristics
Functional suitability	Functional completeness, Functional correctness, Functional appropriateness
Performance efficiency	*Time behaviour*, *Resource utilization*, Capacity
Compatibility	*Co-existence*, *Interoperability*
Usability	Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics, Accessibility
Reliability	Maturity, Availability, Fault tolerance, Recoverability
Security	*Confidentiality*, Integrity, Non-repudiation, Accountability, Authenticity
Maintainability	*Modularity*, Reusability, *Analysability*, *Modifiability*, *Testability*
Portability	*Adaptability*, *Installability*

Table 6.2: The categories and characteristics from the software product quality model in ISO standard 25010:2011. For the emphasized items we found evidence of a preference for either code generation or model interpretation.

mean that the hybrid approaches are not mentioned by the authors (as discussed in Section 6.2). However, the advantages and disadvantages we found were always in terms of the generative and interpretive aspects of an approach.

6.4.1 ISO: Performance efficiency

The characteristics in the category *Performance efficiency* describe the performance of a system: how the system utilizes resources, responds to requests, and meets the capacity requirements. For two of the characteristics evidence was found.

Time behavior - The first characteristic for which we found evidence is the time behavior of the system. For MDEEs, this is a special characteristic, because there are two main use cases for which the response and processing time are important. The run-time time behavior describes the response time of the functionality offered in the application. However, the second important use case for which response time is important, is the translation from model to application. When a generative approach is used, the model execution approach takes up time between model changes and software updates. When an interpretive approach is used, there is no time between model changes and software updates because the execution happens during the execution of normal system functions. These two distinct use cases are confirmed by the literature we studied: we found comments in relation to both approaches. Therefore this characteristic is split into two separate characteristics. Both build-time time behavior and run-time time behavior are used as two separate characteristics in our study.

22 out of the 32 papers mention the time behavior characteristic, it is one of the most frequently commented characteristics. Because of the possibility of doing upfront analysis during code generation, more efficient code can be generated. On the other hand, interpreters add overhead to run-time functionality and thus are slower. While that is the general sentiment, Klint [142] remarked that the overhead of interpreters will diminish with the advent of hardware. Both Ertl & Gregg [72] and Romer et al. [210] show that there is nothing that makes interpreters inherently slow.

The reduced build times that an interpretive approach results in are an advantage;

	Run-time time behavior	Build-time time behavior	Resource utilization	Co-existence	Interoperability	Confidentiality	Modularity	Analysability	Modifiability	Testability	Adaptability	Installability
[13]	G	I	G			G			I		G	G
[20]	G						I	I	I	I		
[38]							G		G I			
[42]	G	I					I		I			
[43]	G		G						I			G
[44]		I							I			
[49]	G	I										
[59]	G	I							I	G		
[72]	G I	I							I		I	
[77]	G		G		I				I			
[91]	G			I								
[98]	G		G I					I	I	I	I	
[101]								I	I	I		
[109]										I		
[116]							I					
[129]	G							I	I	G		
[131]		I		I				I				
[142]	G I		G					I	I	G I	I	
[163]	G	I	G							G	G	G
[166]							I		I	G		I
[179]	G	I			I					G		
[192]										G	I	
[207]		I										
[210]	G I											
[226]		I										
[234]		I				G		I	I	I		
[238]								I		I	I	I
[243]	G	I				G					I	I
[244]	G	I				G					I	I
[247]	G								I			
[246]		I							I			
[252]	G				I					I		
[259]	G	I						G	G	G	G	G
[260]	G	I						G	G	G		
[280]	G	I	G							G		
% generation	88	0	87.5	0	0	100	20	20	15	55.5	30	50
% interpretation	12	100	12.5	100	100	0	80	80	85	44.5	70	50

Table 6.3: The results of the literature review and basis for the ranking of the two approaches. *G* corresponds with a preference for code generation over interpretation. *I* identifies a preference for interpretation over generation. *G I* indicates where advantages or disadvantages where given without a clear preference.

they enable agile development and better prototyping. This advantage is stated by Consel & Marlet [42] and Riehle et al. [207] among many others.

Resource utilization - The general comment that code generation results in improved run-time behavior can be extended to resource utilization as well. Meijler et al. [163] state that generators can optimize for more than run-time behavior only, something that is useful in, for instance, embedded systems and game environments. A difference can also be seen in how generators or interpreters compete with the running application for resources. A generator might use more memory, but could be running on different hardware than the application. Interpreters are part of the application, so it could be hard to run them on different hardware. Gregg & Ertl [98] comment that interpreters often require less memory, but confirm the competition for resources with the application.

Another view on resource utilization is the data storage for an application. Meijler et al. [163] point out that the interpretive approach often leads to a less optimal data schema. The schema might depend on the model and thus can change at run-time, therefore, the schema has to be flexible enough. This requirement often conflicts with optimizations that might be achievable otherwise.

6.4.2 ISO: Compatibility

The category *Compatibility* contains characteristics that express the quality of co-existence and operability of the system.

Co-existence - Only two papers contain evidence for a preference between interpretation or generation based on this characteristic. Gaouar, Benamar & Bendimerad [91] share their experiences in making dynamic user interfaces and point out how the interpretive approach enabled them to use native platform elements. A different side is shown in Jörges [131]: the late binding that interpretation offers makes it possible to re-use the same application instance for different tenants.

Interoperability - Interpreters have access to the dynamic context of the application at run-time. Fabry et al. [77], Ousterhout [179], and Varró, Anjorin & Schürr [252] state this as a preference for interpreters, because it allows them to communicate with the application in a way that is not possible by generators.

6.4.3 ISO: Security

Security describes the quality in terms of integrity, authentication, and confidentiality. The literature only contained evidence for the characteristic confidentiality.

Confidentiality - Tanković [243] and Tanković, Vukotić & Žagar [244] describe the models used in a MDEE as intellectual property. The interpretive approach exposes the model to the application, making it more vulnerable to exposure. In the generative approach the models do not need to be shipped, which makes that approach more secure.

6.4.4 ISO: Maintainability

Maintainability is an important aspect in the quality of software products. Characteristics in this category that were mentioned by literature comment on the testability,

modifiability, analysability, and modularity of the platform.

Modularity - Most literature favors interpreters over generation when looking at the modularity characteristic. Inostroza & Storm [116] and Consel & Marlet [42] propose solutions for modularization within interpreters. Cleenewerck [38] is the only one who argues that generators are more preferred than interpreters when it comes to modularization.

Analysability - An important aspect in MDEEs is the analysis of the resulting application. It should conform to the model and the defined semantics, which is not an easy task. When a generative approach is used, the model is translated into a separate language, without losing the semantics of the model. Proving that translation is correct is hard, according to Guana & Stroulia [101]. According to Jörges [131], the interpreter can play the role of a reference implementation, used to document the semantics of the model. This improves the analysability of the platform.

Debugging is partly analyzing the run-time behavior of an application. According to Voelter [259] and Voelter & Visser [260] this process is easier in a generative approach, because the generated application can be debugged as if it were a normal application.

Modifiability - Many, such as the works of Cook et al. [43] and Díaz et al. [59] among others, claim that interpreters are easier to write. We conclude that easier to write software is also easier to modify. Cordy [44] describes the process of a compiler as being heavy-weight, making it harder to modify. Cleenewerck [38] and Voelter & Visser [260] argue that generators give more freedom to developers, giving them room for better solutions.

Testability - The literature was far from conclusive on the testability of both approaches. On the one hand, interpreters can be embedded in test frameworks, this makes them easier to test. Generators on the other hand add indirection in the testing, because they are a function from model to code. Asserting the correctness of the output becomes fragile when just looking at the written code, the easiest way is to determine the correctness by running the code. Voelter [259] and Voelter & Visser [260] prefer generation when it comes to debugging, because the model translation can be left out of the testing.

6.4.5 ISO: Portability

Portability covers the characteristics of adaptability and installability.

Adaptability - The separation between generation environment and application environment makes the generative approach preferred according to Meijler et al. [163], Batouta et al. [13], and Voelter [259]. The two environments can be evolved at a different pace when adaption needed, which makes it more flexible. In an interpretive approach the whole interpreter needs to be rewritten and although this might be easy, it is more work. However, Tanković [243], Tanković, Vukotić & Žagar [244], and Gregg & Ertl [98] state that porting an interpreter to a new platform is no problem when platform independent technologies (such as programming languages and environments that run on multiple platforms) are used. This matches the results from Section 6.3, where three SPOs stated portability as the rationale for the interpretive approach.

Installability - The two separated environments in the generative approach not only have a clear advantage for adaptability, they also have an advantage with respect to installability. Meijler et al. [163], Cook et al. [43], Batouta et al. [13], and Voelter [259] prefer code generation because it can target any platform, it does not constrain the target application. The initial installation is, however, not all that is important, when the MDEE is updated, re-installations are needed too. The interpretive approach makes re-installations less frequent, because in many cases only the model needs to be updated. This advantage is pointed out by Tanković [243] and Mernik, Heering & Sloane [166].

6.4.6 Utilizing the Preferences

The results of the literature study as presented in Table 6.3 can be used by SPOs to design their execution approach. But before SPOs can use these results, they have to prioritize the quality characteristics; i.e., they have to determine which characteristics are most important for them. When priorities are assigned, the preference for either the generative or the interpretive approach can be calculated by the following formulas:

$$P_{generative} = \sum_{i=1}^{12} P_i G_i \quad and \quad P_{interpretive} = \sum_{i=1}^{12} P_i I_i$$

The formulas summarize all twelve characteristics i, and apply the priority (P_i) on the corresponding preference (from Table 6.3) for both the generative (G_i) and the interpretive (I_i) approach. All priorities add up to a total of 1, and because for every characteristic i G_i I_i add up to 100%, $P_{generative}$ and $P_{interpretive}$ add up to 100%. The outcome shows for a certain set of priorities what the preference for either the generative or interpretive approach is.

How the priorities are determined is not prescribed, however, in the case study described in the next section we will show two possibilities. The first option is by informally giving a weight to every characteristic, dividing 100% among the different characteristics. By doing this informally, the SPO takes the risk of calculating a preference with inaccurate data. Therefore, we also show a second option to prioritize the characteristics: the Analytic Hierarchy Process (AHP) method described by Saaty [214]. Falessi et al. [78] show that the AHP method is helpful in protecting against two difficulties that are relevant for this study. The first is a too coarse grained indication of the solution. When the priorities are determined informally it becomes easy to overlook certain characteristics. The second difficulty is that there are many quality attributes that need to be prioritized, and many attributes have small and subtle differences. The AHP method helps by prioritizing in a pairwise manner, the priorities are only determined relative to other characteristics.

6.5 Case Study

We conducted a case study by observing the design of an MDEE at a Dutch SPO, AFAS Software. The NEXT version of AFAS' ERP software is completely model-driven, cloud-based and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model (OEM, see Schunselaar et al. [227])

will be expressive enough to fully describe the real-world enterprise of virtually any business. The platform initially used a generative approach, generating many lines of C# and JavaScript. However, during the course of 2016 a shift was put into motion towards a hybrid form with more parts being interpreted at run-time. We took part in the discussions surrounding this shift and observed the team while they designed and implemented parts of the MDEE.

We already explained that the context of the MDEE influences the design of the execution approach. This can be seen if we approach the architecture as a set of design decisions as described by Jansen & Bosch [122] and Ven et al. [254]. These decisions are made during the software development life cycle. Every requirement is satisfied by first creating one or more solutions, from which the SPO selects the best fitting alternative. This is done by assessing the solutions, for instance in terms of quality, cost, and feasibility. After a solution is selected, the preferred solution is incorporated into the existing architecture. This process is continuous and will be repeated for every new requirement that needs to be satisfied.

The complete architecture of an MDEE is too large to present in this paper, therefore, we present the most important and guiding requirements and decisions. These are presented in two distinct phases, to illustrate two different utilizations of the results from Table 6.3. The requirements and decisions that form the architecture and are input for the prioritization are summarized in Table 6.4.

Requirements
R1 Target audience for the modeling language are laymen
R2 Users do not manage or maintain the MDEE themselves
R3 Cost effectiveness of the MDEE is important
R4 Use a technology that the developers are familiar with
R5 The MDEE should handle the load from the existing customer base
R6 End users can change the model without intervention

Decisions
D1 Develop an ontological enterprise model
D2 Use a SaaS delivery model
D3 Use multi-tenancy to gain resource sharing
D4 The MDEE should run on the .NET runtime
D5 Deploy the MDEE as a distributed application
D6 Use a hybrid execution approach

Table 6.4: Summary of the requirements and decisions from the design of the MDEE.

The initial requirement that guided the design of the MDEE is the envisioned target audience for the modeling language (**R1**). By choosing laymen as the target audience, it becomes possible for non-technical business users to model their own ERP solution. This requirement is driven by years of experience in the development of an ERP solution, and the knowledge that is accumulated in those years. The resulting design decision is that the modeling language should be a model with a high level of abstraction, an ontological enterprise model (OEM) (**D1**). This model abstracts from

the many details that are needed for creating software, those details are added by the platform (the generator or interpreter) when the model is transformed. A second requirement is that the hosting and management of the MDEE is done by the SPO (**R2**). Delivering the MDEE through a Software-as-a-Service (SaaS) model is the second design decision (**D2**) that satisfies requirement **R2**. A third important requirement is cost effectiveness of the MDEE (**R3**), and multi-tenancy is one of the ways of achieving that as stated by Kabbedijk et al. [132]. The decision for a variant of multi-tenancy forms the last important decision (**D3**) of this initial phase.

After the design of the initial architecture, which solved among many other requirements **R1**, **R2**, and **R3**, the execution approach is designed. At the time of this design, the literature study as presented in Section 6.4 was not yet done. After discussion with the team, we concluded and verified that in hindsight four quality characteristics were especially important for this phase of the development. *Run-time time behavior* and *resource utilization* followed from the decision for SaaS (**D2**) and multi-tenancy (**D3**). *Testability* and *analysability* were important to AFAS in ensuring the quality of the new MDEE. The data in Table 6.3 and the priorities that we assigned in hindsight allow us to calculate the preference for an approach. The possible calculation is shown as an illustration. The first two characteristics (*resource utilization* and *run-time time behavior*) are assigned a priority (or weight) of 35%, the other 30% is split between the other two characteristics (*testability* and *analysability*). The resulting preferences can then be calculated by combining the priorities of the characteristics with their weights (expressed in percentages, summing up to a total of 100%). We apply formulas $P_{generative}$ and $P_{interpretive}$ to the percentages from Table 6.3 and the priorities, resulting in the following calculations:

$$P_{generative} = 0.35 * 0.88 + 0.35 * 0.875 + 0.15 * 0.55.5 + 0.15 * 0.20 = 0.729$$

$$P_{interpretive} = 0.35 * 0.12 + 0.35 * 0.125 + 0.15 * 0.44.5 + 0.15 * 0.80 = 0.271$$

The outcome of the calculation matches the decision that AFAS made, and their initial execution approach was the generative approach. This initial phase of requirements, decision making, and design of the architecture can be summarized in three statements.

- **R1** leads to **D1**
- **R2** in the context of **D1** leads to **D2**
- **R3** in the context of **D1** and **D2** leads to **D3**

As the design of the MDEE advanced new requirements needed to be realized. First of all, the technology that is used to develop the MDEE was selected. The requirement was that a technology should be used that is familiar to the development team (**R4**). This fourth requirement led to the decision for the .NET runtime (**D4**) as the technology to develop the platform on. The next requirement formulated expected load requirements: AFAS has a large existing customer base that needs to be transferred to this new platform. There is an expected load known from the existing customer base that needs to be handled (**R5**). As a result of this requirement, the decision was made to design and deploy the application as a distributed system (**D5**).

The sixth requirement reopened the design of the model execution approach. Therefore, the team decided to backtrack on the earlier decision for the generative approach.

	Priority	Generative	Interpretive
Run-time time behavior	0.059	0.88	0.12
Build-time time behavior	0.278	0.00	1.00
Resource utilization	0.098	0.875	0.125
Co-existence	0.045	0.00	1.00
Interoperability	0.012	0.00	1.00
Confidentiality	0.012	1.00	0.00
Modularity	0.062	0.20	0.80
Analysability	0.023	0.20	0.80
Modifiability	0.150	0.15	0.85
Testability	0.085	0.555	0.445
Adaptability	0.155	0.30	0.70
Installability	0.021	0.50	0.50
Preference		0.293	0.707

Table 6.5: Summary of the priorities of quality characteristics determined by applying AHP as described by Saaty [214]. Columns *Generative* and *Interpretive* show the preferences for code generation and model interpretation from Table 6.3. The final preferences are calculated with the formulas $P_{generative}$ and $P_{interpretive}$.

AFAS envisioned that customers are able to customize the model without intervention from AFAS (**R6**). This requirement leads to other requirements, such as the expected turn around time between model changes and application updates. Based on requirement **R6** and the decisions **D1-D5** the quality characteristics were prioritized. Characteristics *build-time time behavior*, *adaptability*, and *modifiability* became more important. This time the prioritization was done by applying the AHP method: all the characteristics were pair-wise compared and ranked according to the method described by Saaty [214]. The results are shown in Table 6.5, combined with the preferences from Table 6.3. The final outcome preferred interpretation over generation with 71%.

The team decided to implement a simplification approach: the OEM is simplified into a simpler model by the generator. This way the team was able to satisfy the build-time time requirements, without sacrificing performance. Because the MDEE itself had already grown quite large, the team decided to also switch to a mix-and-match approach. The simplification approach was first implemented in a specific component: the messages that are passed between the different parts of the distributed system.

An architecture consists of many decisions, both large and small, both important and non-essential. Our case study only shows the five most important requirements. In the next section we will reflect on the case study and derive a proposed decision support framework for the design of a model execution approach.

6.6 Case Study Reflection

In Section 6.5 we observed an SPO during the design of an MDEE. We have shown how design decisions from the architecture determine the priorities of the quality characteristics. The existing architecture of the MDEE and the design decisions that are present,

together form the context of the model execution approach. It shows that, just as with any component in a larger system, the design of an execution approach does not stand on its own, but needs to be embedded in the overall architecture. Some design decisions might constrain the execution approach, other design decisions might even mitigate the problems that an execution approach gives. As an example we look at build-time time behavior, a requirement that was described in the previous section. From Table 6.3 we learn that the interpretive approach is preferred when a specific build-time time behavior is required. However, when the MDEE is built using a programming language and platform that uses interpretation, such as JavaScript, the decrease in build times with a generative approach might be mitigated. An interpreted language does not need a separate compile step that needs to be executed by the generator, and that reduces the build time. This shows that the design decisions that are already present influence the execution approach.

We have distilled three areas from the decisions described in Section 6.5 that steered the design of the model execution approach. The decisions described in Section 6.5, and summarized in Table 6.4 are used to illustrate the areas.

6.6.1 The Metamodel

The metamodel and its features and requirements have an influence on the most fitting model execution approach. This is illustrated by decision **D1: OEM** and requirement **R6: Customize the model**.

A model with a high-level of abstraction (such as **D1: OEM**) will require a more complex model execution, because the distance in terms of abstraction between a programming language and the model is larger. With an interpretive approach, the application will require more resources to perform this model execution. This influences the run-time behavior of the model execution approach, and thus the application itself.

On the other hand, requirement **R6: Customize the model** increases the priority of the build-time time behavior characteristic. This leads to a preference for run-time interpretation, because that approach is preferred if build-time time behavior is important.

6.6.2 The Architecture

The chosen architecture for the application forms a second area of influence on the most fitting model execution approach. A multi-tenant, distributed application (as defined by **D3: Multi-tenancy** and **D5: Distributed application**) can result in conflicting requirements for the most fitting model execution approach.

On the one hand, multi-tenancy prefers interpretation, because it allows the sharing of a single application instance for multiple tenants (see characteristic co-existence in Section 6.4). This maximizes the resource sharing, and enables fast unloading and loading of changes, which decreases the build times. On the other hand, a distributed application might not benefit from interpretation, because every process has to do the interpretation. Figure 6.2 shows that the interpretation process is part of the application, and is thus duplicated when the application is separated in multiple components and processes. This adds resource utilization to the platform.

The decision for a distributed application (**D5: Distributed application**), makes it possible to design a hybrid model execution approach. A distributed application consists of different (distributed) components that can use their own execution approach, as shown in Section 6.5 where only the messages were re-designed.

6.6.3 The Platform

Although Kelly & Tolvanen [136] make no distinction between the architecture, framework, the operating system, or the runtime environment, we see a different influence from the operating system or runtime environment. As decision **D4: .NET platform** illustrates, the lack of support for dynamic software updating requires a different model execution approach to satisfy the requested build-time time behavior. This matches the approaches of Meijler et al. [163] with their customized Java class loader and Czarnecki & Eisenecker [49] using the extension object pattern.

The SaaS delivery model (**D2: SaaS delivery model**) removes most of the problems around *installability* and *co-existence*: the platform is controlled by the SPO.

6.6.4 The Decision Support Framework

From the observations we see three distinct areas that influence the model execution approach. The metamodel and its features and requirements lead to decisions that influence the execution approach. The architecture and the platform can both constrain the execution approach as well as mitigate challenges. Determining the priorities for the quality characteristics can be a difficult task.

The design of the best fitting model execution approach for an MDEE is not different from other parts of the MDEE; it is not possible without knowledge of the context. The description of the design process that we gave in Section 6.5 is generic for the software development life cycle. We propose, based on the observations made during the case study, a tailored version of the process for the design of a model execution approach (shown in Figure 6.3). It shows that the current architecture is input for the prioritization of the quality characteristics. The priorities can then be used to assess the possible execution approaches. How the priorities are determined is not prescribed by the framework, however, we have shown two possible methods to determine them: an informal method and the AHP method.

The framework offers guidance for SPOs in the design of their model execution approach. By formalizing their architecture in a set of design decisions, and by prioritizing the quality characteristics, SPOs can calculate the preference for either the generative or the interpretive approach. This can then in turn be used to design a fitting hybrid model execution approach.

6.7 Discussion

The validity of our research is threatened by several factors. The internal validity of our study is threatened because the correlation between quality characteristics on the one hand and the execution approach on the other hand are not straightforward. The claims in the reviewed literature, however, do show a convergence towards each other.

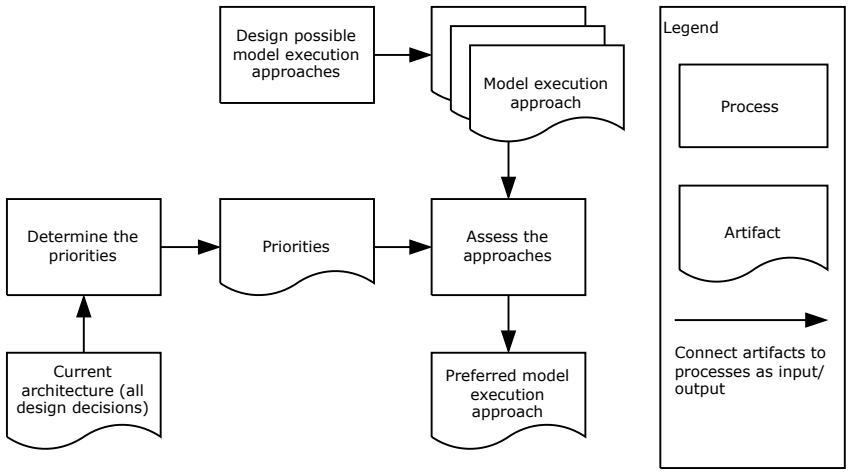

Figure 6.3: The process of selecting a best fitting model execution approach. The process starts with the design of possible execution approaches. The current architecture is the input for the prioritization of quality characteristics. The priorities can be used in assessing possible execution approaches.

Some characteristics lack a significant number of references, making them volatile. However, we regard the claims that are made not as controversial, but in line with existing research. The data that we found in literature consists of anecdotal argumentation, based on the experience of the authors. The claims that were made, were not validated and not supported with empirical evidence. To create a more trustworthy decision support framework, the data presented in Table 6.3 should be validated by empirical research. Experiments or large case studies should provide more quantitative data on the fulfillment of the different quality characteristics.

The construct validity of our case study is threatened by the fact that one of the authors is involved in the object of the study, resulting in a possible bias in our observations. However, the observations were made during a period of several months in which the model execution was actively designed. Our observations were reviewed and commented on by other team members involved. The descriptions of the observations, and the described requirements and decisions were correctly described according to these comments.

The external validity of our research is threatened because our case study is done at a single company. The observations, however, were done over an extensive period of time, and the results were discussed with the team. We argue that the conclusions and observations are in line with existing literature. The decision support framework, however, should be further strengthened by additional case studies.

6.8 Conclusion

We present two contributions to the research on MDD, and in particular to the development of MDEEs. The survey in Section 6.3 illustrates that there is a lack of guidance and knowledge for SPOs. Although the SPOs show that indeed many forms of model execution approaches are used, they do not have an explicit rationale for their design.

In Section 6.4 we studied and summarized existing literature to correlate quality characteristics with the model execution approach. Although this knowledge was already available, it was scattered over many papers. Our study makes the experience and knowledge of many authors available to MDD researchers and practitioners. We summarized the results in Table 6.3, which can be used as a reference in the design of a fitting model execution approach. In Section 6.5 we demonstrate how these results can be used as input for the decision-making in selecting alternatives.

The second contribution that we present is the decision support framework as presented in Section 6.6. With this framework, SPOs have a structured process for the design of the model execution approach. By making these design decisions explicit, and by adding the results from Table 6.3 as input to the decision-making process, SPOs can design the best fitting execution approach. The influence of the context of the MDEE as shown in Section 6.6, and the interplay between existing design decisions and the model execution approaches is made explicit and can lead to better designs.

Although we are not able to relieve SPOs from the hard work of designing a model-driven engineering environment, we argue that our research brings them closer to the best fitting design. By making existing knowledge and experience accessible, the solutions in the decision-making process can be assessed with more confidence. In Section 6.3 we show that many SPOs already use a hybrid form of model execution, but do not have a strong rationale. However, our research also uncovers the need for more empirical research to support SPOs in the design and development of MDEEs. Table 6.3 is primarily based on anecdotes, and often not backed by real evidence. Experiments and case studies should be conducted to strengthen the evidence used in our decision support framework. The framework itself is created by observing a single SPO designing a model execution approach, and it should be evaluated by applying it to other SPOs.

Many questions in the design of software can be answered with “it depends", leaving the questioner puzzled as to what he should do. We present how the context of the MDEE influences the design of a model execution approach for MDEEs. Existing design decisions determine the priorities of quality characteristics, which in term steer the design of the model execution approach. We also show how SPOs can utilize the knowledge presented in this paper to allow them to steer their design process towards the most fitting model execution approach.

Acknowledgements

The authors like to thank Jurgen Vinju, Tijs van der Storm, and their colleagues for their feedback and knowledge early on in the writing process. Finally we thank the team at AFAS Software for their opinions, feedback, and reviews.

7

Proposing a Framework for Impact Analysis for LDCPs

Low-code development platforms accelerate software development by facilitating end-user programming. Through higher-level abstractions citizen developers are enabled to develop increasingly complex software systems. While this improves productivity and efficiency it also introduces new challenges in the development process.
The evolution of the low-code development platform and the applications built on top of it is one of those challenges. Understanding the impact of changes on the software system is crucial for both the maintenance as well as the improvement of running software. Citizen developers can be supported by direct feedback that reflects how their changes impact the system. Professional developers can use impact analysis to correctly migrate existing data. Finally, the operations engineers that are responsible for the availability of the platform and the applications can plan seamless upgrades of new versions. Impact analysis should be at the foundation of the development of low-code development platforms.
This paper proposes the Impact Analysis for Low-Code Development Platforms *framework, a conceptual framework that supports the discussion, research, and implementation of impact analysis. The proposed framework describes the different subsystems and artifacts in a low-code development platform, the different types of professionals involved, and how these professionals can use impact analysis to support their engineering decisions. Through a descriptive case study we discuss the role of impact analysis in an industry low-code development platform. Through the feedback acquired by impact analysis, professionals can stay in control of the evolution of both the applications as well as the low-code development platform itself.*

This work was originally published in *MODELS 21: Proceedings of the 24thACM/IEEE International Conference on Model Driven Engineering Languagesand Systems: Companion Proceedings (2nd LowCode Workshop)*, titled 'Proposing a Framework for Impact Analysis for Low-Code Development Platforms'. It was co-authored by Slinger Jansen.

7.1 Introduction

A trend that is still growing and gaining traction is low-code development platforms (LCDPs) [28]. These LCDPs facilitate end-user programming for *citizen developers*, people without formal programming education that develop software, through Model-driven Development (MDD). Ultimately, the goal of these platforms is to enable citizen developers to build full-stack software applications [217].

While this may have started with relatively simple applications that automated one task, the applications targeted by LCDPs are becoming increasingly complex. From enterprise services [281], Internet of Things [189] to the enablers of digital transformations in the manufacturing industry [219]. This growth and evolution of LCDPs into supporting more different kinds of systems and more complex systems also give rise to new challenges. Challenges that were once the domain of software engineers and operations engineers[1] are now becoming challenges for these citizen developers.

Software evolution is one of these challenges. While it starts with the design, development, and release of an application built with an LCDP, the citizen developer will quickly become aware of the challenge of software evolution. As these applications grow and become more complex, companies will depend more and more on them. Quality characteristics of these applications become more important, and the impact of changes needs to be predicted before they are made, or released into production. However, software evolution for the professional developer and operations engineer also becomes more challenging. A large part of the applications is developed by a new type of professional, the citizen developer. The professional developers and operations engineers are not always aware of the changes made by the citizen developers as they might be part of a different team, or even a different company.

Impact analysis provides all three types of professionals with the needed feedback. The analysis of how changes impact other parts of the system and the running applications supports the professionals in making engineering decisions. We propose a framework to support the discussion and research of impact analysis in low-code development platforms. First we describe the three types of professionals (citizen and professional developers and operations) involved, the subsystems of an LCDP, and the involved artifacts. The *Impact Analysis for Low-Code Development Platforms* itself comprises of the execution of change analysis, the collecting of change analysis results, and the deduction of impact observations that are presented to the involved professionals. Through a case study we show how this framework can be applied, and how impact analysis can result in feedback for those professionals involved in the development of LCDPs and applications. We report on a decade of development of an industry LCDP and application, with 18 months of operational usage. Different forms of impact analysis are used to facilitate control over the evolution of the system and support engineering decisions made by the involved professionals.

The research approach is explained in Section 7.2. In Section 7.3 we discuss LCDPs in general and propose the *Impact Analysis for Low-Code Development Platforms* frame-

[1] Different titles are used for these roles, such as 'DevOps engineers', 'Platform Engineers', and 'Site Reliability Engineers'. We use the term *operations engineers* to refer to the people and/or teams that are responsible for technical management and maintenance of software systems running in production.

work. The case study is described in Section 7.4 and analyzed in Section 7.5. Section 7.6 discusses the case study, the research, and future work. Related work is discussed in Section 7.7. Our conclusions are stated in Section 7.8.

7.2 Research Approach

In this research the role of impact analysis in a low-code development platform (LCDP) is discussed. A descriptive case study is conducted at AFAS Software, during the development of an industry LCDP. The LCDP is used to develop a new ERP system. Currently the platform itself is used only internally, while the resulting ERP system is offered as a Software-as-a-Service (SaaS) product. The research and development of the LCDP *AFAS Focus* started in 2010, but a separate development team was not created until 2013. From 2013 until 2019 the development team grew from 10 people to 50 people, including citizen developers, testers, and professional developers. At the end of 2019, the first customers went live in a private beta program, and in the second half of 2021 the product was publicly launched.

The first author has been part of the research and development team at AFAS Software since 2011, first as Software Architect, but from 2013 as a Lead Software Architect. During the development of the LCDP, the first author was jointly responsible for the development of the LCDP architecture, the data conversion techniques, and the upgrade strategy. The second author has been involved in the project as an independent external researcher since 2015. Our research is based on observations and contributions made by the first author during the development of AFAS Focus. The challenges, the proposed framework, and the results are discussed with the second author since the start of the involvement of the second author. These discussions have bent the inward look from AFAS Focus to LCDPs in general. The research and development of AFAS Focus has resulted in contributions such as a comparison of model execution [182], a framework for data migration [186], and a maturity model for API management that is applied to LCDPs [183]. The *Impact Analysis for Low-Code Development Platforms* framework presented in Section 7.3 is based on the acquired knowledge, research contributions, and experience accumulated throughout these years.

7.3 Impact Analysis for Low-Code Development Platforms

LCDPs accelerate the development of applications by decreasing the amount of hand-coding required [28]. This is accomplished by making software development easier by raising the abstraction level through model-driven development. The higher abstraction label also makes it possible to develop software for people without a formal software development background: the citizen developer. These citizen developers are trained professionals with domain knowledge that are enabled to develop solutions for their domain-specific problems.

The *Impact Analysis for Low-Code Development Platforms* framework is depicted in Figure 7.1. It comprises of the execution of change analysis, the collecting of change analysis results, and the deduction of impact observations that are presented to the involved professionals. We distinguish three types of professionals involved in the

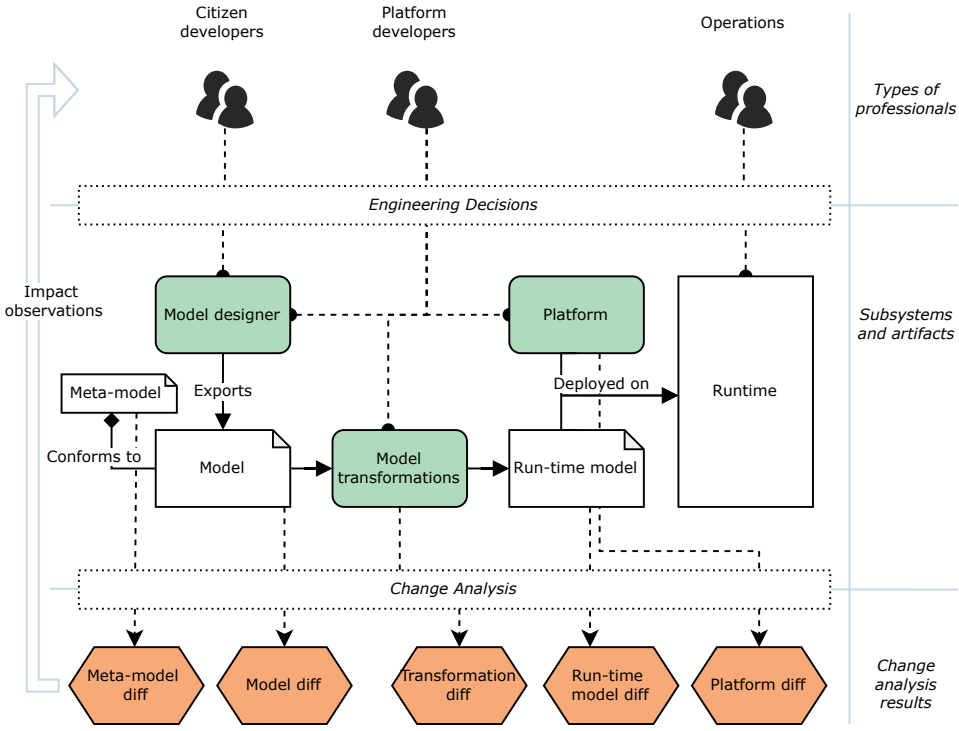

Figure 7.1: The *Impact Analysis for Low-Code Development Platforms* framework comprises of the execution of change analysis, the collecting of change analysis results, and the deduction of impact observations. Changes occur in different parts of the system, but are collected through the change analysis. The resulting changes are represented in diffs. These diffs can be analyzed for impact, resulting in impact observations. The results of impact analysis are used to inform the professionals. They can use this feedback to improve the platform, to redesign existing solutions, or decide to revert certain changes. Furthermore, team coordinators can use the impact framework to orchestrate the process of LCDP evolution.

impact analysis. First of all, the citizen developers, who uses the LCDP to develop applications. Secondly, the LCDP developers, responsible for the development of the LCDP itself. Finally, the operations engineers are responsible for keeping the applications and LCDP available and responsive. The general structure of an LCDP consists of three subsystems and three types of artifacts. These subsystems and artifacts are analyzed for changes, resulting in *change analysis results*, in the form of diffs. These diffs result in impact observations that inform the professionals and support their engineering decisions. The evolution of the LCDP and the applications can be controlled through the created feedback. The remainder of this section explains the subsystems and artifacts, and the impact analysis process in more detail.

The first subsystem is the *model designer*. The designer is the Integrated Development Environment (IDE) offered to the citizen developer. It provides an interface for the development of the model(s) and includes help, and feedback. The first type of

artifact is the *meta-model*. The meta-model describes the model elements and thus the capabilities of the model. An LCDP can utilize multiple models, and thus multiple meta-models, targeting different aspects of an application, such as the data, the business logic, and the user interface. The designed models are the second type of artifact. These are parsed, processed, and transformed by the *model transformations* subsystem. This system transforms the model into a *run-time model*. The run-time model can take different forms, depending on the LCDP implementation. It could either be an intermediate model specific to the LCDP, or a general purpose model (or programming language). The *platform* subsystem contains all the features and infrastructure necessary to execute the run-time model. It contains service frameworks, data access libraries, and other functionality present in the resulting application that is not dependent on the model. The platform and run-time model are deployed to the *runtime* that executes the two subsystems.

Changes are collected by analyzing the subsystems and artifacts in an LCDP. This is done before these subsystems and/or artifacts are deployed on the runtime. The results are expressed in *change analysis results*, representing the changes made, and are generally expressed in a *diff*. From these results the impact that changes have on the system can be derived. The impact observations support the professionals in making engineering decisions on the evolution of the LCDP and the applications. These observations can block the release of new versions, or adjust the future roadmap.

The specifics of the change analysis, the representations chosen for the change results, and the impact observations are LCDP specific. Therefore the remainder of this section will give examples to illustrate the process expressed in the *Impact Analysis for Low-Code Development Platforms* framework. These examples are given by revisiting the three types of professionals that are responsible for the development and operations of the applications developed on the LCDP and the LCDP itself.

First, the point of view of citizen developers who use the LCDP to create solutions is taken. In commercially offered LCDPs these developers are the customer, or work for the customer, of the LCDP. They use the *model designer* as the primary way of interacting with the LCDP. It allows them to create an application by expressing their solution constrained by the meta-model offered by the LCDP. To facilitate rapid application development, technical details will be hidden from them. Citizen developers can, based on impact observations, receive feedback on the quality of the model(s) that they have developed. An example is given: *A change to the model leads to a far bigger change in the run-time model, because a specific model element represents a complex piece of runtime functionality. There is, however, a different solution possible that solves the problem and leads to a smaller change in the run-time model. This solution has preferable characteristics: a smaller impact on the running application. This impact observation is generated by analyzing the model diff and the run-time model diff, and linking the changes made to the model to those in the run-time model.* In general the change results can be analyzed by specific rules, maybe a recommender system [3, 144] could be used, and suggest alternative solutions. Such a system requires the knowledge to link model changes to run-time model changes and to characterize these changes depending on their impact on the runtime by incorporating Software Operational Knowledge.

Secondly, we take the viewpoint of the LCDP developers, who develop and maintain

the three subsystems in an LCDP. The LCDP developers can use the feedback to optimize the platform for both the citizen developers and the operations team. Model changes can be analyzed to identify features that are either neglected or popular. These observations can then be incorporated in the roadmap to optimize the meta-model and the designer. Meta-model changes can be analyzed to find the places in the designer impacted by these changes. The run-time model and platform changes serve as the source for the runtime impact analyses. This analysis points to parts of the system that could harm the operational characteristics of the application. These changed parts can then be reverted before releasing the new version.

Finally, the operations engineers', who are responsible for keeping the applications and LCDP available and responsive, viewpoint is taken. If the LCDP is an internal platform, such as in our case study, there will be a single group of operations engineers. However, if the LCDP is a commercially offered solution there will probably be two groups of operations engineers. The operations engineers that support the platform itself are responsible for the availability of the LCDP: the model designer, the model transformations, and the runtime. The operations engineers that are part of the customer company will focus on the availability of the applications, using the features offered by the LCDP to do so. This team supports the release of new versions, while monitoring the running environments. These operations engineers benefit from the impact observations in planning the upgrades of the platform and/or new applications. The run-time model and platform changes can tell them if they require more runtime resources.

The proposed framework describes the process of impact analysis for LCDPs in generic terms. LCDP providers and consumers should instantiate this framework for their own specific case. However, the *taxonomy for software change impact analysis* [153] can help. The taxonomy lists eight criteria to classify impact analysis approaches.

- *The scope of the analysis*: does the impact analysis operate on code, models, or other artifacts. The framework focuses on static analysis of code and models, and does not incorporate dynamic aspects collected from a running system.
- *The granularity of the analysis*: what level of detail is analyzed and reported. Collected changes can be aggregated, or only collected on a certain level. This determines the impact observations that can be made.
- *The utilized technique*: examples of techniques are call graphs, execution traces, and message dependency graphs. The best fitting technique depends on the kind of LCDP and its architecture.
- *The style of the analysis:* is the analysis global, search based, or exploratory.
- *Tool support*: which tools support the chosen approach.
- *Supported languages*: which programming or modelling languages are supported by an approach. While the model in an LCDP is custom developed, providers can benefit from standard tooling for language and model engineering. The run-time model could also be a standard programming language or intermediate model that is supported by available tools.
- *Scalability*: how scalable is the impact analysis approach.

- *Experimental results*: is the approach tested and shown to be successful.

These criteria list the variability present in the *Impact Analysis for Low-Code Development Platforms* framework. LCDP providers and consumers need to choose an existing approach or design their own approach for impact analysis when applying the framework.

7.4 Case Study

We conduct a case study on how impact analysis is applied in an industry LCDP at AFAS Software. This case study describes the process in which impact analysis is applied and from which we derived the *Impact Analysis for Low-Code Development Platforms* framework. AFAS Software is a Dutch vendor of ERP software based in Leusden, The Netherlands (with additional offices in Belgium, Curaçao, and Aruba). The privately held company currently employs over 500 people and generated 191 million of revenue in last year (2020). AFAS' main software product is called *Profit*, which is an ERP system consisting of different modules such as Taxes, Finance, HRM, Order Management, Payroll, and CRM. This product has over 2 million users across 11.000 organizations of all sizes, ranging from companies with a single employee to companies with thousands of employees.

After 25 years of development, AFAS launched a new version of its ERP system, which is called *SB+*. This new system is based on an internal developed LCDP, called *AFAS Focus*, using an ontological enterprise model [228] (the platform was formerly called NEXT). The system is cloud-based and its architecture applies event sourcing and CQRS [187] to satisfy quality characteristics such as availability and responsibility. The research and development of AFAS Focus started ten years ago. Approximately 60 companies currently use *SB+* for their day-to-day accounting. Figure 7.2 shows the instantiated *Impact Analysis for Low-Code Development Platforms* framework for AFAS Focus.

7.4.1 Involved Professionals

The AFAS Focus LCDP is developed and utilized by AFAS only, the three types of professionals described in Section 7.3 are all employees of AFAS. The citizen developers are a team of professionals who formerly served in roles such as business analyst, software tester, or support engineer. They have multiple years of experience in the domain of ERP software, but have no formal training in software development. Through internal training and knowledge sharing sessions they are trained in the usage of the LCDP. The LCDP developers consist of four teams that are responsible for the development of the model designer, model transformations, and platform. Finally, a team of operations engineers is responsible for maintaining the runtime and deploying upgrades of AFAS Focus. AFAS Focus uses a four-weekly release schedule: every four weeks a new release is developed, tested, and deployed. During the four weeks that a release is in production, smaller releases (called *hotfixes*) are deployed to solve blocking issues. In the first 18 months (January 2020 to June 2021) that AFAS Focus was used by companies for their day-to-day accounting 212 releases (18 regular releases and 194 hotfixes) were deployed.

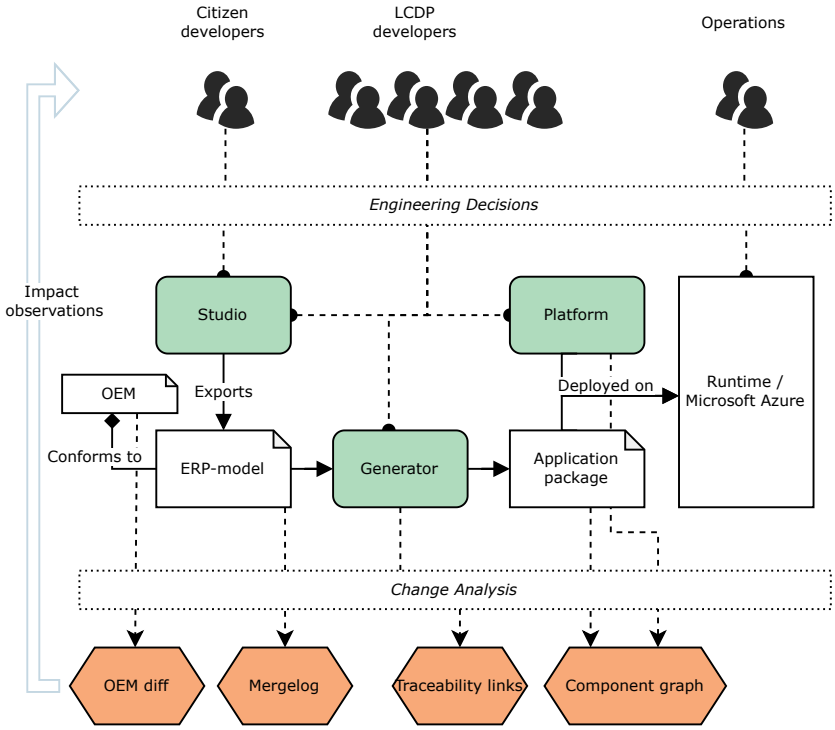

Figure 7.2: The *Impact Analysis for Low-Code Development Platforms* framework as instantiated for AFAS Focus.

7.4.2 Subsystems and Artifacts

AFAS Focus is developed using C# on the .NET platform and TypeScript. As mentioned, an ontological enterprise model [228] is used to develop the ERP system on top of this platform. The LCDP developers are responsible for the model designer, called *Studio* and the meta-model, called *OEM*. Required features are designed and developed in collaboration with the citizen developers, who are the 'customers' of these components. The model is created through the combination of a graphical designer and a text-based designer. Through a suite of model transformations (the *Generator*), this model is transformed into a run-time model called the *application package*. To optimize the model execution approach, AFAS Focus adopted a custom run-time model that is interpreted [182]. The run-time model expresses the component types present in an event sourced system, such as *aggregate roots*, *events*, and *projectors*. Together with a *host package*, containing the platform, the application package is deployed on the runtime which is running in *Microsoft Azure*. The host package contains both the interpreters for the run-time model, as well as features that are not dependent on the model and are not developed by the citizen developers. Examples of these features are generic import functionality and user management. These features are developed using 'traditional' software development methods and contained in the host package.

7.4.3 Change Analysis Results

From the start, impact analysis was applied in the development of AFAS Focus to facilitate the co-evolution between meta-model and model, and model and runtime. We discuss the different applied impact analysis and the context within AFAS Focus.

Meta-model Diff

While currently only one single model is developed (the model representing the new ERP system), many more models exist for testing and exploration purposes. To maintain compatibility between *Studio* and the model, the *OEM* is versioned and every model contains the version to which it conforms. When a new version of the *OEM* is introduced, a manual evolution step is developed to facilitate the upgrade of existing models, based on the *OEM diff*. *Studio* itself is evolved manually. Whenever a model is loaded that conforms to an older *OEM* version, the developed evolution steps are executed to automatically upgrade the model. These evolution steps are developed in such a way that only minimal changes are made to a model to make it conform to the new *OEM*. These evolution steps are developed and tested by the LCDP developers whenever they make a meta-model change. Downgrading a model to an older *OEM* is not supported and facilitated. The citizen developers are briefed and educated on the new *OEM* elements, but are not bothered by details of the evolution steps.

Model Diff

Co-evolution of the system and the customer data is one of the biggest challenges faced in the development of AFAS Focus. For an accounting system it is crucial that customer data remains accessible and available after an upgrade. Changes that originate from the model were a big unknown in that challenge, because it was outside of the direct influence of the LCDP developers.

The co-evolution of the model and customer data is solved through a combination of *manual specification* and *operator-based co-evolution* [211], called the *Mergelog*. The most frequent evolution steps in the model are mapped and formally supported in *Studio*. Citizen developers can select one of the operators to perform co-evolution with the customer data. The manual specification option serves as a fall-back for non-supported operations. Together these make sure that the model and the customer data co-evolves.

The model itself is versioned in a general purpose versioning system (*git*) using a text-based representation. To not obscure the versioning history, the evolution steps automatically executed in *Studio* caused by *OEM*-evolution are confined to a minimum. The LCDP developers review the automated evolution steps in *Studio* to make sure that the model history is not polluted by *Studio*.

Transformation Diff

AFAS Focus uses a general purpose programming language (C#) for the model transformations. Initially the transformations were designed in a single multi-phase transformation system. This monolithic transformation system, together with the fact that a general purpose programming language is used, made it hard to perform change impact analysis.

Therefore, the single transformation system was redesigned into a component-based

transformation system. This not only improves the development process for multiple development teams, it also makes it easier to analyse the transformation system. Currently AFAS implements basic traceability [88] in the transformation system that links elements in the run-time model to the specific transformation component. These traceability links are used in the engineering process by the LCDP developers to find the specific transformation components and model elements that lead to a run-time model element.

Run-time Model Diff

The run-time model of AFAS Focus consists of a small number of component types that exist in an event sourced architecture [187]. To analyse and observe the impact, the run-time model is represented in a message dependency graph [195]. This component graph contains the different micro-services and their event-based communication. A graph of the current release of AFAS Focus consists of around 25.000 nodes and 35.000 edges. These nodes represent the components in an event sourced system: 5.000 nodes are components containing logic, 15.000 nodes represent messages, and the remaining nodes are data objects. The edges represent the usage patterns between these components. The graph can be explored in a visual representation. The size of the AFAS Focus component graph is useless to visualize in one image, however, by enabling developers to explore the graph many useful observations can be made.

To analyse the impact of changes a diff between two of these graphs is created. This diff reflects the changes made to different component types. An example summary of such a diff is shown in Table 7.1. The numbers in Table 7.1 show a 0.1% total size increase. It shows per component type the number of added, removed, and changed elements, together with the totals of the nodes and edges. Note that the edges can only be added or removed, not changed, because they do not contain further information. This summary information acts as a starting point to browse the diff information and drill-down to the lowest possible level: a diff of a specific model element (as shown in Figure 7.3). These diffs can be used to spot specific changes that need the attention of the development team. Example changes that require manual verification are changes that require custom data evolution steps (such as a property that has become mandatory, or a property type that is changed), or changes that could result in data loss (such as an event type that has been removed by error).

Using the component graph the LCDP developers have found bugs, such as

- Components receiving messages with no source component.
- Messages that are received and sent, but for which no contract description exists.

An example of a less obvious observation is the identification of a part of the system that has a high level of complexity (measured in terms of many different messages and many different components involved), but that offers little functionality in return. Parts with those characteristics are discussed in architecture meetings and optimizations are planned accordingly.

Platform Diff

The platform part of AFAS Focus contains the interpreters for the run-time model elements as well as features that are not modelled by the citizen developers and do not depend on the model. Similar to the transformation subsystem the platform too

Component Type	Original	Added	Removed	Changed	New
Command	3679	34	7	51	3706
Event	11005	76	84	570	10997
QueryModelObject	4797	79	91	409	4785
StreamItem	1395	12	6	273	1401
StreamItemRouter	1379	16	103	142	1292
QueryProjector	578	11	6	49	583
StreamProjector	565	0	4	230	561
Nodes	23398	228	301	1724	23325
Edges	35898	474	506		35866

Table 7.1: Example diff summary between version 1.19 and 1.20 from the combined run-time model and platform impact analysis. The changes per component type as well as aggregated totals are shown.

is implemented in C# . However, unlike the transformation system the platform does not have the same challenges for doing change impact analysis. Features that are developed in the platform subsystem use the same component and message types. Therefore, the platform system can be represented through reverse engineering in the same component graph as the run-time model.

Run-time Diff

The run-time model and platform diff are combined into a single run-time diff. This diff represents the whole impact of the new release on the runtime.

The run-time diff is used by the LCDP developers to analyse the impact of a new release and plan the upgrade procedure accordingly. The data upgrade steps are verified using the diff, making sure that all necessary upgrades are specified. Depending on the impact of the release a separate upgrade strategy is used.

- **move** - When a release only contains business logic or user interface changes, a straightforward move upgrade can be performed. During this upgrade a new application process is launched that connects with the same data storages. When this new process is verified to run correctly all incoming connections are transferred to the new process. This is the fastest upgrade process.
- **minor** - When a release only contains business logic, user interface, or volatile storage changes, a minor upgrade can be performed. During this upgrade a new application process is launched, the data storages are copied, and the data schema changes are executed on the copy. When the new application process is verified to run correctly all incoming connections are transferred to the new process.
- **major** - Whenever the event messages have changed, a major upgrade is required. During this upgrade the changed events are rewritten and saved into a copy of the data storage. When the new application process is verified to run correctly, all incoming connections are transferred to the new process.

Depending on the complexity of the changes when a **major** upgrade is required, the operation engineers can decide to allocate more resources for the system during the

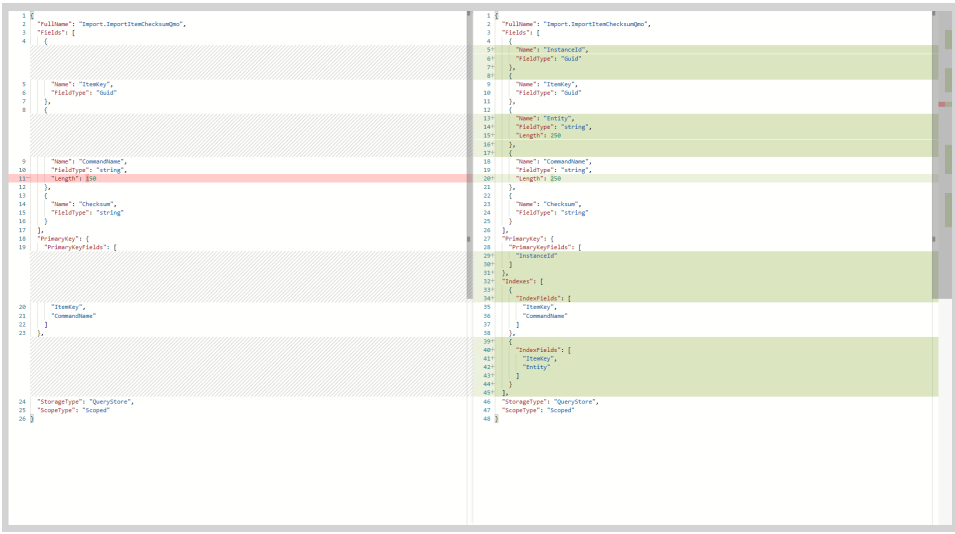

Figure 7.3: The diff of a single data storage element from the generated component graph diff. These elements are represented in JSON, the diff shows a plain text-based diff. The diff shows two added properties, a changed property length, and two new indexes. The changed property length is especially important: a decrease in length requires a data evolution step to make sure existing data conforms to the new schema. This feedback could be presented in *Studio* to warn the citizen developer.

upgrade. Specific upgrade challenges are identified and reported to the operations team.

One metric that can be used to measure the size of the ERP system *SB+* is the total number of components and relations between the components. Table 7.2 shows an overview of the numbers of the last nine versions. These numbers give a sense of the magnitude of the ERP system, however, similar to source code lines the number of components does not relate to productivity. The decrease of 8% between version 1.15 and 1.16 for instance can be attributed to an optimization in the transformation system. These numbers can be used by the LCDP developers and operation engineers to plan the available resources on the runtime platform: more data storage objects require more data resources, while a larger number of components with logic require more memory resources.

The LCDP developers also analyse the run-time diff to identify required optimizations in platform and model transformations. Together with the citizen developers the diff is analyzed to identify optimizations in the model by applying different meta-model elements, or by introducing new meta-model elements.

Taxonomy criteria

Following the eight criteria of the taxonomy for software change impact analysis the approach can be described as follows:

- *The scope*: analysis is done on code and models.

Version	Nodes		Edges	
	Total	% change	Total	% change
1.11	18104		26202	
1.12	18499	+2%	26935	+2%
1.13	19431	+5%	28205	+4%
1.14	20638	+6%	29955	+6%
1.15	21814	+5%	32034	+6%
1.16	20132	-8%	29711	-8%
1.17	23819	+18%	36428	+22%
1.19	23398	-2%	35898	-2%
1.20	23325	-1%	35866	-1%

Table 7.2: The last nine releases of AFAS SB+ with total numbers of nodes and edges of the component graph, enriched with the percentage of change with respect to the previous version. Due to problems with the release of version 1.17, version 1.18 was never released, therefore it is absent in this table.

- *The granularity*: this differs for the different analyses, the *OEM diff* and the required evolution operations are done on all levels, the *mergelog* is recorded on the level of attributes, the *traceability links* are recorded on the level of generator components, and the *component graph* is created on the level of the architectural components.
- *Utilized techniques*: analysis is done by traceability links, message dependency graphs, and model diffs.
- *Style of analysis*: the impact analysis in AFAS Focus is done globally.
- *Tool support*: no generic or open source tools are used.
- *Supported languages*: the implemented approaches are specific to the meta-model and architecture of AFAS Focus.
- *Scalability*: scalability is no real concern, because there is a single model and the analysis is executed on demand.
- *Experimental results*: these are discussed in this research.

7.5 Analysis

The previous section described how impact analysis is embedded in the software development process of AFAS Focus. However, we also observe possible improvements.

The **meta-model diff** is used to manually create model evolution steps. This creation could be automated by analyzing the meta-model diff. The meta-model diff could also be used to automate the evolution of the model designers. These two improvements would make the process of meta-model evolution more efficient.

The chosen solution for co-evolution of model and customer data, based on the **model diff**, serves its intended purpose, but also has a number of drawbacks. First of all, the chosen approach remains laborious and complicated. Citizen developers are required to explicitly specify evolution operators. However, they are only aware

and capable of specifying their own evolution. Changes of several developers are combined into a single release, but how the different evolution operators influence and even conflict with each other is not obvious and cannot be specified. Second, the operator-based co-evolution is only able to express model evolution. Changes in the meta-model, the model transformations, or in the platform cannot be expressed through these operators. Each of those cases needs to be expressed through manual specification, making the process error-prone and laborious. Third, the co-evolution does not support the teams in improving the solutions by providing feedback. It does not facilitate the evaluation of the chosen model changes, which might not be optimal.

While the **transformation system** already generates traceability between the transformation components and the run-time model, it misses a link between the model element and run-time model element. Such a link would facilitate the translation of run-time errors into model errors for the citizen developers.

The run-time diff, the combination of **platform diff** and **run-time model diff**, greatly improves the impact analysis of AFAS Focus. However, this artifact is also not yet utilized to its full potential. It could replace the model diff and serve as a basis for the data evolution steps. As the run-time diff represents the full evolution step it would allow for a more complete and more automated generation of the required data evolution steps. From the 209 releases that AFAS Focus had, 37 required a *major* upgrade. Most of these major upgrades required a data evolution step that could not be automatically generated with the current operation based co-evolution. Example manual specifications are property type transformation, specific calculations for introduced mandatory properties, and renames of event types. Another improvement would be a better suited representation of the generated component graph for the citizen developers to analyse. The traceability between run-time model and model elements could support such a representation. Applying change patterns [265] could be another solution to improve the representation by summarizing smaller changes in higher-level change patterns. An important missing feature is safety guards against public APIs. Earlier research [183] showed the importance of API management for LCDPs. Safety checks on these published APIs support the citizen developer in these tasks. Finally, the diff should be used to generate the required data evolution in a semi-automated way. Certain semantics of the model evolution will be lost through the indirection of the impact analysis, because it is done on the run-time model. This could either be mitigated by also analyzing the model impact, or by manual specification.

7.6 Discussion

LCDPs enable citizen developers to develop increasingly complex software without formal software development training. While this improves productivity and efficiency it also introduces new challenges in the development process. The evolution of the LCDP and the applications built on top of it is one of those challenges. Impact analysis can play an important role in the mitigation of this challenge.

As we have observed and experienced in the development of AFAS Focus, impact analysis supports the professionals in the planning and orchestration of software evolution. The presented *Impact Analysis for Low-Code Development Platforms* framework

offers a conceptual structure to reason about impact analysis for LCDPs. At AFAS Software the framework proved its use in the design of the different subsystems and artifacts, and the implementation of impact analysis. The development process benefits from the different analysis results, even with the identified improvements.

Current research in model-driven development and low-code development platforms offer a lot of the lower-level techniques and approaches to perform impact analysis. However, an overall framework to structure the plan-do-act process of engineering teams is missing. Our research is a proposal for such a framework but requires more grounding and evaluation. While we believe that it can be generalized to other contexts, this should be proven by further research.

For future work we plan to execute a systematic literature review to ground the framework in existing research on model-driven development, low-code development platforms, and change impact analysis. The review should result in a comprehensive overview and concept definitions that would bring together the different research areas.

Second, the framework itself will be validated with other LCDP providers. Case studies at other LCDP providers are necessary to evaluate the framework and correct it from any biases. To prevent the framework from following the biases of a single provider, multiple providers should share and combine their knowledge. We plan to conduct multiple case studies in the near future.

The results of the literature review and the multiple case studies will be used to add more detail to the framework: specific guidance and useful techniques that can be applied. After these improvements the framework can be evaluated on completeness and usefulness through expert interviews.

7.7 Related Work

Co-evolution in model-driven development platforms is a well researched topic. An overview of the different approaches is given by Rose et al. [211], while Lämmel [148] describes and discusses coupled transformations. An approach for creating model diffs is presented by Toulmé [249]. Gruschko, Kolovos & Paige [100], Cicchetti et al. [36], and Di Ruscio, Lämmel & Pierantonio [58] describe approaches for (semi-)automated co-evolution of meta-model and models. Gruschko, Kolovos & Paige [100] categorizes the meta-model changes in *non-breaking changes*, *breaking and resolvable changes*, and *breaking and unresolvable changes*. By using high-order transformation rules, the second category can be used to automatically adapt models to new meta-model versions. The research of Wachsmuth [262] is similar and also focuses on the automated adaption of models to meta-models. A dynamically adapting system is proposed by Ferreira, Correia & Welicki [80]. Impact analysis to support the incremental execution of model transformations is another application proposed by Hearnden, Lawley & Raymond [104]. A megamodeling approach is presented by Iovino, Pierantonio & Malavolta [117], which supports the capturing of change impact in a model. By doing this, change impact becomes a model itself, which allows the application of model-driven tools to the challenge of change impact.

The problem of representation of change impact is researched in the area of *change*

patterns. Change patterns express changes to a process on a higher level of abstraction, making them easier to comprehend. This notion is introduced by Rinderle-Ma, Reichert & Weber [208] and Weber, Rinderle & Reichert [264] for Process-Aware Information Systems (PAIS). The authors applied these change patterns to assess the level of change support in different PAIS. The patterns form a language that allows an easy and comprehensible comparison of the different systems.

Distributed event-based systems pose another challenge in impact analysis. Components in event-based systems are intrinsically loose coupled, which makes them hard to evolve and analyze. Tragatschnig, Stevanetic & Zdun [250] use the notion of change patterns to analyse event-based systems. Their research shows that change patterns are an efficient language to capture the evolution of an event-based system. Popescu et al. [195] propose a static analysis that analyses distributed event-based systems. Analysis of traditional software systems depends on the explicit invocation to create a dependency graph. Their proposed method analyses the message-oriented middleware that these systems are based on and creates a dependency graph from those results.

7.8 Conclusion

Low-code development platforms (LCDPs) accelerate the development of software through new abstractions that remove many of the technical details. However, challenges such as software evolution remain. Citizen developers, LCDP developers, and operations engineers need tools and processes to solve these challenges. Together these professionals, regardless if they are from the same company or not, are responsible for the success of the application of the LCDP. Evolution plays an important role in that success and this requires that these professionals collect feedback that informs and supports their engineering decisions. We believe that impact analysis helps and supports these teams in reaching their goals.

An overall framework to structure the implementation of impact analysis for LCDPs is missing. In Section 7.3 we propose the *Impact Analysis for Low-Code Development Platforms* framework that conceptualizes the process of impact analysis for LCDPs. It describes the different subsystems and artifacts, together with the process of impact analysis. Using the taxonomy of Lehnert [153] we discussed the variability in the framework and how providers can implement this.

Through a case study of an industry LCDP we explore the framework in more depth. Although case studies at other LCDP providers are necessary to evaluate the framework and correct it of any biases, we believe that impact analysis within LCDPs can improve both the applications developed on top of the LCDP as well as the platform itself. Impact analysis should be at the foundations of the LCDP development process.

Part V

Conclusion

8

Conclusion

We begin this chapter by answering our research questions, starting with the individual sub-questions and then following with the main research question. Furthermore, in this chapter we reflect on the contributions of the dissertation, and discuss future work.

8.1 Answers to the Research Questions

8.1.1 Event Sourced Systems and Evolution

Chapter 1 introduces event sourcing as a data modelling approach and architecture pattern that emerged from the Domain-Driven Design (DDD) community [275]. The subtitle of the seminal book on DDD by Evans [74], "Domain-Driven Design", reads "Tackling Complexity in the Heart of Software". The DDD methodology and community attempt to tackle the complexity of software systems by fostering domain understanding through collaboration. Event sourcing is a pattern that, according to the methodology, matches this approach by focusing on the events that happen within business processes.

Four sub-questions directed our research on event sourced systems, focusing on the evolution of event sourced systems while also discussing the pattern in general. While we can confirm that this pattern can be applied to fight complexity in large software systems, we also recognize that it brings its own set of challenges. The challenge of evolution in event sourced systems is discussed in detail, the other challenges are left for future work.

SRQ1.1 *- What types of systems apply event sourcing and why?*

The overview of the 19 systems presented in Chapter 3, detailed in Tables 3.2 and 3.4, shows that event sourcing can be applied in systems of any size: both smaller and larger systems benefit from the pattern. Our primary interest lies in the large and complex systems with millions (or even billions) of events per month, but our interviews support the claim that all of these systems, both large and small, have benefited from event sourcing. As one engineer stated, underlining this conclusion, *"I have never seen an event sourced system that was rewritten to a system with traditional current state storage."* The event sourcing pattern is not tied to a specific type of functional domain, but is applied in many different domains.

Our study identified four reasons for event sourcing: audit, flexibility, complexity, and trending. The rationale audit traces back to the common characteristic of event sourcing: the immutability of the events. However, we determined that not every system applies strict immutability, but that there are actually three levels of immutability that engineers apply, namely *strict*, *cut-off moments*, and *mutable*. Flexibility is seen in the fact that the events can be used both as a means of communication between different modules and as a source for different types of data models. Event sourcing is especially suited for software systems with complex temporal logic, but also supports the implementation of distributed systems. A final reason identified during the interviews is that event sourcing is trending and new, and that engineers like to try out new techniques.

***SRQ1.2** - How should event sourced systems be defined?*

We extracted a definition of event sourced systems (ESSs) from the interviews conducted for the research in Chapter 3. We gave a complete overview of ESSs based on these definitions, combined with our (at that time) five years of experience. We repeat the key definitions of *events* and the *event store*.

Event. *An event is a discrete data object specified in domain terms that represents a state change in an ESS.*

Event Store. *An event store is a set of event streams. These streams form the partitions of the event store and are disjoint.*

Furthermore we define the $project$ function, that consumes stored events to create *projections*. These projections are used to validate new information and to retrieve stored information.

Project function. *The project function takes one or more event streams and creates a projection with the data from the given events. The projection itself can take different forms; for instance, it can be a relational database that is updated through SQL statements, or a search index manipulated through the filesystem.*

Based on the different perspectives discussed by the engineers we added nuance and variation options to the different presented concepts. Section 3.5 presents the definitions in detail.

***SRQ1.3** - How should event sourced data structures be evolved?*

In Chapter 2 we presented the event store upgrade framework. This framework starts with a common language of different event store upgrade operations that can be used to analyze the required upgrade. Based on the operations a matching upgrade technique and strategy should be selected, while finally a matching strategy should be selected. The framework makes the trade-offs explicit, supporting both a utilization as decision support as well as an automated selection that can be implemented. The framework summarizes best practices from literature and industry, and was validated by three experts.

In Chapter 3 we discussed the different event schema evolution techniques with the interviewed engineers. They all had experience with multiple techniques that are often combined in a single system. This affirms the event store upgrade framework that correlates the techniques to the different operations. The framework is further elaborated on with the benefits and the liabilities of the different techniques. We

discussed five event schema evolution techniques in Section 3.7: *versioned events*, *weak schema*, *upcasters*, *in-place transformation*, and *copy-transform*. The technique *lazy transformation* presented in the framework was not mentioned by the engineers, while *weak schema* was mentioned but not present in the framework.

The majority of the interviewed engineers had experience with multiple techniques, often combining them in a single system. As all techniques have their benefits and their liabilities, we did not find a single technique that would be applicable in all scenarios. However, through the framework presented in Chapter 2 and the advice formulated in Chapter 3 we provide the tools to help engineers in choosing the most suitable techniques.

> ***SRQ1.4*** *- What are the challenges faced by software engineers in applying event sourcing?*

We discussed five challenges experienced by the interviewed engineers in Section 3.6. The steep learning curve was addressed by our description of event sourced systems through the presented definitions and operations. These can be used in discussing and teaching of ESSs. Evolution of event sourced systems is discussed in detail in both Chapter 2 and Section 3.7. The other three challenges, lack of technology, rebuilding projections, and privacy, are presented as a start for a research roadmap. We call for researchers to explore these challenges further.

> ***MRQ1*** *- What are the challenges software architects face in the evolution of event sourced systems and how can they be mitigated?*

We have shown that the challenges that software architects face in evolving event sourced systems are caused by, among other things, the assumed immutability of event stores, the implicit schema embedded in the system, and the performance of data migrations. To support architects with the challenge of immutability, Chapter 3 distinguishes three levels of immutability in event sourced systems. These three levels determine the possible event schema evolution techniques that software architects can apply. The central answer to this question, however, is the available event schema evolution techniques that we present, including benefits and liabilities. This overview enables software architects to decide which would best meet their requirements based on their context. Chapter 2 presents the techniques in the form of a framework, further supporting the architects in their choices.

8.1.2 API Management in Software Ecosystems

Both software systems in a software ecosystem as well as modules in a single large system have the need to communicate with each other. Communication happens through Application Programming Interfaces (APIs), which can be realized by different technologies. In the microservice architecture style, APIs are preferred over HTTP protocols. To maximize the value of their APIs, Software Producing Organizations (SPOs) need to evaluate and improve their API management capabilities as needed.

> ***SRQ2.1*** *- How should SPOs that expose their APIs to third parties evaluate their API management practices?*

In Chapter 4 we present the *API management focus area maturity model* (API-m-

FAMM), a new framework that captures the topics and processes API management consists of. The API-m-FAMM is a focus area maturity model that enables SPOs to assess their current API management practices. After establishing their current maturity, the model offers a path for improvement in various areas. The model improves the transparency and availability of API management assessment frameworks and tools by constructing, evaluating and validating a publicly available framework scientifically grounded and validated in industry. The API-m-FAMM was successfully deployed in practice with minimal involvement of the researchers using the constructed *do-it-yourself* kit.

> ***SRQ2.2*** *- How mature are the API management capabilities that LCPs offer to their customers?*

In Chapter 5 we use the API-m-FAMM to evaluate four Low-Code Platforms (LCPs). We conclude that these LCPs support around half of the practices described in the API-m-FAMM but leave the other practices to be implemented by the customers of the LCPs. From the four LCPs, only *Mendix* places API management firmly on its roadmap. The other platforms defer much of the work to either third-party vendors or the LCP customers.

We suspect that LCP providers will soon be challenged in providing capabilities that enable citizen developers to transform their applications into platforms. However, our research shows that LCP providers are currently unable to support such capabilities for citizen developers and require technical staff to implement such architectures and mechanisms through either third-party solutions or custom solutions built on top of the LCP. We concluded that as LCPs are becoming more powerful, they can use the API-m-FAMM to evaluate and update their roadmaps. Finally, we identified five engineering challenges that, if solved, will create a next generation of citizen developers who can independently create complete software platforms and software ecosystems, and subsequently manage them without the requirement for highly specialized technical knowledge.

> ***MRQ2*** *- What kind of support for API management practices is offered by LCPs, and how should they evaluate and improve that support?*

The four LCPs that we evaluated in our research only implemented half the API management practices listed in the API-m-FAMM. The other half is left to the customers to be implemented. The API-m-FAMM proved to be a useful tool in the assessment of API management maturity and enables the LCP providers to plan their improvements.

We believe that by offering more API management capabilities LCP providers will further democratize software development. Using the API-m-FAMM LCP providers can plan the required improvements to enable the development of software ecosystems. This creates a more powerful platform that supports the development of business critical software platforms.

8.1.3 Evolution Supporting Architecture

Architectural design decisions [122] influence many operational characteristics of a software system, such as the manageability of a system when it undergoes evolution.

Designing an architecture that supports upgradability and manageability is crucial for SPOs.

> ***SRQ3.1*** *- How should SPOs make an informed decision between a generative or interpretive model execution approach?*

What the best fitting design is for a certain software system always depends on the context. In Chapter 6 we presented how the context of the Model-Driven Engineering Environments (MDEE) (a synonym for LCPs) influences the design of a model execution approach. This was done by summarizing existing literature and correlating quality characteristics with the different model execution approach.

Through a survey among 22 product experts of sixteen different SPOs we illustrated that there is a lack of guidance and knowledge for SPOs in building an MDEE. Although the survey shows that many forms of model execution approaches are used, SPOs do not have an explicit rationale for their design. Although the presented knowledge was already available, it was scattered over different sources such as textbooks and scientific papers. Our study makes the experience and knowledge of these many authors available to MDD researchers and practitioners. We also presented this in the decision support framework in Section 6.6. With this framework, SPOs have a structured process for the design of the model execution approach.

> ***SRQ3.2*** *- What is the role of change impact analysis in an LCP?*

LCPs aim to make software development more efficient by raising the level of abstraction at which the software is developed. The challenge of software evolution, changing the software as a response to outside forces, within LCPs remains. Citizen developers, LCP developers, and operations engineers need tools and processes to solve these challenges. We believe that change impact analysis helps and supports these teams in reaching their goals. An overall framework to structure the implementation of impact analysis for LCPs is missing. In Chapter 7 we propose the *Impact Analysis for Low-Code Development Platforms* framework that conceptualizes the process of impact analysis for LCPs.

Through a case study of an industry LCP we explored the framework in more depth. Although case studies at other LCP providers are necessary to validate the framework and correct it from any biases, we believe that change impact analysis within LCPs can improve both the applications developed on top of the LCP as well as the platform itself. Change impact analysis should be at the foundation of the LCP development process.

> ***MRQ3*** *- How should the architecture of an LCP support the evolution of both the platform as well as the applications?*

The architecture of an LCP co-determines the maintainability of the platform and the applications under evolution. In the first sub-question we showed how the model execution approach relates to the quality characteristics of the platform. Interpretation might negatively effect the performance, while it could positively effect the upgradability of an application. Chapter 7 shows how different architectural decisions influence how well impact analysis in an LCP can be executed.

We cannot give cut-and-paste answers or prescribe a specific architecture, however,

our research shows how architectural decisions influence the maintainability of an LCP. Engineers should take this advice and design their architecture consciously. However, we also agree that there are a number of remaining open questions that require future research.

8.2 Reflections

8.2.1 Reflections on Research Approach

Throughout our research we used different research methods, as detailed in Chapter 1. These methods were used to gather knowledge, create models and frameworks, and evaluate them. We reflect on three of these methods because of their importance to our research.

In Chapter 3 we applied constructivist Grounded Theory (GT). This method allowed us to conduct exploratory research. Instead of forcing our own ideas and theories, we started the interviews open-minded and let the engineers guide our exploration. Our perspectives, and our experience, did shape the data that we conducted. However, constructivist GT assumes that neither data nor theories are discovered, but are constructed by the researchers out of the interactions with the field and its participants. Data are co-constructed by researchers and participants, and shaped by the researchers' perspectives, and values.

For our API management model API-m-FAMM, as described in Chapter 4, we created a Focus Area Maturity Model (FAMM). We followed the method described by Steenbergen et al. [235, 236], but incorporated Design Science Research methods such as the card sorting technique, described by Nielsen [175], to perform maturity level assignments, conducted multiple evaluation cycles, and made use of criteria for artifact evaluation introduced by Prat, Comyn-Wattiau & Akoka [197]. We provided a detailed description of the construction of the FAMM through the published source data [159] that can be used by researchers as an example in future works. The API-m-FAMM was successfully deployed in practice with minimal involvement of the researchers using the constructed *do-it-yourself* kit. This shows that we as researchers can make maturity models more relevant for industry by investing in the usability of these assessment and improvement tools.

Finally, the Systematic Literature Review (SLR) method was central in Chapters 4 and 6. By conducting these reviews we were able to synthesize existing research and create new models. SLRs are an important tool to present existing research in a condensed manner. Although this made it accessible for a new audience, our contributions did not stop there. In both cases we constructed an actionable model from the synthesized knowledge. The decision support framework from Chapter 6 not only presents the relation between quality characteristics and model execution approach, it also shows how software architects can rate desired characteristics and receive decision support. The API-m-FAMM not only enables the assessment of API management maturity but also shows the capabilities that can be improved. The conducted case studies supported the usefulness of these tools for industry.

We show the value of a diverse toolkit for research in software architecture and its

creation process. We hope that these studies are picked up in the software architecture curriculum.

8.2.2 Reflections on Academic Impact

In Chapter 1 we stated four research challenges related to the innovations that lie at the heart of this dissertation. In this section we discuss those challenges and the impact of our research on them.

> **RC1**: Software systems are offered as a service, making SPOs responsible for the operation of the software. Techniques for reliable systems that continue operating through the deployment of new versions need to be designed and evaluated.

Through our research we have shown that event sourcing and CQRS are identified by industry as patterns for performant, reliable, and scalable software systems. Although event sourcing and CQRS are already receiving more and more attention in research literature [70, 278] scientific knowledge on the pattern itself is sparse. Our research adds to the body of knowledge based on interviews with multiple engineers. We show in which contexts and systems the pattern works and add rationales and challenges that are experienced in industry. We also show upgrade strategies that explicitly support software architects in the deployment of new versions. New research can built upon this new knowledge.

> **RC2**: LCPs are increasingly used to build business-critical systems and companies depend on the stability and reliability of the platform. Changes made to the platform and applications threaten these characteristics; the evolution of the platform and applications need to be controlled to mitigate these risks.

We propose the *Impact Analysis for Low-Code Development Platforms* framework with the goal of synthesizing existing literature on change impact analysis within the context of LCPs. This proposal should guide new research in the building of scientific knowledge that can improve the current support for software evolution in LCPs.

We have also shown how the model execution approach in a platform applying model-driven engineering influences the quality characteristics of a system. Designing the best fitting model execution approach can improve among other things the maintainability of the system. Our decision support framework supports engineers in this design.

> **RC3**: Event Sourcing and CQRS are identified by industry as techniques for performant, reliable, and scalable software systems. However, the evolution of event sourced systems require new techniques and strategies that software architects can employ.

We discuss the benefits and liabilities of different event schema evolution techniques. Our research can be used to mitigate the challenges that software evolution poses on event sourced systems. The framework presented in Chapter 2 supports engineers in the design of their upgrade strategy.

> **RC4**: More and more SPOs turn their software products into platforms and, at the same time, allow external complementors to access their plat-

forms. To effectively grow their system into an ecosystem, they need to be supported in the management of their integration capabilities.

The API-m-FAMM as presented in Chapter 4 enables SPOs to evaluate their current API management capabilities. Based on the evaluations of four LCPs we have shown what the current state of API management in LCPs is. We argued that LCPs need to improve their supported API management practices to enable their customers to create software ecosystems, but also stated specific research engineering challenges that LCPs face on this road to improvement. The API-m-FAMM not only enables SPOs to evaluate their current practices, it also offers SPOs a clear roadmap for improvement of their capabilities.

8.2.3 Reflections on Industry Impact

This dissertation was conducted in the context of the AMUSE project, which is a joint research project of Utrecht University, the Vrije Universiteit Amsterdam, and AFAS Software B.V. Given that I already worked on the in-house LCP developed by AFAS Software, the industry context of the problems and challenges were clear. Although the previous sections answered the stated research questions and reflected on our contributions to the scientific methods we applied, our work had an impact on industry as well.

The biggest impact was made with our research on event sourcing. There was a lack of knowledge on evolution of event sourced systems that was filled by the research presented in Chapters 2 and 3. Our research work has been shared on numerous occasions since it was published (see the listing on page 212 for a complete overview).

Our work on *The Dark Side of Event Sourcing*, Chapter 2, was presented during two local meetups and one conference in Berlin, Germany. We presented different forms of *An Empirical Characterization of Event Sourced Systems and Their Schema Evolution*, Chapter 3, in two meetups and three conferences (in Austria, The Netherlands, and virtually). The experience we gathered through developing the *Focus* platform was shared in meetups, conferences, and a podcast. Finally, we also contributed a chapter to a book on Domain-Driven Design: *Tackling Complexity in ERP Software: a Love Song to Bounded Contexts*, in *Domain-Driven Design, The First 15 Years*.

Presenting research at an industry conference differs from presenting at an academic conference. Instead of the research methodology and a sound scientific approach, the audience is far more interested in results and applicability. This translation of science is crucial if presented to an industry audience. However, scientific research has great value to offer to industry, because it transcends the anecdotal that is sometimes offered by industry speakers. We believe it is crucial that software engineering research is translated into applicable knowledge that furthers the profession.

8.2.4 Personal Reflections

This dissertation took six years to complete, from start to finish. At the start of my PhD trajectory I had eight years of experience as Software Engineer and Architect. During my PhD I combined the research and writing with my role as Lead Software Architect at AFAS Software B.V. The result was a combination of practical software development experience and conducting scientific research. This uniquely formed my research in

the manner that Kruchten et al. [146] describes; it gave context to the problems that I worked on. For me this was an ideal situation, working in industry but also dipping my toes into the academic world. Although research without a substantive real-world context is definitely valuable, it is not as appealing to me. My favorite quote, and the motto of this dissertation, has always been "Talk is cheap, show me the code." (attributed to Linus Torvalds). This is not to say that research is cheap. If I have learned something in the past six years, it is that conducting research and writing scientific papers is hard work. However, I have always been fond of research that also shows its practical application in a real-world problem. The real proof of research, for me, is showing its value in the implementation of a software system.

A consequence of combining my work as a software architect with the research as a PhD, was the lack of specialization. A PhD often specializes in a single topic and becomes an expert in that field. However, the combination of both my interest in a broad range of topics and the broad responsibility as a software architect resulted in a dissertation that lacks a specialization. Instead I decided to address three aspects and illustrate how the contributions all support the umbrella idea of supporting software architects in the evolution of low-code platforms.

This combination of industry and scientific research was not always a bed of roses. Translating our, in our own eyes, interesting results into scientific contributions proved to be difficult at times. The challenge of translating industry experience into scientific research is similar to that of translating scientific research into industry. While the responses we got during meetups and conferences proved that our results were relevant to industry, our reviewers were critical of the scientific contribution. Through this process of submitting articles one learns to handle criticism. However, after six years I have to conclude that every paper became better after the feedback. Although the review system seems unpredictable at times, peer review is a crucial tool in the creation and communication of scientific knowledge. I know that the feedback I received improved the quality of my articles and dissertation.

From the process of writing articles I did learn that I like, and perhaps even need, a system that allows for iterative work. If, after a mere six years of experience, I could offer suggestions for the scientific system it would be this: support, better yet, encourage, early ideas and results. There are already places where these early ideas and results can be presented; the yearly *Belgium-Netherlands Software Evolution Workshop* is one that comes to mind. But it seems that there is room for improvement as well. Fast and multiple review cycles should enable a more iterative approach that would allow researchers to grow and develop their ideas. These ideas can then be supported by prototypes or other proof of concepts. Newer forms such as hackathons and competitions (such as the *MSR Mining Challenge* or *Language Workbench Challenge*) already stimulate these approaches. There is still an important place for more theoretical work in the form of articles, but other forms could stimulate a "show, not tell" culture.

8.3 Future Work

Our research discusses software evolution in low-code platforms. As we already stated, our research stands on the shoulders of giants. At the same time we acknowledge that

we are not finished yet. There is room for improvement in all three of the following areas: future research should be conducted.

8.3.1 Event Sourced Systems and Evolution

Our studies on event sourced systems started to uncover challenges that require attention. We contributed a framework with techniques, including benefits and liabilities, to the knowledge on evolution of event sourced systems. Follow-up studies on the frequency of schema changes in event sourced systems and the required operations would improve our framework. Similar to work on relational databases such as that published by Roddick [209], Maule, Emmerich & Rosenblum [160], and Curino, Moon & Zaniolo [47] would further the scientific knowledge of schema evolution in event sourced systems. Further study is also necessary on upgrades of the query-side of an event sourced system. While those upgrades can build on knowledge of schema evolution in relational databases and document databases, the interaction between schema evolution in the event store and derived databases is a new topic that requires additional research.

In Chapter 3 we address other challenges such as the steep learning curve, lack of technology, rebuilding projections, and privacy. These challenges require follow-up studies to provide knowledge, techniques, and tools to address these challenges. The lack of technology will not be solved by new research but tools to evaluate event source technology would improve the selection process. The privacy aspect within event sourced systems is an interesting challenge. Recent legislation around privacy protects the personal information of users by ensuring that they can be forgotten by a system. However, an immutable event store conflicts with these directives. New research should evaluate possible solutions that make event sourced systems compliant with the law.

8.3.2 API Management in Software Ecosystems

During the development of the API-m-FAMM we listed several improvements that could be made. In this list of future work we join Spruit & Röling [233] and Sanchez-Puchol & Pastor-Collado [218] in their suggestions for future work. First of all, more research is required on how to successfully deploy focus area maturity models in industry. For instance, by developing a web application through which practitioners may easily navigate, as well as read focus area, capability, and practice descriptions, and then mark which practices are or are not implemented within their organization. This would replace our Excel spreadsheet and source document that we distributed through our do-it-yourself kit and make the API-m-FAMM more actionable. The second opportunity lies in the potential to customize and adapt the API-m-FAMM depending on certain organizational characteristics and goals. The conducted case studies have shown that certain focus areas are irrelevant for organizations that exclusively utilize internal APIs. Other information that could be used to perform this adaptation may include characteristics such as the size of the organization, whether a third-party API management platform is used, or what type of product or services the organization provides. Moreover, we hypothesize that there are opportunities for automation, customization, and adaptation. This would create incentives for practitioners to reuse maturity mod-

els and FAMMs for a longer period of time. Finally, it should be investigated whether significant differences exist in terms of API management maturity between organizations that do use commercial platforms and those that do not.

In our research on API management among LCPs we outlined five engineering research challenges that could be the topic of future work. These challenges are discussed in Chapter 5 and are in the areas *Life Cycle Management*, *Performance*, *Observability*, *Community*, and the abstractions for citizen developers. While the term 'citizen developer' indicates that it has been the goal to open up software engineering to people without formal software engineering education, we can hardly claim that this has been successfully accomplished for API management. The complexity of modern software solutions and the inherent simplification required to create LCPs are constantly in direct conflict with each other. The platformisation trend is an example of this conflict: whenever LCPs venture into more complex types of software, new models and abstractions are required to truly make software engineering accessible to any citizen developer. We see it as future work to design new abstractions that make LCP solutions simpler and more powerful in supporting API management practices.

8.3.3 Evolution Supporting Architecture

The final chapters on architecture in LCPs show that in general, knowledge on developing fully capable platforms is missing. Future research should gather and synthesize this knowledge, first of all through systematic literature reviews. This existing knowledge can then be used to ground the proposed *Impact Analysis for Low-Code Development Platforms* framework. The result should be a comprehensive overview, including definitions for the different concepts, that would bring together the different research areas around impact analysis, model-driven engineering, and software evolution. We join Tisi et al. [248] in stating that scalable artifact management is required for LCPs to support large-scale platforms. These management tools would further improve the options for impact analysis over these artifacts.

Luo et al. [156] state the high learning curve, and the lack of ease of use in LCPs as challenges reported by practitioners. This matches our evaluation of support for API management practices in LCPs. Tasks just outside the obvious design and development of business applications, such as API management and testing [137], require new abstractions.

The results from these opportunities should be validated with current LCP providers. This validation balances the theoretical knowledge with real-world, context-specific experiences. Case studies at multiple LCP providers are necessary to evaluate the collected knowledge and remove biases. This would result in actionable best practices that will move LCPs forward in democratizing software development.

Bibliography

[1] Aalst, W. M. van der, Hee, K. M. van, Werf, J. M. van der & Verdonk, M. Auditing 2.0: Using process mining to support tomorrow's auditor. *Computer*, vol. 43, no. 3. 2010, pp. 90–93. DOI: 10.1109/MC.2010.61 (Cited on page 53).

[2] Adolph, S., Hall, W. & Kruchten, P. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, vol. 16, no. 4. 2011, pp. 487–513. DOI: 10.1007/s10664-010-9152-6 (Cited on pages 15, 45, 46).

[3] Almonte, L., Guerra, E., Cantador, I. & De Lara, J. Recommender Systems in Model-Driven Engineering A Systematic Mapping Review. *Software and Systems Modeling*. 2021. DOI: 10.1007/s10270-021-00905-x (Cited on page 159).

[4] Ampatzoglou, A., Bibi, S., Avgeriou, P., Verbeek, M. & Chatzigeorgiou, A. Identifying, Categorizing and Mitigating Threats to Validity in Software Engineering Secondary Studies. *Information and Software Technology*, vol. 106. 2019, pp. 201–230 (Cited on page 109).

[5] Andreessen, M. Why software is eating the world. *Wall Street Journal*, vol. 20, no. 2011. 2011, p. C2 (Cited on page 3).

[6] Andreo, S. & Bosch, J. API Management Challenges in Ecosystems. In: *International Conference on Software Business*. 2019, pp. 86–93. DOI: 10.1007/978-3-030-33742-1_8 (Cited on pages 10, 116).

[7] Anh, D. T. T., Zhang, M., Ooi, B. C. & Chen, G. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Transactions on Knowledge and Data Engineering*, vol. 4347, no. c. 2018, pp. 1–20. DOI: 10.1109/TKDE.2017.2781227 (Cited on page 51).

[8] Arsanjani, A. & Holley, K. The Service Integration Maturity Model: Achieving Flexibility in the Transformation to SOA. In: *2006 IEEE International Conference on Services Computing (SCC'06)*. 2006, p. 515 (Cited on page 88).

[9] Avery, P. & Reta, R. Scaling Event Sourcing for Netflix Downloads. 2017. URL: https://www.infoq.com/presentations/netflix-scale-event-sourcing (Cited on pages 44, 57).

[10] AxonIQ. AxonDB. 2019. URL: https://axoniq.io/product-overview/axondb (Cited on pages 65, 68).

[11] AxonIQ. Reference Guide Axon Framework reference guide - Event Upcasting. 2019. URL: https://docs.axoniq.io/reference-guide/configuring-

infrastructure-components/event-processing/event-bus-and-event-store (Cited on pages 31, 70).

[12] Basole, R. C. On the Evolution of Service Ecosystems: A Study of the Emerging API Economy. In: *Handbook of Service Science, Volume II*. Springer, 2019, pp. 479–495 (Cited on page 82).

[13] Batouta, Z. I., Dehbi, R., Talea, M. & Hajoui, O. Multi-criteria Analysis and Advanced Comparative Study Between Automatic Generation Approaches in Software Engineering. *Journal of Theoretical and Applied Information Technology*, vol. 81, no. 3. 2015, pp. 609–620 (Cited on pages 137, 143, 145, 146).

[14] Becker, J., Knackstedt, R. & Pöppelbuß, J. Developing Maturity Models for IT Management. *Business & Information Systems Engineering*, vol. 1, no. 3. 2009, pp. 213–222 (Cited on pages 13, 86).

[15] Betts, D., Dominguez, J., Melnik, G., Simonazzi, F. & Subramanian, M. Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure. Microsoft patterns & practices, 2013 (Cited on pages 31, 68).

[16] Bider, I., Johannesson, P. & Perjons, E. *Using Empirical Knowledge and Studies in the Frame of Design Science Research*. Tech. rep. 2013, pp. 463–470 (Cited on page 15).

[17] Biemans, F. P., Lankhorst, M. M., Teeuw, W. B. & Van De Watering, R. G. Dealing with the complexity of business systems architecting. *Systems Engineering*, vol. 4, no. 2. 2001, pp. 118–133. DOI: 10.1002/sys.1010 (Cited on page 53).

[18] Bock, A. C. & Frank, U. In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms. In: *Proceedings of the 24th ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings*. 2021 (Cited on page 8).

[19] Bock, A. C. & Frank, U. Low-Code Platform. *Business & Information Systems Engineering*, vol. 63, no. 6. Dec. 2021, pp. 733–740. DOI: 10.1007/s12599-021-00726-8. (Cited on page 3).

[20] Brady, E. C. & Hammond, K. Scrapping your inefficient engine. *ACM SIGPLAN Notices*, vol. 45, no. 9. 2010, p. 297. DOI: 10.1145/1932681.1863587 (Cited on page 143).

[21] Brandolini, A. Introducing Event Storming. Leanpub, 2018 (Cited on page 44).

[22] Breaux, T. & Moritz, J. The 2021 software developer shortage is coming: companies must address the difficulty of hiring and retaining high-skilled employees from an increasingly smaller labor supply. *Communications of the ACM*, vol. 64. 7 June 2021, pp. 39–41. DOI: 10.1145/3440753 (Cited on page 3).

[23] Brewer, E. A. Lessons from Giant-Scale Services. *IEEE Internet Computing*, vol. 5, no. 4. 2001, pp. 46–55. DOI: 10.1109/4236.939450. (Cited on pages 33, 34).

[24] Brown, A. W. *An introduction to Model Driven Architecture - Part 1; MDA and Today's Systems*. Tech. rep. IBM DeveloperWorks, RationalEdge, 2004 (Cited on pages 8, 134).

[25] Bruin, T. de, Rosemann, M., Freeze, R. & Kulkarni, U. Understanding the main phases of developing a maturity assessment model. *ACIS 2005 Proceedings -*

16th Australasian Conference on Information Systems. 2005 (Cited on pages 86, 117, 118).

[26] Bui, D. H. Design and Evaluation of a Collaborative Approach for API Lifecycle Management. MA thesis. Technical University of Munich, 2018 (Cited on page 82).

[27] CA Technologies. The API Management Playbook: Understanding Solutions for API Management. 2019. URL: https://docs.broadcom.com/docs/the-api-management-playbook (Cited on page 85).

[28] Cabot, J. Positioning of the low-code movement within the field of model-driven engineering. *Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings*. 2020, pp. 535–537. DOI: 10.1145/3417990.3420210 (Cited on pages 3, 8, 116, 156, 157).

[29] Callaghan, M. Facebook - Online Schema Change for MySQL. 2010. URL: https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/ (Cited on pages 27, 31).

[30] Carmel, E. & Agarwal, R. The Maturation of Offshore Sourcing of Information Technology Work. In: *Information systems outsourcing*. Springer, 2006, pp. 631–650 (Cited on page 88).

[31] Castillo-Montoya, M. Preparing for interview research: The interview protocol refinement framework. *Qualitative Report*, vol. 21, no. 5. 2016, pp. 811–831 (Cited on page 48).

[32] Chandy, K. M. Event Driven Architecture. In: *Encyclopedia of Database Systems*. Ed. by LIU, L. & ÖZSU, M. T. Boston, MA: Springer US, 2009, pp. 1040–1044. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_570 (Cited on page 8).

[33] Charmaz, K. & Bryant, A. Grounded theory. *International Encyclopedia of Education*. 2010, pp. 406–412. DOI: 10.1016/B978-0-08-044894-7.01581-5 (Cited on pages 15, 46, 47, 72).

[34] Chen, L. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, vol. 32, no. 2. 2015, pp. 50–54. DOI: 10.1109/MS.2015.27 (Cited on page 24).

[35] Choudhary, V. Software as a service: Implications for investment in software development. In: *40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 2007, 209a–209a (Cited on page 7).

[36] Cicchetti, A., Ruscio, D. D., Eramo, R., Pierantonio, A., Informatica, D. & Aquila, I. L. Automating Co-evolution in Model-Driven Engineering. In: *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE (pp. 222-231)*. 2008 (Cited on page 169).

[37] Claps, G. G., Berntsson Svensson, R. & Aurum, A. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, vol. 57, no. 1. 2015, pp. 21–31. DOI: 10.1016/j.infsof.2014.07.009. (Cited on page 24).

[38] Cleenewerck, T. Modularizing Language Constructs: A Reflective Approach. PhD thesis. Vrije Universteit Brussel, 2007. (Cited on pages 143, 145).

[39] Clements, P. Coming Attractions in Software Architecture. In: *Proceedings of 5th International Workshop on Parallel and Distributed Real-Time Systems and 3rd Workshop on Object-Oriented Real-Time Systems*. April. IEEE Comput. Soc, 1997, pp. 2–9. ISBN: 0-8186-8096-2. DOI: 10.1109/WPDRTS.1997.637857. (Cited on page 53).

[40] Cleve, A., Gobert, M., Meurice, L., Maes, J. & Weber, J. Understanding database schema evolution: A case study. *Science of Computer Programming*, vol. 97, no. P1. 2015, pp. 113–121. DOI: 10.1016/j.scico.2013.11.025. (Cited on page 27).

[41] Coleman Parkes Research. APIs: Building A Connected Business in the App Economy. 2017. URL: https://docs.broadcom.com/doc/apis-building-a-connected-business-in-the-app-economy (Cited on page 82).

[42] Consel, C. & Marlet, R. Architecturing Software Using A Methodology For Language Development. *Principles Of Declarative Programming*, vol. 1490, no. October. 1998, pp. 170–194. (Cited on pages 143–145).

[43] Cook, W. R., Delaware, B., Finsterbusch, T., Ibrahim, A. & Wiedermann, B. *Model Transformation by Partial Evaluation of Model Interpreters*. Tech. rep. 2008 (Cited on pages 143, 145, 146).

[44] Cordy, J. R. TXLA language for programming language tools and applications. *In Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications*. 2004, pp. 1–27. DOI: 10.1016/j.entcs.2004.11.006 (Cited on pages 143, 145).

[45] Crawford, J. K. Project Management Maturity Model. Auerbach Publications, 2006. DOI: 10.1201/9780849379468 (Cited on page 88).

[46] Curino, C., Moon, H. J., Deutsch, A. & Zaniolo, C. Automating the database schema evolution process. *VLDB J.*, vol. 22, no. 1. 2013, pp. 73–98. DOI: 10.1007/s00778-012-0302-x (Cited on page 27).

[47] Curino, C., Moon, H. J. & Zaniolo, C. Graceful database schema evolution: the PRISM workbench. *PVLDB*, vol. 1, no. 1. 2008, pp. 761–772 (Cited on pages 27, 182).

[48] Cusumano, M. A. Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People. Simon & Schuster Trade, 1998 (Cited on page 7).

[49] Czarnecki, K. & Eisenecker, U. W. Generative programming: Methods, Tools, and Applications. Addison-Wesley Professional, 2000, p. 864 (Cited on pages 136, 143, 151).

[50] Dahan, U. Clarified CQRS. 2009. URL: http://www.udidahan.com/2009/12/0 (Cited on pages 24, 61).

[51] Daigneau, R. Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services. Addison-Wesley, 2011 (Cited on page 68).

[52] Date, C. J. An Introduction to Database Systems. 8th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321197844 (Cited on page 58).

[53] De, B. API Management. Berkeley, CA: Apress, 2017, pp. 15–28. DOI: 10.1007/978-1-4842-1305-6_2 (Cited on pages 9, 82, 85, 89, 107, 109, 119, 125).

[54] De Bruin, T., Rosemann, M., Freeze, R. & Kaulkarni, U. Understanding the Main Phases of Developing A Maturity Assessment Model. In: *Australasian Conference on Information Systems (ACIS):* 2005, pp. 8–19 (Cited on pages 88, 94, 107–111).

[55] Debski, A., Szczepanik, B., Malawski, M., Spahr, S. & Muthig, D. In Search for a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing Case Study. *IEEE Software*, vol. PP, no. 99. 2017, pp. 1–1. DOI: 10.1109/MS.2017.265095722 (Cited on page 44).

[56] Deursen, A. van, Klint, P. & Visser, J. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, vol. 35, no. 6. 2000, pp. 26–36. DOI: 10.1145/352029.352035. (Cited on pages 3, 141).

[57] Devoteam. API Management at Liberty Global Inc. 2016. URL: https://nl.devoteam.com/en/our-case-studies/api-management-liberty-global-inc (Cited on pages 85, 86).

[58] Di Ruscio, D., Lämmel, R. & Pierantonio, A. Automated co-evolution of GMF editor models. In: *International Conference on Software Language Engineering*. June 2010, pp. 143–162. (Cited on page 169).

[59] Díaz, V. G., Valdez, E. R. N., Espada, J. P., Bustelo, b. C. P. G., Lovelle, J. M. C. & Marín, C. E. M. A brief introduction to model-driven engineering. *Tecnura*, vol. 18, no. 40. 2014, pp. 127–142 (Cited on pages 134, 143, 145).

[60] Domínguez, E., Lloret, J., Rubio, A. L. & Zapata, M. A. MeDEA: A database evolution architecture with traceability. *Data & Knowledge Engineering*, vol. 65, no. 3. 2008, pp. 419–441. DOI: 10.1016/j.datak.2007.12.001 (Cited on page 27).

[61] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., Lluch-lafuente, A., Giallorenzo, S., Lafuente, A. L. & Mazzara, M. Microservices : yesterday , today , and tomorrow. *Present and Ulterior Software Engineering*, no. November. 2017, pp. 195–216. DOI: 10.13140/RG.2.1.3257.4961. (Cited on page 55).

[62] Dreyer, W., Dittrich, A. K. & Schmidt, D. Research perspectives for time series management systems. *ACM SIGMOD Record*, vol. 23, no. 1. 1994, pp. 1015. DOI: 10.1145/181550.181553. (Cited on page 50).

[63] Dumitraş, T. *No Downtime for Data Conversions : Rethinking Hot Upgrades*. Tech. rep. Carnegie Mellon University, 2009 (Cited on pages 27, 31).

[64] Dumitraş, T. & Narasimhan, P. Why do upgrades fail and what can we do about It? Toward dependable, online upgrades in enterprise system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5896 LNCS, no. August 1996. 2009, pp. 349–372. DOI: 10.1007/978-3-642-10445-9_18 (Cited on pages 27, 69).

[65] Dumitraş, T., Narasimhan, P. & Tilevich, E. To Upgrade or Not to Upgrade Impact of Online Upgrades across Multiple Administrative Domains. *ACM SIG-*

PLAN NOTICES, vol. 45, no. 10. 2010, pp. 865–876. DOI: 10.1145/1932682.1869530 (Cited on page 34).

[66] Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. Guide to Advanced Empirical Software Engineering. *Guide to Advanced Empirical Software Engineering*. Ed. by Shull, F., Singer, J. & Sjøberg, D. I. K., 2008, pp. 285–311. DOI: 10.1007/978-1-84800-044-5_11. (Cited on page 12).

[67] Elisabeth Hove, S. & Anda, B. *Experiences from Conducting Semi-Structured Interviews in Empirical Software Engineering Research*. Tech. rep. 2005 (Cited on page 14).

[68] Endjin. API Maturity Matrix. 2017. URL: https://endjin.com/blog/2017/07/kickstart-your-api-proposition-with-the-api-maturity-matrix (Cited on page 84).

[69] Engström, E., Storey, M. A., Runeson, P., Höst, M. & Baldassarre, M. T. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, vol. 25, no. 4. July 2020, pp. 2630–2660. DOI: 10.1007/s10664-020-09818-7 (Cited on page 13).

[70] Erb, B. Distributed Computing on Event-Sourced Graphs. PhD thesis. Universität Ulm, 2019 (Cited on pages 50, 179).

[71] Erb, B. & Hauck, F. J. On the Potential of Event Sourcing for Retroactive Actor-based Programming. In: *First Workshop on Programming Models and Languages for Distributed Computing*. Vol. 1. ACM, 2016, 4:1–4:5 (Cited on page 44).

[72] Ertl, M. A. & Gregg, D. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, vol. 5. 2003, pp. 1–25. DOI: 10.1145/248208.237175. (Cited on pages 142, 143).

[73] Etikan, I., Musa, S. A. & Alkassim, R. S. Comparison of Convenience Sampling and Purposive Sampling. *American journal of theoretical and applied statistics*, vol. 5, no. 1. 2016, pp. 1–4 (Cited on pages 14, 90).

[74] Evans, E. Domain-Driven Design. Addison-Wesley Professional, 2003 (Cited on pages 3, 44, 54, 61, 173).

[75] Evans, E. Domain-Driven Design Reference. Eric Evans, 2015. (Cited on page 50).

[76] Event Store, L. Event Store. 2019. URL: https://eventstore.org/ (Cited on pages 58, 65, 68, 70).

[77] Fabry, J., Dinkelaker, T., Noye, J. & Tanter, E. A Taxonomy of Domain-Specific Aspect Languages. *ACM Computing Surveys*, vol. 47, no. 3. 2015, pp. 1–44. DOI: 10.1145/2685028. (Cited on pages 137, 143, 144).

[78] Falessi, D., Cantone, G., Kazman, R. & Kruchten, P. Decision-making techniques for software architecture design. *ACM Computing Surveys*, vol. 43, no. 4. 2011, pp. 1–28. DOI: 10.1145/1978802.1978812. (Cited on page 146).

[79] Feijter, R. de, Overbeek, S., Vliet, R. van, Jagroep, E. & Brinkkemper, S. DevOps competences and maturity for software producing organizations. In: *Enterprise, Business-Process and Information Systems Modeling*. Vol. 318. Springer Verlag, 2018, pp. 244–259. ISBN: 9783319917030. DOI: 10.1007/978-3-319-91704-7_16 (Cited on page 117).

[80] Ferreira, H. S., Correia, F. F. & Welicki, L. Patterns for data and metadata evolution in adaptive object-models. *Proceedings of the 15th Conference on Pattern Languages of Programs PLoP 08*. 2008, p. 1. DOI: 10.1145/1753196.1753203. (Cited on page 169).

[81] Flyvbjerg, B. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, vol. 12, no. 2. 2006, pp. 219–245. DOI: 10.1177/1077800405284363 (Cited on page 70).

[82] Forbrig, P. Use cases, user stories and bizdevops. In: 2018 (Cited on page 3).

[83] Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002 (Cited on page 44).

[84] Fowler, M. Event sourcing. 2005. URL: http://martinfowler.com/eaaDev/EventSourcing.html (Cited on pages 5, 24, 44).

[85] Fowler, M. BlueGreenDeployment. 2010. URL: http://martinfowler.com/bliki/BlueGreenDeployment.html (Cited on page 34).

[86] Fowler, M. Schemaless Data Structures. 2013. URL: http://martinfowler.com/articles/schemaless/ (Cited on pages 26, 60, 66).

[87] Fowler, M. What do you mean by Event-Driven? 2017. URL: https://martinfowler.com/articles/201701-event-driven.html (Cited on pages 4, 44, 47).

[88] Galvão, I. & Goknil, A. Survey of Traceability Approaches in Model-Driven Engineering. In: *11th IEEE International Enterprise Distributed Object Computing Conference*. 2007, pp. 313–313 (Cited on page 164).

[89] Gámez Díaz, A., Fernández Montes, P. & Ruiz Cortés, A. Towards SLA-driven API Gateways. *XI Jornadas De Ciencia E IngenierÍA De Servicios*. 2015 (Cited on page 84).

[90] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, 1995 (Cited on pages 49, 50, 59, 70).

[91] Gaouar, L., Benamar, A. & Bendimerad, F. T. Model Driven Approaches to Cross Platform Mobile Development. *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*. 2015, 19:1–19:5. DOI: 10.1145/2816839.2816882. (Cited on pages 143, 144).

[92] Garousi, V., Petersen, K. & Ozkan, B. Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, vol. 79. Nov. 2016, pp. 106–127. DOI: 10.1016/j.infsof.2016.07.006. (Cited on page 16).

[93] Gartner. A Guidance Framework for Evaluating API Management Solutions. 2019. URL: https://www.gartner.com/en/documents/3956412/a-guidance-framework-for-evaluating-api-management-solut (Cited on page 85).

[94] Golfashani, N & Nahid, G. Understanding reliability and validity in qualitative research. *The qualitative report*, vol. 8, no. 4. 2003, pp. 597–607. (Cited on page 71).

[95] Google Inc. Protocol Buffers. 2019. URL: https://github.com/google/protobuf (Cited on pages 57, 68).

[96] Gorodinski, L. Scaling Event-Sourcing at Jet. 2017. URL: https://medium.com/@eulerfx/scaling-event-sourcing-at-jet-9c873cac33b8 (Cited on page 44).

[97] Gray, J. & Reuter, A. Transaction processing: concepts and techniques. Elsevier, 1992 (Cited on page 50).

[98] Gregg, D. & Ertl, M. A. A Language and Tool for Generating Efficient Virtual Machine Interpreters. In: *Domain-Specific Program Generation*. Springer Berlin Heidelberg, 2004, pp. 196–215 (Cited on pages 143–145).

[99] Greiler, M., Deursen, A. van & Storey, M. A. Test confessions: A study of testing practices for plug-in systems. In: *Proceedings - International Conference on Software Engineering*. 2012, pp. 244–254. ISBN: 9781467310673. DOI: 10.1109/ICSE.2012.6227189 (Cited on page 46).

[100] Gruschko, B., Kolovos, D. S. & Paige, R. F. Towards Synchronizing Models with Evolving Metamodels. In: *Proceedings of the International Workshop on Model-Driven Software Evolution*. 2007, pp. 3–3 (Cited on page 169).

[101] Guana, V. & Stroulia, E. How Do Developers Solve Software-engineering Tasks on Model-based Code Generators? An Empirical Study Design. *First International Workshop on Human Factors in Modeling*, no. May. 2015 (Cited on pages 138, 143, 145).

[102] Haddad, C. Comparison Evaluation Matrix. 2015. URL: https://wso2.com/whitepapers/comparison-evaluation-matrix/ (Cited on page 84).

[103] Harrison, N. B., Avgeriou, P. & Zdun, U. Using patterns to capture architectural decisions. *IEEE Software*, vol. 24, no. 4. 2007, pp. 38–45. DOI: 10.1109/MS.2007.124 (Cited on page 44).

[104] Hearnden, D., Lawley, M. & Raymond, K. Incremental Model Transformation for the Evolution of Model-Driven Systems. In: *9th International Conference, MoDELS 2006*. 2006 (Cited on page 169).

[105] Helland, P. Immutability changes everything. *Communications of the ACM*, vol. 59, no. 1. 2015, pp. 64–70. DOI: 10.1145/2844112 (Cited on page 54).

[106] Hevner, A. & Chatterjee, S. Design Research in Information Systems. Springer, 2010. DOI: 10.1007/978-1-4419-5653-8 (Cited on page 13).

[107] Hevner, A. R., March, S. T., Park, J. & Ram, S. Design Science in Information Systems Research. *MIS quarterly*, vol. 28, no. 1. 2004, pp. 75–105 (Cited on page 13).

[108] Hick, J. M. & Hainaut, J. L. Database application evolution: A transformational approach. *Data and Knowledge Engineering*, vol. 59, no. 3. 2006, pp. 534–558. DOI: 10.1016/j.datak.2005.10.003 (Cited on page 27).

[109] Hinkel, G., Denninger, O., Krach, S. & Groenda, H. Experiences with model-driven engineering in neurorobotics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9764. 2016, pp. 217–228. DOI: 10.1007/978-3-319-42061-5_14 (Cited on page 143).

[110] Hoda, R., Noble, J. & Marshall, S. Developing a grounded theory to explain the practices of self-organizing Agile teams. *Empirical Software Engineering*, vol. 17, no. 6. 2012, pp. 609–639. DOI: 10.1007/s10664-011-9161-0 (Cited on page 46).

[111] Hohpe, G. & Woolf, B. Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional, 2004 (Cited on pages 44, 68).

[112] Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A. & Ducasse, S. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, vol. 26, no. 1. 2018, pp. 161–191. DOI: 10.1007/s11219-016-9344-4 (Cited on page 125).

[113] Humble, J. & Farley, D. Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional, 2010. (Cited on pages 33, 34).

[114] Hüner, K. M., Ofner, M. & Otto, B. Towards A Maturity Model for Corporate Data Quality Management. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. 2009, pp. 231–238 (Cited on page 88).

[115] Hutton, G. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, vol. 9, no. 4. 1999, pp. 355–372. DOI: 10.1017/S0956796899003500 (Cited on page 59).

[116] Inostroza, P. & Storm, T. van der. Modular Interpreters for the Masses Implicit Context Propagation Using Object Algebras. No. Section 3. 2015. DOI: 10.1145/2814204.2814209 (Cited on pages 143, 145).

[117] Iovino, L., Pierantonio, A. & Malavolta, I. On the impact significance of metamodel evolution in MDE. *Journal of Object Technology*, vol. 11, no. 3. 2012. DOI: 10.5381/jot.2012.11.3.a3 (Cited on page 169).

[118] ISO. *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Tech. rep. Geneva, CH: International Organization for Standardization, 2011 (Cited on pages 31, 141).

[119] Iversen, J., Nielsen, P. A. & Norbjerg, J. Situated Assessment of Problems in Software Development. *ACM SIGMIS Database: the Database for Advances in Information Systems*, vol. 30, no. 2. 1999, pp. 66–81 (Cited on page 88).

[120] Jagadish, H., Mumick, I. S. & Silberschatz, A. View maintenance issues for the chronicle data model. In: *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1995, pp. 113–124. ISBN: 0-89791-730-8. DOI: http://doi.acm.org/10.1145/212433.220201. (Cited on page 50).

[121] Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J. & Tilkov, S. Microservices: The journey so far and challenges ahead. *IEEE Software*, vol. 35, no. 3. 2018, pp. 24–35. DOI: 10.1109/MS.2018.2141039 (Cited on pages 4, 125).

[122] Jansen, A. & Bosch, J. Software Architecture as a Set of Architectural Design Decisions. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*,

vol. 2005, no. May. 2005, pp. 109–120. DOI: 10.1109/WICSA.2005.61. (Cited on pages 12, 138, 147, 176).

[123] Jansen, S., Ballintijn, G. & Brinkkemper, S. A Process Model and Typology for Software Product Updaters. *Ninth European Conference on Software Maintenance and Reengineering*. 2005, pp. 265–274. DOI: 10.1109/CSMR.2005.3. (Cited on page 33).

[124] Jansen, S. Theres no business like software business: trends in software intensive business research. In: vol. 370 LNBIP. Springer, 2019, pp. 19–27. ISBN: 9783030337414. DOI: 10.1007/978-3-030-33742-1_3 (Cited on page 4).

[125] Jansen, S. A Focus Area Maturity Model for Software Ecosystem Governance. *Information and Software Technology*, vol. 118, no. November 2019. 2020, p. 106219. DOI: 10.1016/j.infsof.2019.106219 (Cited on pages 88, 107, 108, 116, 117).

[126] Jansen, S., Brinkkemper, S. & Cusumano, M. A. Software ecosystems: Analyzing and managing business networks in the software industry. Edward Elgar Publishing, 2013. ISBN: 9781781955628. DOI: 10.4337/9781781955635 (Cited on pages 4, 7, 82, 116).

[127] Jensen, C. S., Dyreson, C. E., Böhlen, M. H., Clifford, J., Elmasri, R., Gadia, S. K., Grandi, F., Hayes, P. J., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N. A., Mitsopoulos, Y. G., Montanari, A., Nonen, D. A., Peressi, E., Pernici, B., Roddick, J. F., Sarda, N. L., Scalas, M. R., Segev, A., Snodgrass, R. T., Soo, M. D., Tansel, A. U., Tiberio, P. & Wiederhold, G. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. *Temporal Databases, Dagstuhl*, no. February. 1998, pp. 367–405. DOI: 10.1007/BFb0053710 (Cited on page 26).

[128] Jolak, R., Ho-Quang, T., Michel, R. V. & Schiffelers, R. R. Model-based software engineering: a multiple-case study on challenges and development efforts. In: Association for Computing Machinery, Inc, Oct. 2018, pp. 213–223. ISBN: 9781450349499. DOI: 10.1145/3239372.3239404 (Cited on page 3).

[129] Jones, N. D., Gomard, C. K. & Sestoft, P. Partial Evaluation and Automatic Program Generation. Prentice-Hall International, 1993. (Cited on page 143).

[130] Jong, M. de & Deursen, A. van. Continuous Deployment and Schema Evolution in SQL Databases. In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. 2015, pp. 16–19. ISBN: 978-1-4673-7070-7. DOI: 10.1109/RELENG.2015.14 (Cited on pages 27, 69).

[131] Jörges, S. Construction and evolution of code generators: A model-driven and service-oriented approach. Vol. 7747. 2013, pp. 1–265. DOI: 10.1007/3-540-68339-9_34. (Cited on pages 143–145).

[132] Kabbedijk, J., Bezemer, C.-P., Jansen, S. & Zaidman, A. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, vol. 100. 2015, pp. 139–148. DOI: 10.1016/j.jss.2014.10.034. (Cited on pages 6, 148).

[133] Kabbedijk, J., Jansen, S. & Brinkkemper, S. A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software. In: *Proceedings of*

the 17th European Conference on Pattern Languages of Programs. ACM, 2012, p. 2. DOI: 10.1145/2602928.2603078 (Cited on pages 26, 44).

[134] Kassab, M., Mazzara, M., Lee, J. Y. & Succi, G. Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, vol. 14, no. 4. 2018, pp. 263–271. DOI: 10.1007/s11334-018-0319-4. (Cited on page 44).

[135] Keller, W. The Bridge to the New Town - A Legacy System Migration Pattern. In: *EuroPLoP*. 2000, pp. 261–268 (Cited on page 27).

[136] Kelly, S. & Tolvanen, J.-P. Domain-Specific Modeling: enabling full code generation. John Wiley & Sons, 2008 (Cited on pages 136, 151).

[137] Khorram, F., Mottu, J.-m., Sunyé, G., Khorram, F., Mottu, J.-m., Challenges, G. S. & Testing, O. L.-c. Challenges & Opportunities in Low-Code Testing. 2020 (Cited on page 183).

[138] Kitchenham, B. & Charters, S. Guidelines for performing Systematic Literature Reviews in Software Engineering. *Keele University and Durham University Joint Report*. 2007 (Cited on pages 14, 89).

[139] Klein, G., Andronick, J., Keller, G., Matichuk, D., Murray, T. & OConnor, L. Provably trustworthy systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375. 2104 Oct. 2017. DOI: 10.1098/rsta.2015.0404 (Cited on page 5).

[140] Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc., 2017 (Cited on page 50).

[141] Kleppmann, M. Thinking in events: from databases to distributed collaboration software: keynote at the 15th acm international conference on distributed and event-based systems (debs). In: Association for Computing Machinery, Inc, June 2021, pp. 15–24. ISBN: 9781450385558. DOI: 10.1145/3465480.3467835 (Cited on page 4).

[142] Klint, P. Interpretation Techniques. *Software: Practice and Experience*, vol. 11, no. June 1979. 1981, pp. 963–973 (Cited on pages 142, 143).

[143] Klint, P., Storm, T. van der & Vinju, J. Rascal, 10 years later. In: IEEE, Sept. 2019, pp. 139–139. ISBN: 978-1-7281-4937-0. DOI: 10.1109/SCAM.2019.00023. (Cited on page 3).

[144] Kögel, S. Recommender system for model driven software development. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Vol. Part F130154. Association for Computing Machinery, Aug. 2017, pp. 1026–1029. ISBN: 9781450351058. DOI: 10.1145/3106237.3119874 (Cited on page 159).

[145] Korkmaz, N. Practitioners view on command query responsibility segregation. MA thesis. Lund University, 2014 (Cited on page 26).

[146] Kruchten, P., Briand, L., Bianculli, D., Nejati, S., Pastore, F. & Sabetzadeh, M. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. *IEEE Software*, vol. 34, no. 05. 2017, pp. 72–75. DOI: 10.1109/MS.2017.3571562 (Cited on pages 16, 181).

[147] Kruchten, P., Capilla, R. & Duenas, J. C. The Decision View's Role in Software Architecture Practice. *IEEE Software*, vol. 26, no. 2. Mar. 2009, pp. 36–42. DOI: 10.1109/MS.2009.52. (Cited on pages 138, 140).

[148] Lämmel, R. Coupled Software Transformations - Revisited. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*. 2016, pp. 239–252. ISBN: 9781450344470. DOI: 10.1145/2997364.2997366. (Cited on page 169).

[149] Lassing, N., Rijsenbrij, D. & Van Wet, H. Towards a broader view on software architecture analysis of flexibility. *Proceedings - 6th Asia Pacific Software Engineering Conference, APSEC 1999*. 1999, pp. 238–245. DOI: 10.1109/APSEC.1999.809608 (Cited on page 53).

[150] Lees, C., Mallick, A., Paar, T., Fontenay, E. & Pimakova, K. APIs: The Digital Glue - How Banks Can Thrive in an API Economy. 2019. URL: https://www.accenture.com/_acnmedia/PDF-100/Accenture-How-Banks-Can-Thrive-API-Economy.pdf (Cited on page 85).

[151] Lehman, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, vol. 1. Jan. 1980, pp. 213–221. DOI: 10.1016/0164-1212(79)90022-0. (Cited on pages 4, 10).

[152] Lehman, M. M. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, vol. 68, no. 9. 1980, pp. 1060–1076. DOI: 10.1109/PROC.1980.11805 (Cited on page 24).

[153] Lehnert, S. A taxonomy for software change impact analysis. In: *IWPSE-EVOL'11 - Proceedings of the 12th International Workshop on Principles on Software Evolution*. 2011, pp. 41–50. ISBN: 9781450308489. DOI: 10.1145/2024445.2024454 (Cited on pages 160, 170).

[154] Li, Z., Liang, P. & Avgeriou, P. Application of knowledge-based approaches in software architecture: A systematic mapping study. *Information and Software Technology*, vol. 55, no. 5. 2013, pp. 777–794. DOI: 10.1016/j.infsof.2012.11.005. (Cited on page 44).

[155] Luckham, D. C. Event processing for business: organizing the real-time enterprise. John Wiley & Sons, 2011 (Cited on page 50).

[156] Luo, Y., Liang, P., Wang, C., Shahin, M. & Zhan, J. Characteristics and Challenges of Low-Code Development: The Practitioners' Perspective. July 2021. DOI: 10.1145/3475716.3475782. (Cited on pages 3, 7, 8, 183).

[157] Maddodi, G., Jansen, S., Guelen, J. P. & Jong, R. de. The Daily Crash : A Reflection on Continuous Performance Testing. In: *ICSEA 2016, The Eleventh International Conference on Software Engineering Advances*. 2016, pp. 100–107. ISBN: 9781612084985 (Cited on page 26).

[158] Mathijssen, M., Overeem, M. & Jansen, S. Identification of Practices and Capabilities in API Management: A Systematic Literature Review. 2020. URL: http://arxiv.org/abs/2006.10481 (Cited on pages 14, 18, 89, 113, 118).

[159] Mathijssen, M., Overeem, M. & Jansen, S. Source Data for the Focus Area Maturity Model for API Management. 2020. URL: https://arxiv.org/abs/2007.10611v3 (Cited on pages 18, 89, 95, 107, 111, 113, 117, 118, 178).

[160] Maule, A, Emmerich, W & Rosenblum, D. Impact analysis of database schema changes. *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 451–460. DOI: 10.1145/1368088.1368150 (Cited on page 182).

[161] Maule, A., Emmerich, W. & Rosenblum, D. S. Impact analysis of database schema changes. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. ACM, 2008, pp. 451–460. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368150 (Cited on pages 27, 66).

[162] Medjaoui, M., Wilde, E., Mitra, R. & Amundsen, M. Continuous API Management: Making the Right Decisions in an Evolving Landscape. O'Reilly Media, 2018. ISBN: 9781492043553 (Cited on pages 82, 119, 125).

[163] Meijler, T. D., Nytun, J. P., Prinz, A. & Wortmann, H. Supporting fine-grained generative model-driven evolution. *Software & Systems Modeling*, vol. 9, no. 3. 2010, pp. 403–424. DOI: 10.1007/s10270-009-0144-1 (Cited on pages 137, 141, 143–146, 151).

[164] Meißner, D., Erb, B., Kargl, F. & Tichy, M. retro-λ : An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 2018, pp. 76–87. ISBN: 9781450357821 (Cited on page 59).

[165] Mendix. *The State of Low-Code 2021: a Look Back, the Light Ahead*. Tech. rep. 2021 (Cited on pages 7, 8).

[166] Mernik, M., Heering, J. & Sloane, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, vol. 37, no. 4. 2005, pp. 316–344. DOI: 10.1145/1118890.1118892 (Cited on pages 141, 143, 146).

[167] Mettler, T., Rohner, P. & Winter, R. Towards A Classification of Maturity Models in Information Systems. In: *Management of the interconnected world*. Springer, 2010, pp. 333–340 (Cited on page 88).

[168] Meurice, L., Nagy, C. & Cleve, A. Detecting and Preventing Program Inconsistencies under Database Schema Evolution. *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2016, pp. 262–273. DOI: 10.1109/QRS.2016.38. (Cited on pages 27, 66).

[169] Meyer, B. Object-Oriented Software Construction. Prentice-Hall, 1988. ISBN: 0-13-629031-0 (Cited on pages 8, 25).

[170] Michelson, B. M. *Event-driven architecture overview*. Tech. rep. Patricia Seybold Group, 2006, pp. 210–1571 (Cited on pages 26, 57).

[171] Murillas, E. G. L. de, Aalst, W. M. van der & Reijers, H. A. Process mining on databases: Unearthing historical data from redo logs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9253. 2015, pp. 367–385. DOI: 10.1007/978-3-319-23063-4_25 (Cited on page 51).

[172] Musil, J., Musil, A. & Biffl, S. SIS: An architecture pattern for collective intelligence systems. *ACM International Conference Proceeding Series*, vol. 08-12-July. 2015, pp. 21–30. DOI: 10.1145/2855321.2855342 (Cited on page 45).

[173] Neamtiu, I. & Dumitraş, T. Cloud software upgrades: Challenges and opportunities. In: *2011 International Workshop on the Maintenance and Evolution of*

Service-Oriented and Cloud-Based Systems. 2011, pp. 1–10. ISBN: 978-1-4577-0645-5. DOI: 10.1109/MESOCA.2011.6049037 (Cited on page 24).

[174] NEventStore Dev team. NEventStore. 2019. URL: http://neventstore.org (Cited on page 70).

[175] Nielsen, J. Card Sorting to Discover the Users Model of the Information Space. *Nielsen Norman Group*. 1995 (Cited on pages 88, 108, 111, 178).

[176] Okoli, C. A Guide to Conducting a Standalone Systematic Literature Review. 2015 (Cited on pages 14, 89).

[177] Oltrogge, M., Derr, E., Stransky, C., Acar, Y., Fahl, S., Rossow, C., Pellegrino, G., Bugiel, S. & Backes, M. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In: *Proceedings - IEEE Symposium on Security and Privacy*. Vol. 2018-May. 2018, pp. 634–647. ISBN: 9781538643525. DOI: 10.1109/SP.2018.00005 (Cited on page 8).

[178] Onwuegbuzie, A. J. & Leech, N. L. Validity and qualitative research: An oxymoron? *Quality and Quantity*, vol. 41, no. 2. 2007, pp. 233–249. DOI: 10.1007/s11135-006-9000-3 (Cited on page 71).

[179] Ousterhout, J. K. Scripting: Higher-Level Programming for the 21st Century. *Computer*, vol. 31, no. 3. 1998, pp. 23–30 (Cited on pages 143, 144).

[180] Overeem, M. & Jansen, S. An Exploration of the 'It' in 'It Depends': Generative versus Interpretative Model-Driven Development. In: *5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. 2017, pp. 100–111 (Cited on page 19).

[181] Overeem, M. & Jansen, S. Proposing a Framework for Impact Analysis for Low-Code Development Platforms. In: *MODELS '21: Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (2nd LowCode Workshop)*. 2021 (Cited on pages 16, 19).

[182] Overeem, M., Jansen, S. & Fortuin, S. Generative versus Interpretive Model-Driven Development: Moving past 'It Depends'. In: *Model-Driven Engineering and Software Development. MODELSWARD 2017. Comm. in Comp. and Inf. Science*. Ed. by Pires, L., Hammoudi, S. & Selic, B. Vol. 880. Cham: Springer International Publishing, 2018, pp. 222–246. ISBN: 9783319947631. DOI: 10.1007/978-3-319-94764-8_10 (Cited on pages 16, 19, 116, 157, 162).

[183] Overeem, M., Jansen, S. & Mathijssen, M. API Management Maturity of Low-Code Development Platforms. In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Augusto Adriano, Gill, A., Nurcan Selmin, Reinhartz-Berger Iris, Schmidt Rainer & Zdravkovic Jelena. Cham: Springer International Publishing, 2021, pp. 380–394. ISBN: 978-3-030-79186-5 (Cited on pages 19, 157, 168).

[184] Overeem, M., Jansen, S. & Mathijssen, M. Evaluations of the API Management Maturity of Four Low-Code Development Platforms. 2021. DOI: 10.17632/wdtg5ytdpf.1 (Cited on pages 19, 121, 129).

[185] Overeem, M., Mathijssen, M. & Jansen, S. API-m-FAMM: a Focus Area Maturity Model for API Management. *Information and Software Technology*, vol. 147.

2022, p. 106890. DOI: https://doi.org/10.1016/j.infsof.2022.106890 (Cited on pages 16, 18).

[186] Overeem, M., Spoor, M. & Jansen, S. The Dark Side of Event Sourcing: Managing Data Conversion. In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 193–204. ISBN: 9781509055012 (Cited on pages 16, 18, 44, 46, 47, 65, 66, 157).

[187] Overeem, M., Spoor, M., Jansen, S. & Brinkkemper, S. An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry. *Journal of Systems and Software*, vol. 178, no. 110970. 2021. DOI: 10.1016/j.jss.2021.110970 (Cited on pages 16, 18, 161, 164).

[188] Overeem, M., Spoor, M., Jansen, S. & Brinkkemper, S. An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry - Accompanying Anonymized Transcripts. 2021. DOI: 10.17632/dgbxyn7yw3.1 (Cited on pages 14, 18, 48, 50, 77).

[189] Pantelimon, S. G., Rogojanu, T., Braileanu, A., Stanciu, V. D. & Dobre, C. Towards a seamless integration of iot devices with iot platforms using a low-code approach. *IEEE 5th World Forum on Internet of Things, WF-IoT 2019 - Conference Proceedings*. 2019, pp. 566–571. DOI: 10.1109/WF-IoT.2019.8767313 (Cited on pages 7, 156).

[190] Paulk, M. C., Curtis, B., Chrissis, M. B. & Weber, C. V. Capability Maturity Model, Version 1.1. *IEEE software*, vol. 10, no. 4. 1993, pp. 18–27 (Cited on page 88).

[191] Perry, D. E. "Large" abstractions for software engineering. In: *Proceedings of the 2nd international workshop on The role of abstraction in software engineering - ROA '08*. New York, New York, USA: ACM Press, 2008, p. 31. DOI: 10.1145/1370164.1370172 (Cited on page 4).

[192] Pessoa, L., Fernandes, P., Castro, T., Alves, V., Rodrigues, G. N. & Carvalho, H. Building reliable and maintainable Dynamic Software Product Lines: An investigation in the Body Sensor Network domain. *Information and Software Technology*, vol. 86. 2017, pp. 54–70. DOI: 10.1016/j.infsof.2017.02.002 (Cited on page 143).

[193] Poell, T., Nieborg, D. & Dijck, J. van. Platformisation. *Internet Policy Review*, vol. 8, no. 4. 2019, pp. 1–13. DOI: 10.14763/2019.4.1425 (Cited on pages 7, 82, 116).

[194] Polák, M. & Holubová, I. REST API management and evolution using MDA. In: *C3S2E '15: Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*. 2015, pp. 102–109. ISBN: 9781450334198. DOI: 10.1145/2790798.2790820 (Cited on page 126).

[195] Popescu, D., Garcia, J., Bierhoff, K. & Medvidovic, N. Impact analysis for distributed event-based systems. *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS'12*. 2012, pp. 241–251. DOI: 10.1145/2335484.2335511 (Cited on pages 164, 170).

[196] Pöppelbuß, J. & Röglinger, M. What Makes A Useful Maturity Model? A Framework of General Design Principles for Maturity Models and its Demonstration in Business Process Management. 2011 (Cited on page 86).

[197] Prat, N., Comyn-Wattiau, I. & Akoka, J. A Taxonomy of Evaluation Methods for Information Systems Artifacts. *Journal of Management Information Systems*, vol. 32, no. 3. 2015, pp. 229–267 (Cited on pages 108, 111, 178).

[198] Proença, D. & Borbinha, J. Maturity models for Information Systems: A State of the Art. *Procedia Computer Science*, vol. 100. 2016, pp. 1042–1049 (Cited on pages 106, 107).

[199] Prooph Components. Prooph. 2019. URL: http://getprooph.org/ (Cited on page 70).

[200] Pulkkinen, V. Continuous deployment of software. In: *Proc. of the Seminar no. 58312107: Cloud-based Software Engineering*. 2013, pp. 46–52 (Cited on page 33).

[201] Qiu, D., Li, B. & Su, Z. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 125–135. DOI: 10.1145/2491411.2491431. (Cited on pages 24, 27).

[202] Rademacher, F., Sorgalla, J. & Sachweh, S. Challenges of Domain-Driven Microservice Design. *IEEE Software*. 2018, p. 8. DOI: 10.1109/MS.2018.2141028 (Cited on page 4).

[203] Ralph, P. Fundamentals of software design science. PhD thesis. University of British Columbia, 2010. (Cited on page 13).

[204] Ralph, P., Baltes, S., Bianculli, D., Dittrich, Y., Felderer, M., Feldt, R., Filieri, A., Furia, C. A., Graziotin, D., He, P., Hoda, R., Juristo, N., Kitchenham, B. A., Robbes, R., Méndez, D., Molleri, J., Spinellis, D., Staron, M., Stol, K., Tamburri, D. A., Torchiano, M., Treude, C., Turhan, B. & Vegas, S. Empirical Standards for Software Engineering Research. 2021. URL: https://arxiv.org/abs/2010.03525 (Cited on pages 13, 14, 117).

[205] Ralph, P. & Wand, Y. *A Proposal for a Formal Definition of the Design Concept*. Tech. rep. 2009, pp. 103–136 (Cited on page 13).

[206] Richardson, C. & Rymer, J. R. New Development Platforms Emerge For Customer-Facing Applications. 2014 (Cited on pages 3, 7).

[207] Riehle, D., Fraleigh, S., Bucka-Lassen, D. & Omorogbe, N. The architecture of a UML virtual machine. *International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA)*, no. February. 2001, pp. 327–341. DOI: 10.1145/504311.504306 (Cited on pages 143, 144).

[208] Rinderle-Ma, S., Reichert, M. & Weber, B. On the formal semantics of change patterns in process-aware information systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5231 LNCS. 2008, pp. 279–293. DOI: 10.1007/978-3-540-87877-3-21 (Cited on page 170).

[209] Roddick, J. F. A survey of schema versioning issues for database systems. *Information and Software Technology*, vol. 37, no. 7. Jan. 1995, pp. 383–393. DOI: 10.1016/0950-5849(95)91494-K (Cited on pages 31, 182).

[210] Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. a., Baer, J.-L., Bershad, B. N. & Levy, H. M. The structure and performance of interpreters. *ACM SIGPLAN Notices*, vol. 31, no. 9. 1996, pp. 150–159. DOI: 10.1145/248209.237175 (Cited on pages 142, 143).

[211] Rose, L., Paige, R., Kolovos, D. & Polack, F. An analysis of approaches to model migration. In: *Models and Evolution (MoDSE-MCCM) Workshop*. 2009, pp. 6–15. (Cited on pages 163, 169).

[212] Runeson, P. & Höst, M. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, vol. 14, no. 2. 2009, p. 131 (Cited on page 109).

[213] Ruscio, D. D., Kolovos, D., Lara, J. de, Pierantonio, A., Tisi, M. & Wimmer, M. Low-code development and model-driven engineering: two sides of the same coin? *Software and Systems Modeling*. Jan. 2022. DOI: 10.1007/s10270-021-00970-2. (Cited on page 3).

[214] Saaty, T. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, vol. 48, no. 1. 1990, pp. 9–26. DOI: 10.1016/0377-2217(90)90057-i. (Cited on pages 146, 149).

[215] Sadalage, P. J. & Fowler, M. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley, 2012 (Cited on pages 27, 31, 68).

[216] Sadri, F. & Kowalski, R. Variants of the Event Calculus Fariba Sadri and Robert Kowalski Abstract. *Proceedings of the Twelfth International Conference on Logic Programming*, no. October. 1995, pp. 67–81 (Cited on page 50).

[217] Sahay, A., Indamutsa, A., Di Ruscio, D. & Pierantonio, A. Supporting the understanding and comparison of low-code development platforms. In: *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*. 2020, pp. 171–178. ISBN: 9781728195322. DOI: 10.1109/SEAA51224.2020.00036 (Cited on pages 8, 156).

[218] Sanchez-Puchol, F. & Pastor-Collado, J. A. Focus area maturity models: A comparative review. *Lecture Notes in Business Information Processing*, vol. 299. 2017, pp. 531–544. DOI: 10.1007/978-3-319-65930-5_42 (Cited on page 182).

[219] Sanchis, R., García-Perales, O., Fraile, F. & Poler, R. Low-code as enabler of digital transformation in manufacturing industry. *Applied Sciences (Switzerland)*, vol. 10, no. 1. 2020. DOI: 10.3390/app10010012 (Cited on pages 8, 116, 156).

[220] Santos, J. C. S., Sejfia, A., Corrello, T., Gadenkanahalli, S. & Mirakhorli, M. Achilles heel of plug-and-Play software architectures: A grounded theory based approach. *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 671–682. DOI: 10.1145/3338906.3338969 (Cited on page 46).

[221] Santos, W. APIs Show Faster Growth Rate in 2019 Than Previous Years. 2019. URL: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17 (Cited on page 82).

[222] Sato, D. ParallelChange. 2014. URL: http://martinfowler.com/bliki/ParallelChange.html (Cited on page 35).

[223] Saur, K., Dumitraş, T. & Hicks, M. Evolving NoSQL Databases Without Downtime. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 166–176. (Cited on pages 26, 27, 68).

[224] Scherzinger, S., Klettke, M. & Störl, U. Managing Schema Evolution in NoSQL Data Stores. In: *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013)*. Ed. by Green, T. J. & Schmitt, A. 2013. (Cited on pages 26, 27, 30, 68).

[225] Scherzinger, S., Klettke, M. & Störl, U. Cleager: Eager Schema Evolution in NoSQL Document Stores. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*. 2015, pp. 659–662 (Cited on pages 24, 31).

[226] Schramm, A., Preußner, A., Heinrich, M. & Vogel, L. Rapid UI development for enterprise applications: Combining manual and model-driven techniques. *Models*, vol. 6394 LNCS, no. PART 1. 2010, pp. 271–285. DOI: 10.1007/978-3-642-16145-2_19. (Cited on page 143).

[227] Schunselaar, D. M. M., Gulden, J, Schuur, H. van der & Reijers, H. A. A Systematic Evaluation of Enterprise Modelling Approaches on Their Applicability to Automatically Generate Software. In: *18th IEEE Conference on Business Informatics*. 2016 (Cited on page 146).

[228] Schuur, H. van der, Ven, E. van de, Jong, R. de, Schunselaar, D., Reijers, H. A., Overeem, M., Graaf, M. de, Jansen, S. & Brinkkemper, S. NEXT: Generating tailored ERP applications from ontological enterprise models. In: *IFIP Working Conference on The Practice of Enterprise Modeling*. Vol. 305. Springer, 2017, pp. 283–298. ISBN: 9783319702407. DOI: 10.1007/978-3-319-70241-4_19 (Cited on pages 17, 161, 162).

[229] Sein, M. K.., Henfridsson, O., Purao, S., Rossi, M., Lindgren, R. & Sein, M. K.. Action Design Research. *Source: MIS Quarterly*, vol. 35, no. 1. 2011, pp. 37–56. DOI: 10.2307/23043488 (Cited on page 70).

[230] Slotos, T. The star pattern - Representing domain concepts in a uniform way. *ACM International Conference Proceeding Series*, no. July 2016. 2016, p. 8. DOI: 10.1145/3011784.3011792 (Cited on page 45).

[231] Sorgalla, J., Rademacher, F., Sachweh, S. & Zündorf, A. On Collaborative Model-driven Development of Microservices. In: *MSE Workshop @ STAF2018*. 2018, pp. 1–8. (Cited on page 4).

[232] Spinellis, D. The changing role of the software architect. *IEEE Software*, vol. 33. 6 Nov. 2016, pp. 4–6. DOI: 10.1109/MS.2016.133. (Cited on page 5).

[233] Spruit, M. & Röling, M. ISFAM: The Information Security Focus Area Maturity Model. 2014 (Cited on pages 106–108, 182).

[234] Stahl, T., Völter, M., Bettin, J., Haase, A. & Helsen, S. Model-Driven Software Development: Technology, Engineering, Management. 2006, p. 446. ISBN: 978-0-470-02570-3 (Cited on pages 141, 143).

[235] Steenbergen, M. van, Bos, R., Brinkkemper, S., Weerd, I. van de & Bekkers, W. The design of focus area maturity models. In: *International Conference on Design Science Research in Information Systems*. Vol. 662. Berlin: Springer, 2010, pp. 317–332 (Cited on pages 13, 18, 82, 88, 118, 178).

[236] Steenbergen, M. van, Bos, R., Brinkkemper, S., Weerd, I. van de & Bekkers, W. Improving IS Functions Step by Step: The use of focus area maturity models. *Scandinavian Journal of Information Systems*, vol. 25, no. 2. 2013, pp. 35–56 (Cited on pages 13, 18, 82, 88, 107, 178).

[237] Stol, K.-J., Ralph, P. & Fitzgerald, B. Grounded Theory in Software Engineering Research : A Critical Review and Guidelines. *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, no. Aug. 2015, pp. 120–131. DOI: http://dx.doi.org/10.1145/2884781.2884833 (Cited on page 47).

[238] Sundharam, S. M., Altmeyer, S. & Navet, N. Model Interpretation for an AUTOSAR compliant Engine Control Function. In: *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2016 (Cited on page 143).

[239] Svahnberg, M., Wohlin, C., Lundberg, L. & Mattsson, M. A Quality-Driven Decision-Support Method for Identifying Software Architecture Candidates. *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, no. 05. 2003, pp. 547–573. DOI: 10.1142/S0218194003001421 (Cited on page 138).

[240] Taibi, D., Lenarduzzi, V. & Pahl, C. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, vol. 2018-Janua. 2018, pp. 221–232. DOI: 10.5220/0006798302210232 (Cited on page 44).

[241] Tamburri, D. A. & Kazman, R. General methods for software architecture recovery: a potential approach and its evaluation. *Empirical Software Engineering*, vol. 23, no. 3. 2018, pp. 1457–1489. DOI: 10.1007/s10664-017-9543-z (Cited on page 46).

[242] Tan, L. & Katayama, T. Meta Operations for Type Management in Object-Oriented Databases. In: *DOOD*. 1989, pp. 241–258 (Cited on page 31).

[243] Tanković, N. *Model Driven Development Approaches : Comparison and Opportunities*. Tech. rep. (Cited on pages 141, 143–146).

[244] Tanković, N., Vukotić, D. & Žagar, M. Rethinking Model Driven Development : Analysis and Opportunities. *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference*. 2012, pp. 505–510. DOI: 10.2498/iti.2012.0414. (Cited on pages 143–145).

[245] The Apache Software Foundation. Apache Avro. 2019. URL: http://avro.apache.org/ (Cited on pages 57, 60, 68).

[246] Thibault, S. & Consel, C. A framework for application generator design. *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 3, May 1997. 1997, pp. 131–135. (Cited on page 143).

[247] Thibault, S. A., Marlet, R. & Consel, C. Domain-Specific Languages : From Design to Implementation Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, vol. 25, no. 3. 1999, pp. 363–377 (Cited on page 143).

[248] Tisi, M., Mottu, J. M., Kolovos, D. S., Lara, J. de, Guerra, E., Di Ruscio, D., Pierantonio, A. & Wimmer, M. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. *CEUR Workshop Proceedings*, vol. 2405. 2019 (Cited on page 183).

[249] Toulmé, A. Presentation of EMF Compare Utility. In: *Eclipse Modeling Symposium*. 2006 (Cited on page 169).

[250] Tragatschnig, S., Stevanetic, S. & Zdun, U. Supporting the evolution of event-driven service-oriented architectures using change patterns. *Information and Software Technology*, vol. 100. 2018, pp. 133–146. DOI: 10.1016/j.infsof.2018.04.005 (Cited on page 170).

[251] Tung, T. Accenture API Management Suite API Maturity Model. 2014. URL: https://expydoc.com/doc/4730985/download-pdf (Cited on pages 83, 88).

[252] Varró, G., Anjorin, A. & Schürr, A. Unification of compiled and interpreter-based pattern matching techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7349 LNCS. 2012, pp. 368–383. DOI: 10.1007/978-3-642-31491-9_28 (Cited on pages 143, 144).

[253] Vassiliadis, P. A survey of Extract transform Load technology. *International Journal of Data Warehousing & Mining*, vol. 5, no. 3. 2009, pp. 1–27. DOI: 10.4018/jdwm.2009070101. (Cited on page 50).

[254] Ven, J. S. van der, Jansen, A. G. J., Nijhuis, J. A. G. & Bosch, J. Design decisions: The bridge between rationale and architecture. *Rationale Management in Software Engineering*. 2006, pp. 329–348. DOI: 10.1007/978-3-540-30998-7_16 (Cited on pages 138, 147).

[255] Vernon, V. Implementing Domain-Driven Design. Addison-Wesley, 2013 (Cited on page 50).

[256] Vincent, P., Iijima, K., Driver, M., Wong, J. & Natis, Y. *Magic Quadrant for Enterprise Low-Code Application Platforms*. Tech. rep. September. Gartner, 2019, pp. 1–33 (Cited on pages 18, 117, 124, 126).

[257] Vinju, J. Kan de biologie een rol spelen in het oplossen van problemen die gepaard gaan met de groeiende technologische complexiteit van onze software? In: *Hoe zwaar is licht?* Ed. by de Graaf, B. & Rinnooy Kan, A. Uitgeverij Balans, 2017 (Cited on page 4).

[258] Visser, J. Change is the constant: keynote. *Ercim News*, vol. 2012. 2012, pp. 3–3 (Cited on page 4).

[259] Voelter, M. Best Practices for DSLs and Model-Driven Software Development. *Journal of Object Technology*, vol. 8, no. 6. 2009, pp. 79–102. (Cited on pages 141, 143, 145, 146).

[260] Voelter, M. & Visser, E. Product Line Engineering Using Domain-Specific Languages. *15th International Software Product Line Conference*, no. Section II. 2011, pp. 70–79. DOI: 10.1109/SPLC.2011.25 (Cited on pages 143, 145).

[261] Vogels, W. Eventually consistent. *Communications of the ACM*, vol. 52, no. 1. 2009, pp. 40–44 (Cited on pages 9, 25, 61).

[262] Wachsmuth, G. Metamodel adaptation and model co-adaptation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4609 LNCS. 2007, pp. 600–624. DOI: 10.1007/978-3-540-73589-2_28. (Cited on page 169).

[263] Waszkowski, R. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, vol. 52. 10 2019, pp. 376–381. DOI: 10.1016/j.ifacol.2019.10.060. (Cited on page 3).

[264] Weber, B., Rinderle, S. & Reichert, M. Change patterns and change support features in process-aware information systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4495 LNCS. 2007, pp. 574–588. DOI: 10.1007/978-3-540-72988-4_40 (Cited on page 170).

[265] Weber, B., Rinderle-Ma, S. & Reichert, M. Change Support in Process-Aware Information Systems-A Pattern-Based Analysis. *Data Knowledge Eng*, vol. 66, no. 3. 2007, pp. 438–466. (Cited on page 168).

[266] Weir, L. Enterprise API Management: Design and Deliver Valuable Business APIs. Packt Publishing Ltd, 2019 (Cited on pages 9, 82, 107, 125).

[267] Wieringa, R. J. Design Science Methodology for Information Systems and Software Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8. (Cited on page 13).

[268] Wlaschin, S. Domain Modeling Made Functional. The Pragmatic Bookshelf, 2018 (Cited on page 59).

[269] Wohlin, C. Empirical Software Engineering Research with Industry: Top 10 Challenges. In: *CESI '14: Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry*. IEEE, 2013, pp. 43–46 (Cited on page 16).

[270] Wohlin, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*. 2014, pp. 1–10. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. (Cited on pages 14, 47, 141).

[271] WSO2. API Management Platform Technical Evaluation Framework. 2015. URL: https://wso2.com/whitepapers/api-management-platform-technical-evaluation-framework/ (Cited on page 84).

[272] Wu, E., Diao, Y. & Rizvi, S. High-performance complex event processing over streams. In: *Proceedings of the 2006 ACM SIGMOD international conference on*

Management of data - SIGMOD '06. 2006, p. 407. ISBN: 1595934340. DOI: 10.1145/1142473.1142520. (Cited on page 50).

[273] Xu, L. & Brinkkemper, S. Concepts of product software. *European Journal of Information Systems*, vol. 16, no. 5. 2007, pp. 531–541. DOI: 10.1057/palgrave.ejis.3000703 (Cited on page 8).

[274] Yin, R. K. Case Study Research and Applications: Design and Methods. 5th edition. Thousand Oaks, California: Sage Publications, Inc, 2017. ISBN: 978-1-4522-4256-9 (Cited on page 14).

[275] Young, G. CQRS and Event Sourcing. 2010. URL: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing (Cited on pages 5, 24, 50, 61, 173).

[276] Young, G. A Decade of DDD, CQRS, Event Sourcing - Domain-Driven Design Europe 2016. 2016. URL: https://www.youtube.com/watch?v=LDW0QWie21s (Cited on page 24).

[277] Young, G. Versioning in an Event Sourced System. Leanpub, 2017 (Cited on pages 50, 65, 69).

[278] Zhong, Y., Li, W. & Wang, J. Using event sourcing and CQRS to build a high performance point trading system. *ACM International Conference Proceeding Series*. 2019, pp. 16–19. DOI: 10.1145/3317614.3317632 (Cited on pages 50, 179).

[279] Zhu, L., Aurum, A., Gorton, I. & Jeffery, R. Tradeoff and Sensitivity Analysis in Software Architecture Evaluation Using Analytic Hierarchy Process. *Software Quality Journal*, vol. 13, no. 4. 2005, pp. 357–375. DOI: 10.1007/s11219-005-4251-0 (Cited on page 138).

[280] Zhu, M. Model-Driven Game Development Addressing Architectural Diversity and Game Engine-Integration. PhD thesis. Norwegian University of Science and Technology, 2014 (Cited on page 143).

[281] Zolotas, C., Chatzidimitriou, K. C. & Symeonidis, A. L. RESTsec: a low-code platform for generating secure by design enterprise services. *Enterprise Information Systems*, vol. 12, no. 8-9. Oct. 2018, pp. 1007–1033. DOI: 10.1080/17517575.2018.1462403 (Cited on pages 7, 156).

Summary

Our world is driven by software, from everything we do online to the cars we drive. The role software plays is so large that you could say that every company is a software company. Companies have to partake in this transformation to a software-driven world but face various challenges in doing so. There is a scarcity of software engineers which only appears to be increasing. Next to that, software development is complex and requires a joint effort of software engineers and domain experts. Finally, when the software is developed, the process does not stop. Companies do not operate in a vacuum; they are part of a larger world inhabited by customers, suppliers, governments, and competitors. Therefore, companies have to constantly update their software to comply with wishes and demands.

A recent development, low-code platforms, could be the solution to these challenges. The term *low-code* emphasizes that these platforms enable the development of software systems with a low effort of coding. Through the introduction of higher-level abstractions, such as domain-specific models, these platforms enable *citizen developers* (professionals without specific software development training) to develop software systems. Enabling untrained professionals to participate in the software development process not only means that fewer trained IT personnel are needed, but that the business side is also automatically more involved in the development of the software.

Low-code platforms have to live up to certain expectations to be successful. They have to support the development of modern, cloud-based applications. Successful software is no longer an application accessible only through a company computer, modern applications are always available from every device. Not only are they accessible for humans, we also expect these applications to be open for communication with other systems. This means that other companies can interact with the data and processes inside these applications to create collaborations. Finally, low-code platforms need to support companies in the long run. Maintaining software should be as easy as creating new software.

This dissertation presents, in three parts, the *evolution of low-code platforms* and how they support the new generation of digital companies. In each of these three parts the software architect and his role in the development of low-code platforms stands central.

The first part discusses *software evolution in event sourced systems*. Event-driven architectures enable the development of large and complex software systems. *Event sourcing* is a form of event-driven architecture that offers a lot of benefits for the software system by storing every change as an *event*. An increased flexibility in making

future changes is gained because the full history of a system is stored. However, event sourcing also introduces new challenges in the evolution of a software system. Various evolution techniques are presented that can be applied to confront these challenges.

API management in software ecosystems is central in the second part. Modern software systems are open, which means that external parties can connect with a software system to exchange data. Through these connections software ecosystems can grow. In this ecosystem, the central software system is enriched by external complementors. These connections are created through *Application Programming Interfaces*, abbreviated APIs. The management of these APIs is essential for the success of software ecosystems. Low-code platforms have to support these processes and enable them for *citizen developers* to be successful. The *API-m-FAMM* gives software architects a tool to evaluate and plan the improvement of their API management capabilities.

Evolution supporting architecture is the third and final part. Changes that are made within a low-code platform have an impact on other parts of the platform or even on complementors. Some of these changes, for example, require data conversion. The analysis of the impact is difficult because of the higher-level abstractions offered by low-code platforms. For companies using low-code platforms this analysis is essential in maintaining control over the software. The *Impact Analysis for Low-Code Development Platforms* framework allows software architects to design the process of software evolution, making sure that companies stay in control of their systems.

Nederlandse samenvatting

Onze wereld kan niet meer zonder software, van alles wat we online doen tot aan de auto's die we rijden. Software speelt een dermate grote rol dat je kan stellen dat elk bedrijf tegenwoordig een softwarebedrijf is. Bedrijven moeten mee doen aan deze transformatie naar een software-gedreven wereld, maar komen verschillende uitdagingen tegen. Zo is er een groot tekort aan professionele softwareontwikkelaars en het ziet er naar uit dat dit tekort alleen maar groter gaat worden. Daarnaast is het ontwikkelen van succesvolle software complex en vereist het de samenwerking tussen softwareontwikkelaars en domeinexperts. En als de software eenmaal ontwikkelt is stopt het proces niet. Bedrijven opereren niet in een vacuüm, maar zijn onderdeel van een wereld met klanten, leveranciers, concurrenten en overheden. Deze partijen zorgen ervoor dat bedrijven hun software constant moeten aanpassen om zo te zorgen dat deze blijft voldoen aan wensen en eisen.

Low-code platformen, een recente ontwikkeling, zouden wel eens de oplossing kunnen zijn voor deze uitdagingen. De term *low-code* refereert aan de minimale hoeveelheid code die nodig is om applicaties te ontwikkelen. Door het aanbieden van hogere abstracties, zoals domein specifieke modellen, maken deze platformen het mogelijk voor professionals zonder programmeer-training om applicaties te ontwikkelen. Deze professionals zijn veelal de domeinexperts die voorheen alleen konden vertellen wat ze nodig hadden. Door deze platformen zijn er dus niet alleen minder getrainde ontwikkelaars nodig, maar zijn de domeinexperts ook automatisch meer betrokken.

Om succesvol te zijn moeten low-code platformen de verwachtingen wel waar maken. Zo moeten ze het ontwikkelen van moderne cloud-applicaties ondersteunen. Succesvolle applicaties zijn niet langer alleen bereikbaar via een bedrijfscomputer, ze zijn beschikbaar vanaf elk apparaat. Daarnaast moeten deze applicaties ook beschikbaar zijn voor communicatie met andere software systemen. Dat betekent dat andere bedrijven toegang kunnen krijgen tot de data en de processen zodat ze kunnen samenwerken. En als laatste moet applicaties bijwerken net zo makkelijk zijn als nieuwe applicaties ontwikkelen.

In dit proefschrift bespreken we in drie delen de *evolutie van low-code platformen* en hoe zij de nieuwe generatie digitale bedrijven ondersteunen. In elk deel staat de software architect en zijn rol in de ontwikkeling van een low-code platform centraal.

Het eerste deel bespreekt *software evolutie in event sourced systemen*. Event-gedreven architecturen maken het mogelijk om grote en complexe software systemen te ontwikkelen. *Event sourcing* is een specifieke event-gedreven architectuurstijl die veel voordelen biedt aan architecten. In een event sourced systeem wordt elke wijziging in de

applicatie bijgehouden. Hierdoor is de volledige historie beschikbaar waardoor er flexibiliteit voor toekomstige aanpassingen ontstaat. De evolutie van het systeem wordt echter uitdagender, omdat de historie continue meegenomen moet worden. Software architecten kunnen met de gepresenteerde evolutie technieken deze uitdagingen het hoofd bieden.

API beheer in software ecosystemen staat centraal in het tweede gedeelte. Moderne applicaties ondersteunen koppelingen met externe software systemen om data uit te wisselen. Hierdoor kunnen zich software ecosystemen vormen waarin de centrale applicatie wordt verrijkt door externe partijen. In deze ecosystemen worden *Applicatie Programmeer Interfaces* (API's) gebruikt om te communiceren. Het beheren van deze API's is essentieel voor het succes van een software ecosysteem. Low-code platformen moeten het mogelijk maken voor professionals zonder programmeer-training om deze beheerstaken uit te voeren. Het ontwikkelde *API-m-FAMM* geeft software architecten een hulpmiddel om de volwassenheid van hun API beheer mogelijkheden vast te stellen en verbeteringen te plannen.

Evolutie ondersteunende architectuur is het derde en laatste deel van dit proefschrift. Er worden continue veranderingen gemaakt in een low-code platform, en deze veranderingen hebben impact op andere onderdelen van het platform, of zelfs op externe partijen die gebruik maken van de API's. Voorbeelden hiervan zijn veranderingen die het nodig maken om de bestaande data te transformeren. De nieuwe abstracties in low-code platformen maken het niet eenvoudig om deze analyse te doen. Voor bedrijven die gebruik maken van deze platformen is deze analyse noodzakelijk om grip te houden op de software. Het *Impact Analysis for Low-Code Development Platforms* framework toont software architecten hoe ze deze analyse mogelijk maken, zodat bedrijven hun applicaties onder controle houden.

Publication List

Publications Included in this Dissertation

M. Overeem, M. Mathijssen, and S. Jansen. *API-m-FAMM: a Focus Area Maturity Model for API Management.* In Information and Software Technology, volume 147 (2022), 106890.

M. Overeem, S. Jansen. (2021). *Proposing a Framework for Impact Analysis for Low-Code Development Platforms.* In MODELS 21: Proceedings of the 24thACM/IEEE International Conference on Model Driven Engineering Languagesand Systems: Companion Proceedings (2nd LowCode Workshop).

M. Overeem, M. Mathijssen, and S. Jansen. (2021). *API Management Maturity of Low-Code Development Platforms.* In Enterprise, Business-Process and Information Systems Modeling. BPMDS 2021, EMMSAD 2021. Lecture Notes in Business Information Processing, vol 421. Springer, Cham.

M. Overeem, M. Spoor, S. Jansen, and S. Brinkkemper. (2021). *An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry.* In Journal of Systems and Software: In Practice, volume 178 (2021), 110970.

M. Overeem, S. Jansen, and S. Fortuin. (2018). *Generative versus Interpretive Model-Driven Development: Moving Past 'It Depends'.* In Pires L., Hammoudi S., Selic B. (eds) Model-Driven Engineering and Software Development. MODELSWARD 2017. Communications in Computer and Information Science, vol 880. Springer, Cham.

M. Overeem, M. Spoor, and S. Jansen. (2017). *The Dark Side of Event Sourcing: Managing Data Conversion.* In Proceedings of the 24th Conference on Software Analysis, Evolution, and Reengineering (SANER 2017), pages 193-204.

Other Publications not Included

G. Maddodi, S. Jansen, and M. Overeem. (2020). *Aggregate Architecture Simulation in Event-Sourcing Applications using Layered Queuing Networks.* In Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20).

M. de Graaf, and M. Overeem. (2019). *Tackling Complexity in ERP Software: a Love Song to Bounded Contexts.* In Domain Driven Design, The First 15 Years (Leanpub).

M.Overeem, and S. Jansen. (2018). *Continuous Migration of Mass Customized Applications.* In Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop (BENEVOL 2018).

H. van der Schuur, E. van de Ven, R. de Jong, D. Schunselaar, H. Reijers, M. Overeem, M. de Graaf, S. Jansen, and S. Brinkkemper. (2017). *NEXT: Generating Tailored ERP Applications from Ontological Enterprise Models.* In Proceedings of the 10th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM'17).

M. Overeem, S. Jansen. (2017). *An Exploration of the 'It in 'It Depends: Generative versus Interpretive Model-Driven Development.* In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017).

P. Jansson, J. Jeuring, L. Cabenda, G. Engels, J. Kleerekoper, S. Mak, M. Overeem, and K. Visser. (2007). *Testing Properties of Generic Functions.* In Implementation and Application of Functional Languages, IFL 2006. Lecture Notes in Computer Science, vol 4449.

Accompanying Data Packages

M. Overeem, S. Jansen, and M. Mathijssen. (2021). *Evaluations of the API Management Maturity of Four Low-Code Development Platforms.*
`https://data.mendeley.com/datasets/wdtg5ytdpf/1`

M. Overeem, M. Spoor, S. Jansen, and S. Brinkkemper. (2021). *An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry - Accompanying Anonymized Transcripts.*
`https://data.mendeley.com/datasets/dgbxyn7yw3/1`

M. Mathijssen, M. Overeem, and S. Jansen. (2020). *Source Data for the Focus Area Maturity Model for API Management.*
`https://arxiv.org/abs/` 2007.10611v3

M. Mathijssen, M. Overeem, and S. Jansen. (2020). *Identification of Practices and Capabilities in API Management: A Systematic Literature Review.*
`http://arxiv.org/abs/2006.10481`

Presentations at Scientific and Industrial Conferences[1]

Low-Code Platforms, Tales from a Software Architect. (2022). At exec(ut).

A system with thousands of event types. (2021). At EventSourcing Live 2021.

Event system evolution - A Scientific Study on Event Sourcing, Lessons from Industry. (2021). At EventSourcing Live 2021.

Low code: AFAS software. (2021). Episode 24 of the DevTalks podcast.

A technical introduction to AFAS Focus. (2020). At the Mendix Tech Lead meeting.

103 years of event sourcing experience. (2019). At DDD Vienna Meetup.

[1] Slides and recordings, if available, can be found at `https://www.movereem.nl/pubspres`

Applying event sourcing and CQRS in a large ERP system. (2019). At DDD Vienna Meetup.

The last barrier: code the coder. (2019). At Emerce Next.

The Dark Side of Event Sourcing: Managing Data Conversion. (2018). At Landelijk Architectuur Congres (LAC) 2018.

Building an event sourced system in .NET. (2018). At Dutch .NET Group Meetup.

Event Sourcing after launch. (2018). At Drukwerkdeal.nl Developer Meetup.

Event Sourcing after launch, how to evolve your event store along with your application. (2018). At DDDEurope 2018.

The Dark Side of Event Sourcing: Managing Data Conversion. (2017). At KanDDDinsky 2017.

Event Sourcing after launch, how to evolve your event store along with your application. (2017). At TechDays 2017, Netherlands.

The Dark Side of Event Sourcing. (2017). At DomCode meetup.

The Dark Side of Event Sourcing. (2017). At DDD Belgium Meetup.

Safer Software Upgrades With Continuous Deployment and Model Driven Development. (2016). At the Belgium-Netherlands Software Evolution Workshop (BENEVOL) 2016.

Curriculum Vitæ

Michiel Overeem was born on June 15th, 1984 in The Netherlands. He studied Computer Science at Utrecht University from 2002 until 2007, obtaining a Master's degree in Software Technology in August, 2007. During his Master's education he already started working as a Software Engineer. Until 2011 he worked at *DEVENTit B.V.*, an independent software vendor in The Netherlands. He then joined AFAS Software B.V. as a Software Architect and became a Lead Software Architect in 2013. He is part of the team responsible for the development of AFAS Focus, an in-house low-code platform. In 2015 he joined the research project *AMUSE*, a collaboration between Universiteit Utrecht, Vrije Universiteit Amsterdam, and AFAS Software, as a PhD candidate. He finished his PhD research in 2022.

Michiel's main research interests are in low-code platforms, event sourced systems, and the microservice architecture style. Next to that he enjoys long runs, training in the local CrossFit box, and spending time with his wife and two sons.

SIKS Dissertation Series

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground

22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks

48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch

27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
02 Felix Mannhardt (TUE), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process

06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TUE), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TUE), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources

04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations

29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OUN), Learning visually grounded and multilingual representations

---

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TUE), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OUN), Towards a software architecture for reusable game components
08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
10 Alifah Syamsiyah (TUE), In-database Preprocessing for Process Mining
11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
12 Ward van Breda (VU), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
15 Konstantinos Georgiadis (OUN), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling

17 Daniele Di Mitri (OUN), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VU), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VU), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nobrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TUE), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatieve, Creatiefst
31 Gongjin Lan (VU), Learning better – From Baby to Better
32 Jason Rhuggenaath (TUE), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TUE), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (MU), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

---

2021 01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
02 Rijk Mercuur (TUD), Simulating Human Routines:Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UVA), Modeling Users Interacting with Smart Devices
04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems

06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Childrens Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), BIMODAL EMOTION RECOGNITION FROM AUDIO-VISUAL CUES
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
19 Roberto Verdecchia (VU), Architectural Technical Debt: Identification and Management
20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
22 Sihang Qiu (TUD), Conversational Crowdsourcing
23 Hugo Manuel Proença (LIACS), Robust rules for prediction and description
24 Kaijie Zhu (TUE), On Efficient Temporal Subgraph Query Processing
25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
26 Benno Kruit (CWI & VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs

---

2022 1 Judith van Stegeren (UT), Flavor text generation for role-playing video games
2 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey

3 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
4 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
5 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
6 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
7 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
8 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
9 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
14 Michiel Overeem (UU), Evolution of Low-Code Platforms

Errata

As with any sufficient large document, errors are bound to show up after printing. I've decided to update the PDF version, available through my website `https://www.movereem.nl/`. This chapter lists the corrections made.

2022-07-29

- Chapter 5 was wrongly titles "API Management Maturity of L**D**CPs".
- In Figure 4.2 the latest version was wrongly labelled "v0.6".