The Journal of Systems & Software 178 (2021) 110970

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In practice

An empirical characterization of event sourced systems and their schema evolution — Lessons from industry☆,☆☆

Michiel Overeem [a,b,*], Marten Spoor [a], Slinger Jansen [b,c], Sjaak Brinkkemper [b]

[a] AFAS Software, Inspiratielaan 1, Leusden, The Netherlands
[b] Utrecht University, Princetonplein 5, Utrecht, The Netherlands
[c] School of Engineering Science, LUT University, Finland

ARTICLE INFO

Article history:
Received 7 March 2020
Received in revised form 1 February 2021
Accepted 1 April 2021
Available online 9 April 2021

Keywords:
Event sourcing
CQRS
Event-driven architecture
Schema evolution
Software architecture patterns
Grounded theory

ABSTRACT

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.

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 built these systems using knowledge acquired by other engineers. Influential books such as *Patterns of Enterprise Application Architecture* by Fowler (2002) and *Enterprise Integration Patterns* by Hohpe and Woolf (2004) demonstrate the impact of pattern descriptions on software engineering. Architectural patterns are part of the trend of knowledge based architecture design; Li et al. (2013). Kassab et al. (2018), Taibi et al. (2018), and Harrison et al. (2007) 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 (Avery and Reta, 2017), and Walmart's Jet.com (Gorodinski, 2017), with the goal of creating scalable and reliable critical systems. Event sourcing is informally described by Fowler (2005) 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 and Reta (2017) as rationale for using event sourcing. Debski et al. (2017) and Erb and Hauck (2016) show how event sourcing can be applied to achieve scalable, reactive systems. Kabbedijk et al. (2012) describes event sourcing as a sub pattern of Command Query Responsibility Segregation (CQRS) in his work on the improved variability and scalability of systems applying CQRS.

The events in event sourcing, as opposed to general event-driven architectures (EDAs) (Fowler, 2017), 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

☆ This research was supported by the NWO AMUSE project (628.006.001): a collaboration between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software in the Netherlands.

☆☆ Editor: Marcos Kalinowski.

* Corresponding author at: AFAS Software, Inspiratielaan 1, Leusden, The Netherlands.

E-mail addresses: michiel.overeem@afas.nl (M. Overeem), marten.spoor@afas.nl (M. Spoor), slinger.jansen@uu.nl (S. Jansen), s.brinkkemper@uu.nl (S. Brinkkemper).

https://doi.org/10.1016/j.jss.2021.110970

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 be also used as a design approach. Domain-Driven Design (DDD), as described by Evans (2003), advocates events as a design tool for the process flow of a software system. Brandolini (2018) 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.

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 (Overeem et al., 2017), that 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 complexity. 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 et al. (2015), 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 (2016) 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 et al. (2011) 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. That 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 2 explains how we applied GT to form a basis for 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 contrasts ESSs with other existing architectural patterns, such as EDAs and blockchain, and shows that ESSs are insufficiently described in existing literature.
- Section 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 5 provides a thorough description of ESSs based on the findings of the interviews, presenting the pattern itself including its relation to CQRS. It also reflects on the role of the (implicit) schema present in ESSs.
- Section 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 7 focuses on of 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 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.

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 et al. (2011) 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 to gathering architecture knowledge.

GT is a common research strategy in software engineering research and induces theory from empirically collected material, such as through interview or case studies. For instance, Hoda et al. (2012) explore the practices of self-organizing agile teams using GT. Greiler et al. (2012) apply GT to improve the understanding of testing practices for plug-in systems. Tamburri and Kazman (2018) recover software architectures by applying GT. Last, Santos et al. (2019) 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 colored 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 (Charmaz, 1996) 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.

2.1. Research questions and motivation

The motivation of our research is formed by five years experience in the development of an event sourced system and earlier research on schema evolution in ESSs (Overeem et al., 2017). 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 to 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 (Overeem et al., 2017) 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 (Stol et al., 2015; Charmaz, 1996). 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 (Overeem et al., 2017) ourselves. We made our previous work part of the initial data set and also included the works of Fowler, e.g. Fowler (2017). 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.

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 channels. 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 (Wohlin, 2014): we explicitly asked every 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 1 summarizes the engineers, including their role, years of experience with ESSs, 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 4. We will refer to the engineers by the number given to them in Table 1.

Table 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.

	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

2.3. Interview techniques and GT

Each interview took 30-90 min, either in person or via video conference. The protocol presented in Appendix was created using the guidelines of Castillo-Montoya (2016). 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 (Overeem et al., 2021).

The protocol was followed freely: the answers given by the engineers guided the interviews. The four protocol parts remained stable over 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 at the last interview is presented in the appendix. The first part of the interview 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.

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 of 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 before 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 overall 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. (1995) 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 8 discusses the relation between our concepts and the guidelines of Gamma et al. (1995). The categories, concepts, and codes found during the interviews are presented in Sections 4–7. Tables 2–7 summarizes the results.

The interview protocol, the anonymized transcripts of the interviews, and the classification codes with links to the interviews are made available as data package (Overeem et al., 2021).

3. Background

The foundational idea of event sourcing is the domain event as described by Evans (2015). His seminal book on Domain-Driven Design (DDD), however, does not mention the pattern. Vernon (2013) describes event sourcing only briefly in his book on the implementation of various DDD patterns. Young (2017), 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, 2010), a pattern strongly related to event sourcing. Recent academic literature (Erb, 2019; Zhong et al., 2019) 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 and Reuter (1992) describe how transaction logs can be used to replicate 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 (2017) 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 (Vassiliadis, 2009). 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 et al. (1995), Time series, as described by Dreyer et al. (1994), is another data model that deals with the temporal aspects of data. 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 (Wu et al., 2006), 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 (2011). 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 as the event calculus formalism. Event calculus, as described by Sadri and Kowalski (1995), 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 de Murillas et al. (2015) 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. (2018) describes 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 build using a blockchain solution. However, the distribution and consensus features offered by blockchain do not improve the goals targeted by event sourcing.

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 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

Table 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.

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

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)

the interviews. We did not encounter systems using event sourcing for IoT systems, or other stream processing systems. This is in accordance with 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.

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.

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 essentially destroying your older copy of the data. And who told you are allowed to delete data?"* We classified this group of rationale with the category **audit** (Van Der Aalst et al., 2010) (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 debug scenarios for instance) is given as reason too (EmailSys).

The second category is **flexibility** (Lassing et al., 1999) (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 5 explains how this flexibility is achieved through the implementation of different *projections* and *projectors*.

The third category is **complexity** (Biemans et al., 2001) (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, and only the rationale for three of the 19 systems, is **trending** (Clements, 1997) (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 to 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.

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 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 from the characteristics that are listed in Table 2.

Event sourcing is a pattern that stores every state change, immutability is thus at the core of the pattern. Helland (2015) 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 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 (2003) 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. An ESS that contains events not representing real world business domain events will undergo more changes to the software according to E11. 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 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 (Dragoni et al., 2017) 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 4 summarizes the details that were reported per discussed system. The systems have a size ranging from smaller than three gigabytes, up to 250 GB (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 the 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 shows that none of the reported systems passes the 500 event types, but is rather spread out between 20 and 450 types. Overall, Table 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. And the event store size shows that event sourcing can be used for both small and large systems.

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.

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 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 (Avery and Reta, 2017) that represents a real world business event is given in JSON format:

Table 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.

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 do not 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

```
{ "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 (2006): "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 (The Apache Software Foundation, 2019), or Protobuf (Google Inc., 2019). 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*

$$s = \langle (e_1, 1), (e_2, 2), \ldots, (e_n, n) \rangle$$

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 (2019) implements the *append* function with a version that atomically appends multiple events to the stream.

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 of 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 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 cannot 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 (Date, 2003): 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 depends 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 by for instance by E8, E12, E23), an analysis to report 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 lays 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 reason for choosing event sourcing (Section 6) is in large part caused by the *project* function.

The *project* function also poses a risk for the performance of the system, a challenge we discussed in Section 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 rather plan the rebuilding in weekends, instead of investing developer effort for 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 opportunistic: 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* (Hutton, 1999), a recursion operator that works on lists, as stated by Meißner et al. (2018). 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 (2018).

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 an 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. (1995). The system first builds a projection, and then validates the command using the *accept* function. Validation of the command can result in either an 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.

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 (Fowler, 2013), 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 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 (The Apache Software Foundation, 2019). 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 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, such as the fact that when a specific stream contains a certain event another stream should exist is also present in the event schema, can also be specified in the event store schema.

An explicit implementation of event stream schemas or event store schemas was not encountered during the interviews.

5.4. Event sourced systems based on CQRS

As we have seen in Section 4, every ESS under study also applies CQRS. CQRS was introduced by Young (2010) and Dahan (2009), and the goal of this pattern is to separate actions that change data (those are called commands) from requests that ask for data (called 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 Fig. 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 (2003)) 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 will be 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 were multiple aggregates use the same stream. This variation is used to share generic behavior 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, pre-building 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. 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 (2009) 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 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 6.

Four engineers explain how there 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 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.

Table 5
The concepts and codes extracted from the interviews related to the implementation of CQRS based ESSs.

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)

6. Challenges faced in applying event sourcing

A pattern description without discussing the consequences is incomplete, and would lead engineers astray. While Section 4 discusses the positive consequences that engineers experienced, they also discussed the negatives in the interviews. In this Section we discus 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 Sections 5 and 7. The summary of mentioned challenges by engineers is listed in Table 6.

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 thirteen of the 25 engineers, thinking in events and state transfers is completely different from thinking in current state and database transactions. Section 5 proposes a description that improves the understanding, and support the teaching of event sourcing and event sourced systems (ESSs).

However, 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

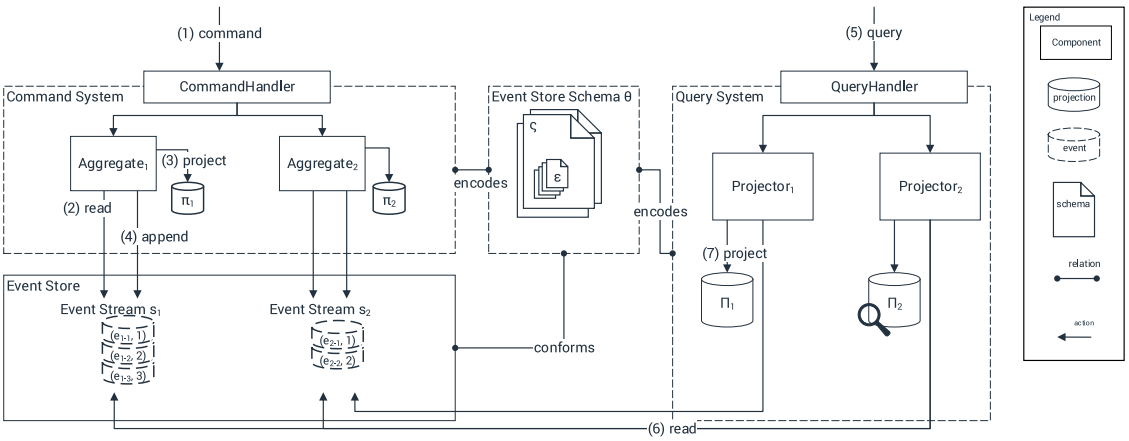

Fig. 1. An event sourced system based on CQRS. The event store conforms to the schema θ, which is encoded by the command and query systems. The command system validates commands using the events. These same events are read by the query system to build the projections, which are used to respond to queries.

Table 6
The challenges faced by the practitioners while implementing ESSs.

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 7

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. However, 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 eventual 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 misses 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 Overeem et al. (2017) and the work of Young (2017) underline this. During the interviews we identified five fundamental techniques for schema evolution in ESSs, which are described in Section 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 (2019) has started to offer commercial support for ESSs, similar to what Event Store (2019) does.

How can Projections be Optimized? — Projections, as discussed in Section 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 rebuilt those projections that are required to, 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 the personal information. If that information is no longer present (because of removal requests), default values are supplied.

7. Schema evolution in event sourced systems

A challenge discussed by multiple engineers is evolution of event sourced systems (ESSs) (as stated in Section 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 5.

We encountered two reasons why event schema evolution in ESSs is difficult. First of all, the implicit schema (as described by Fowler (2013)) makes evolution in ESSs difficult. Solutions as proposed by Meurice et al. (2016) and Maule et al. (2008) 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 than 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 4) are not always requirements. The different degrees of immutability, as shown in Table 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 always 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 Overeem et al. (2017) 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 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 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 techniques 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 (2011) as the *tolerant reader* pattern. Serialization formats such as Protobuf by Google Inc. (2019) and AVRO by The Apache Software Foundation (2019) 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. (2013). 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 $\mathit{conforms}(es, \theta')$ holds. However, events already stored in *es* still *conforms* 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 to *message translators* as described by Hohpe and Woolf (2004).

FINDINGS Twelve engineers use upcasters, claiming benefits as no domain pollution, immutability of events, and simplicity of implementation. One of the stated liabilities is an 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 of 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

Table 7
Benefits and liabilities of event sourcing evolution techniques.

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)

Store, 2019) and AxonDB (AxonIQ, 2019), deliberately do not offer these operations.

RELATED WORK This technique is similar to migration scripts for relational databases. Scherzinger et al. (2013) and Saur et al. (2016) 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 and Fowler (2012).

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 (2017) describes this technique as *copy and replace*. The *parallel universe* of IMAGO, as described by Dumitraş and Narasimhan (2009), is similar to this technique. QuantumDB, created by de Jong and van Deursen (2015), 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 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 to not 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 interview are

- The application of *upcasters*, with *copy-transform* to clean up the upcasters when there are to 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 5.

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 Axon Framework (2019), Event Store (2019), NEventStore Dev team (2019), Prooph Components (2019), 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, 2006), 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. (2011)) 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 emerged from the GT, and not according to a specific pattern description format. We did, however, use the examples of Gamma et al. (1995) to evaluate the completeness of our pattern description.

Gamma et al. states 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 4. The description of the pattern, the solution, is covered in Section 5. Finally, the consequences, are split over two sections: Section 4 covers the positive consequences by linking them to the problems that are solved. Section 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 not different in ESSs. Section 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 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 6) and in the challenge discussion about evolving event sourced systems (Section 7).

The number of interviews does not allow use to generalize over 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 weight 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.

9. Threats to validity

Both Golfasni (2003) and Onwuegbuzie and Leech (2007) 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 are willing and able to discuss the pattern at length. It is for instance remarkable that they all combine CQRS with event sourcing. Table 1 shows a diverse variety of experiences, and Table 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 full-time involved in the development of the ESS. The exploratory questions (Appendix) focus on topics that can sufficiently be 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 (Appendix) has 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 lead to many interactions with practitioners in gatherings, conferences, and on-line. These interactions have served as an informal triangulation that support the findings presented in this article.

As a constructivist GT approach (Charmaz, 1996) 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 us to 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 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 cannot 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, while we do not claim generalizability to other domains, we do believe that those domains can be inspired by our findings in designing event sourced systems. Also, the common occurrence of all event sourcing evolution techniques in Table 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.

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 of the improved scalability that ESSs systems provide.

These benefits give enough reason 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 4 and especially in Tables 2 and 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 all of these systems have benefited from event sourcing, according to their engineers. 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 Table 2 substantiate that event sourcing can be applied in a diversity of domains, and technologies.

How can event sourced systems be defined? Section 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 the topics to define through the interviews.

How can event sourced data structures be evolved? Five event schema evolution techniques are discussed in Section 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, as summarized in Table 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 found 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 6 and summarized in Table 6. We address the steep learning curve in Section 5 by giving definitions and operations that can be used in discussing and teaching of ESSs. Evolution is discussed in detail in Section 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 2 and 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.

CRediT authorship contribution statement

Michiel Overeem: Conceptualization, Writing - original draft, Data curation, Investigation. **Marten Spoor:** Data curation, Validation, Writing - review & editing. **Slinger Jansen:** Validation, Supervision, Writing - review & editing. **Sjaak Brinkkemper:** Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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.

Appendix. 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 performance 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?

References

Adolph, S., Hall, W., Kruchten, P., 2011. Using grounded theory to study the experience of software development. Empir. Softw. Eng. 16 (4), 487–513. http://dx.doi.org/10.1007/s10664-010-9152-6.

Anh, D.T.T., Zhang, M., Ooi, B.C., Chen, G., 2018. Untangling blockchain: A data processing view of blockchain systems. IEEE Trans. Knowl. Data Eng. 4347 (c), 1–20. http://dx.doi.org/10.1109/TKDE.2017.2781227, arXiv:1708.05665.

Avery, P., Reta, R., 2017. Scaling event sourcing for netflix downloads. https://www.infoq.com/presentations/netflix-scale-event-sourcing.

Axon Framework, 2019. Reference guide axon framework reference guide - Event upcasting. https://docs.axoniq.io/reference-guide/configuring-infrastructure-components/event-processing/event-bus-and-event-store.

AxonIQ, 2019. AxonDB. https://axoniq.io/product-overview/axondb.

Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., Subramanian, M., 2013. Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure. Microsoft patterns & practices.

Biemans, F.P., Lankhorst, M.M., Teeuw, W.B., Van De Watering, R.G., 2001. Dealing with the complexity of business systems architecting. Syst. Eng. 4 (2), 118–133. http://dx.doi.org/10.1002/sys.1010.

Brandolini, A., 2018. Introducing Event Storming. Leanpub.

Castillo-Montoya, M., 2016. Preparing for interview research: The interview protocol refinement framework. Qualitative Rep. 21 (5), 811–831.

Charmaz, K., 1996. The search for meanings - Grounded theory. Rethinking Methods Psychol. 27–49. http://dx.doi.org/10.1016/B978-0-08-044894-7.01581-5.

Clements, P.C., 19997. 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, pp. 2–9.

Dahan, U., 2009. Clarified CQRS. http://www.udidahan.com/2009/12/0.

Daigneau, R., 2011. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and Restful Web Services. Addison-Wesley.

Date, C.J., 2003. An Introduction to Database Systems, eigth ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Debski, A., Szczepanik, B., Malawski, M., Spahr, S., Muthig, D., 2017. In search for a scalable & reactive architecture of a cloud application: CQRS and event sourcing case study. IEEE Softw. PP (99), 1. http://dx.doi.org/10.1109/MS.2017.265095722.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., 2017. Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering. Springer, pp. 195–216, arXiv:arXiv:1606.04036v4.

Dreyer, W., Dittrich, A.K., Schmidt, D., 1994. Research perspectives for time series management systems. ACM SIGMOD Record 23 (1), 10–15. http://dx.doi.org/10.1145/181550.181553.

Dumitraş, T., Narasimhan, P., 2009. Why Do Upgrades Fail and What Can We Do About It? Toward Dependable, Online Upgrades in Enterprise System. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5896 LNCS, pp. 349–372. http://dx.doi.org/10.1007/978-3-642-10445-9_18.

Erb, B., 2019. Distributed Computing on Event-Sourced Graphs (Ph.D. thesis).

Erb, B., Hauck, F.J., 2016. 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, pp. 4:1–4:5.

Evans, E., 2003. Domain-Driven Design. Addison-Wesley Professional.

Evans, E., 2015. Domain-Driven Design Reference. Eric Evans, http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf.

Event Store, L., 2019. Event store. https://eventstore.org/.

Flyvbjerg, B., 2006. Five misunderstandings about case-study research. Qualitative Inquiry 12 (2), 219–245. http://dx.doi.org/10.1177/1077800405284363, arXiv:1304.1186.

Fowler, M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley.

Fowler, M., 2005. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html.

Fowler, M., 2013. Schemaless data structures. http://martinfowler.com/articles/schemaless/.

Fowler, M., 2017. What do you mean by "Event-Driven"?. https://martinfowler.com/articles/201701-event-driven.html.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Golfasni, N., 2003. Understanding reliability and validity in qualitative research. Qualitative Report 8 (4), 597–607, http://www.news-medical.net/health/Thalassemia-Prevalence.aspx.

Google Inc., 2019. Protocol buffers. https://github.com/google/protobuf.

Gorodinski, L., 2017. Scaling event-sourcing at jet. https://medium.com/@eulerfx/scaling-event-sourcing-at-jet-9c873cac33b8.

Gray, J., Reuter, A., 1992. Transaction Processing: Concepts and Techniques. Elsevier.

Greiler, M., Van Deursen, A., Storey, M.A., 2012. Test confessions: A study of testing practices for plug-in systems. In: Proceedings - International Conference on Software Engineering, pp. 244–254. http://dx.doi.org/10.1109/ICSE.2012.6227189.

Harrison, N.B., Avgeriou, P., Zdun, U., 2007. Using patterns to capture architectural decisions. IEEE Softw. 24 (4), 38–45. http://dx.doi.org/10.1109/MS.2007.124.

Helland, P., 2015. Immutability changes everything. Commun. ACM 59 (1), 64–70. http://dx.doi.org/10.1145/2844112, arXiv:arXiv:1011.1669v3.

Hoda, R., Noble, J., Marshall, S., 2012. Developing a grounded theory to explain the practices of self-organizing Agile teams. Empir. Softw. Eng. 17 (6), 609–639. http://dx.doi.org/10.1007/s10664-011-9161-0.

Hohpe, G., Woolf, B., 2004. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional.

Hutton, G., 1999. A tutorial on the universality and expressiveness of fold. J. Funct. Programming 9 (4), 355–372. http://dx.doi.org/10.1017/S0956796899003500.

Jagadish, H., Mumick, I.S., Silberschatz, A., 1995. View maintenance issues for the chronicle data model. In: Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. ACM, pp. 113–124. http://dx.doi.org/10.1145/212433.220201.

de Jong, M., van Deursen, A., 2015. Continuous deployment and schema evolution in SQL databases. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering, pp. 16–19, http://dx.doi.org/10.1109/RELENG.2015.14.

Kabbedijk, J., Jansen, S., Brinkkemper, S., 2012. 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, p. 2. http://dx.doi.org/10.1145/0000000.0000000.

Kassab, M., Mazzara, M., Lee, J.Y., Succi, G., 2018. Software architectural patterns in practice: an empirical study. Innov. Syst. Softw. Eng. 14 (4), 263–271. http://dx.doi.org/10.1007/s11334-018-0319-4.

Kleppmann, M., 2017. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc..

Lassing, N., Rijsenbrij, D., Van Wet, H., 1999. Towards a broader view on software architecture analysis of flexibility. In: Proceedings - 6th Asia Pacific Software Engineering Conference, APSEC 1999, pp. 238–245. http://dx.doi.org/10.1109/APSEC.1999.809608.

Li, Z., Liang, P., Avgeriou, P., 2013. Application of knowledge-based approaches in software architecture: A systematic mapping study. Inf. Softw. Technol. 55 (5), 777–794. http://dx.doi.org/10.1016/j.infsof.2012.11.005.

Luckham, D.C., 2011. Event Processing for Business: Organizing the Real-Time Enterprise. John Wiley & Sons.

Maule, A., Emmerich, W., Rosenblum, D.S., 2008. Impact analysis of database schema changes. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. ACM, pp. 451–460. http://dx.doi.org/10.1145/1368088.1368150.

Meißner, D., Erb, B., Kargl, F., Tichy, M., 2018. 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, pp. 76–87.

Meurice, L., Nagy, C., Cleve, A., 2016. Detecting and preventing program inconsistencies under database schema evolution. In: 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS, pp. 262–273. http://dx.doi.org/10.1109/QRS.2016.38.

Michelson, B.M., 2006. Event-Driven Architecture Overview. Technical report, Patricia Seybold Group, pp. 210–1571.

de Murillas, E.G.L., van Der Aalst, W.M., Reijers, H.A., 2015. Process mining on databases: Unearthing historical data from redo logs. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9253, pp. 367–385. http://dx.doi.org/10.1007/978-3-319-23063-4_25.

Musil, J., Musil, A., Weyns, D., Biffl, S., 2015. An architecture pattern for collective intelligence systems. In: 12th Working IEEE/IFIP Conference on Software Architecture, pp. 21–30. http://dx.doi.org/10.1145/2855321.2855342.

NEventStore Dev team, 2019. NEventStore. http://neventstore.org.

Onwuegbuzie, A.J., Leech, N.L., 2007. Validity and qualitative research: An oxymoron? Quality Quantity 41 (2), 233–249. http://dx.doi.org/10.1007/s11135-006-9000-3.

Overeem, M., Spoor, M., Jansen, S., 2017. The dark side of event sourcing: Managing Data conversion. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, pp. 193–204.

Overeem, M., Spoor, M., Jansen, S., Brinkkemper, S., 2021. An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry - Accompanying Anonymized Transcripts. Mendeley Data, http://dx.doi.org/10.17632/dgbxyn7yw3.1.

Prooph Components, 2019. Prooph. http://getprooph.org/.

Sadalage, P.J., Fowler, M., 2012. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley.

Sadri, F., Kowalski, R., 1995. Variants of the event calculus Fariba Sadri and Robert Kowalski abstract. In: Proceedings of the Twelfth International Conference on Logic Programming, pp. 67–81.

Santos, J.C.S., Sejfia, A., Corrello, T., Gadenkanahalli, S., Mirakhorli, M., 2019. Achilles' heel of plug-and-Play software architectures: a grounded theory based approach. In: ESEC/FSE '19, Tallinn, Estonia, pp. 671–682. http://dx.doi.org/10.1145/3338906.3338969.

Saur, K., Dumitraş, T., Hicks, M., 2016. Evolving NoSQL databases without downtime. In: IEEE International Conference on Software Maintenance and Evolution, ICSME, pp. 166–176, arXiv:1506.08800.

Scherzinger, S., Klettke, M., Störl, U., 2013. Managing schema evolution in nosql data stores. In: Green, T.J., Schmitt, A. (Eds.), Proceedings of the 14th International Symposium on Database Programming Languages {(DBPL} 2013), August 30, 2013. Riva Del Garda, Trento, Italy, http://arxiv.org/abs/1308.0514.

Sein, M.K., Henfridsson, O., Purao, S., Rossi, M., Lindgren, R., 2011. Action design research. MIS Quarterly Manag. Inform. Syst. 35 (1), 37–56. http://dx.doi.org/10.2307/23043488.

Slotos, T., 2016. The star pattern - Representing domain concepts in a uniform way. In: Proceedings of the 21st European Conference on Pattern Languages of Programs, pp. 8. http://dx.doi.org/10.1145/3011784.3011792.

Stol, K.-J., Ralph, P., Fitzgerald, B., 2015. grounded theory in software engineering research : A critical review and guidelines. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, pp. 120–131. http://dx.doi.org/10.1145/2884781.2884833.

Taibi, D., Lenarduzzi, V., Pahl, C., 2018. Architectural patterns for microservices: A systematic mapping study. In: CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science, pp. 221–232. http://dx.doi.org/10.5220/0006798302210232.

Tamburri, D.A., Kazman, R., 2018. General methods for software architecture recovery: a potential approach and its evaluation. Empir. Softw. Eng. 23 (3), 1457–1489. http://dx.doi.org/10.1007/s10664-017-9543-z.

The Apache Software Foundation, 2019. Apache avro. http://avro.apache.org/.

Van Der Aalst, W.M., Van Hee, K.M., Van Der Werf, J.M., Verdonk, M., 2010. Auditing 2.0: Using process mining to support tomorrow's auditor. Computer 43 (3), 90–93. http://dx.doi.org/10.1109/MC.2010.61.

Vassiliadis, P., 2009. A survey of extract – transform – load technology. Intl. J. Data Warehousing Mining 5 (3), 1–27. http://dx.doi.org/10.4018/jdwm.2009070101.

Vernon, V., 2013. Implementing Domain-Driven Design. Addison-Wesley.

Vogels, W., 2009. Eventually consistent. Commun. ACM 52 (1), 40–44.

Wlaschin, S., 2018. Domain Modeling Made Functional. The Pragmatic Bookshelf.

Wohlin, C., 2014. 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, pp. 1–10. http://dx.doi.org/10.1145/2601248.2601268. http://dl.acm.org/citation.cfm?doid=2601248.2601268.

Wu, E., Diao, Y., Rizvi, S., 2006. High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data - SIGMOD '06, pp. 407. http://dx.doi.org/10.1145/1142473.1142520.

Young, G., 2010. CQRS and Event Sourcing. p. 56, http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing.

Young, G., 2017. Versioning in an Event Sourced System. Leanpub.

Zhong, Y., Li, W., Wang, J., 2019. Using event sourcing and CQRS to build a high performance point trading system. ACM Int. Conf. Proc. Ser. 16–19. http://dx.doi.org/10.1145/3317614.3317632.

Michiel Overeem is a Lead Software Architect at AFAS Software and part of the team that is responsible for their future ERP Cloud platform. As a Ph.D. candidate with Utrecht University he conducts research on the upgrading of model-driven, cloud-based software.

Marten Spoor is a Software Architect at AFAS Software and part of the team that is responsible for their future ERP Cloud platform.

Slinger Jansen is an assistant professor at the Department of Information and Computer Science at Utrecht University. He is one of the leading researchers in the domain of software ecosystems and co-founders of the International Conference on Software Business and International Workshop on Software Ecosystems. He is lead editor of the book "Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry" and of several others. Besides his academic endeavors he actively supports new enterprises and sits on the boards of advisors of several startups.

Sjaak Brinkkemper is full professor of organization and information at the Department of Information and Computing Sciences of the Utrecht University, the Netherlands. He leads a group of about twenty researchers specialized in product software development and entrepreneurship. In essence, he studies the implications of market conditions on software development methods and processes. He received the best paper awards for several papers, he has given keynote talks at several conferences. He has been awarded the IFIP Silver Core Medal. He is member of the steering committee of CAiSE, ECIS, ICSOB and IWSPM. He is co-founder of the International Software Product Management Board.