# The Dark Side of Event Sourcing: Managing Data Conversion

Michiel Overeem[1], Marten Spoor[1], and Slinger Jansen[2]

[1]{m.overeem, m.spoor}@afas.nl, Department Architecture and Innovation, AFAS Software, The Netherlands
[2]slinger.jansen@uu.nl, Department of Information and Computing Sciences, Utrecht University, The Netherlands

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

***Index Terms*—Event Sourcing, CQRS, Event Driven Architecture, Schema Evolution, Software Evolution, Schema Versioning, Deployment Strategy, Data Transformation, Data Conversion**

## I. Introduction

Applications that do not evolve in response to changing requirements or changing technology become less useful, as Lehman [1] in his law of *continuing change* stated many years ago. Neamtiu and Dumitras [2] shows that this is a reality for modern cloud systems as many of them update more than once a week. Chen [3] describes how they applied continuous delivery on multiple projects to achieve shorter time to market, and an improved productivity and efficiency. Several technical challenges including seamless upgrades are identified by Claps et al. [4]. The fast pace of evolution and deployment of cloud systems conflicts with the requirement to be available always and support uninterrupted work. For modern cloud systems to support the fast pace of evolution, upgrade strategies that are fast, efficient, and seamless have to be designed and implemented.

One of the architectural patterns that in recent years emerged in the development of cloud systems is Command Query Responsibility Segregation (CQRS). The pattern was introduced by Young [5] and Dahan [6], and the goal of the pattern is to handle actions that change data (those are called commands) in different parts in the system than requests that ask for data (called queries). By separating the command-side (the part that validates and accepts changes) from the query-side (the part that answers queries), the system can optimize the two parts for their very different tasks.

Young [7] describes CQRS as a stepping stone for event sourcing. Event sourcing is a data storage model that does not store the current (or last) state, but all changes leading up to the current state. Fowler [8] explains event sourcing by comparing it to an audit trail: every data change is stored without removing or changing earlier events. The events stored in an event store are stored as schema-less data, because the different events often do not share properties. A store with an explicit schema would make it more difficult to append events in the store to a single stream. Data in schema-less stores is not without schema, but the schema is implicit: the application assumes a certain schema. This makes the problem of schema evolution and data conversion more difficult as observed by Scherzinger et al. [9]. Schema-less data is more difficult to evolve as the store is unaware of structure and thus cannot offer tools to transform the data into a new structure. Relational data stores that have explicit knowledge of the structure of the data can use the standardized *data definition language* (DDL) to upgrade the schema and convert the data. Another problem in the evolution of event sourced systems is the amount of data that is stored, not only the current state, but also every change leading up to that state. This huge amount of data makes the problem of performing a seamless upgrade even more important: upgrades may need more time, but they are required to be imperceptible.

The frequency of schema changes is researched by Qiu et al. [10]. Although the storage model is different and the architectural pattern is relatively new there is no indication that (implicit) schema changes in event sourcing are less of a challenge. Recovery of the implicit schema does not solve the problem for event stores, it only helps in finding the right operations to transform to a new schema.

This is an AMUSE paper. See amuse-project.org for more information.

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

## II. Command Query Responsibility Segregation

The foundations of CQRS were laid by Meyer [11] in the Command-Query Separation (CQS) principle. He defined a command as "serving to modify objects" and a query is "to return information about objects", or informally worded "asking a question should not change the answer". Figure 1 shows the CQRS pattern: commands are accepted by the command-side and produce events which are processed by the query-side. The query-side projects these events into a form that is suitable for querying and presenting. The command-side and the query-side both have their own data store: the first store is used to maintain data that is used in validating requested changes, and the second store is used to retrieve data for displaying or reporting.

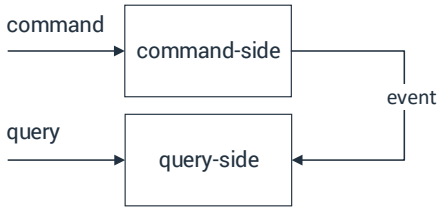


Figure 1. The architectural pattern CQRS.

The command-side communicates with the query-side through asynchronously sending events. These events are used by the query-side to build a view of the state that can be used to query and present data. By doing this asynchronously the query-side does not influence the performance of the command-side. However, this does lead to eventual consistency. This is a weaker form of consistency that Vogels [12] defines as "when no updates are made to the object, the object will eventually have the last updated value". The system guarantees that the query-side eventually will reflect the events produced in the command-side. However, there are no guarantees on how fast this will be done. A system with a large delay is unfeasible, because in that case queries will often return data that does not reflect the latest changes send to the command-side. There are difficulties introduced by eventual consistency, such as returning items to a client that in fact are already deleted through commands send to the command-side. The patterns to overcome this difficulty and others are out of scope for the current paper.

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

The loosely coupled nature of CQRS combined the benefits of the event sourcing approach makes it a fitting architectural pattern for cloud systems. Event sourcing itself is not tied exclusively to CQRS, the coupling based on events is similar to that in more general event-driven architectures, as described by Michelson [13]. The events in the event store are processed by the system to build the query-side or execute complex processes. The CQRS pattern and its sub-patterns are described in more detail by Kabbedijk et al. [14]. CQRS from a practitioners viewpoint is studied by Korkmaz [15] in order to gain better understanding of the benefits and challenges. Maddodi et al. [16] studies a CQRS system in the context of continuous performance testing.

## III. Related Work

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

**Data Conversion -** Two approaches to data conversion are defined by Jensen et al. [17]: *schema versioning* and *schema evolution*. Schema versioning is accommodated when a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version

interfaces. Schema evolution is accommodated when a database system facilitates the modification of the database schema without loss of existing data. Section V will show that both schema versioning and schema evolution are suitable techniques for event store upgrades.

The event store, used as a storage for the command-side of the CQRS system is schema-less, and in that respect similar to a NoSQL database as described by Scherzinger et al. [18] and Saur et al. [19]. Although the store is schema-less the data itself does have a schema, but it is implicit: the application, as defined by Fowler [20], assumes a certain schema without this schema being actually present in the store. Within relational stores the standardized DDL can be used to upgrade the schema and convert the data, a possibility missing in NoSQL stores. Scherzinger et al. [18] approach the implicit schema and lack of a DDL for NoSQL by proposing a new language that can be used to convert the data in a NoSQL store. Although this fills a gap in the standardization of NoSQL stores, without support in the stores the problem of data conversion in NoSQL stores remains. To aid the evolution of the data stored Saur et al. [19] describe an extension to one specific NoSQL database. This extension implements an approach that Sadalage and Fowler [21] describe as *incremental migration*: migrating data when it is accessed. While the research of Saur et al. [19] is similar to the research described in this paper, the solution is tied to a specific technology and not applicable in systems that use a similar data model, but with a different database technology.

Both Cleve et al. [22] and Qiu et al. [10] quantity schema changes occurring in the evolution of applications. Their work is aimed at relational models, and it is not clear how these results translate to event stores. Future studies need to be conducted before these results can be applied to event stores.

The impact of schema changes on application source code is studied by Meurice et al. [23] and Maule et al. [24]. However, the direction of the impact is different with schema-less stores and implicit schemas. The change originates in the application holding the implicit schema and impacts the data in the schema-less store.

**Application Deployment -** Blue-green deployment is an upgrade strategy that utilizes two slots to which different versions of an application can be deployed. One of the slots is active, while the other one is inactive. Upgrading is always done in the inactive slot, and the user is not hindered while upgrading. This strategy is followed by different authors. Callaghan [25] describes a tool written by Facebook to perform online (and zero downtime) upgrades on MySql in four phases: (1) copy the original database, (2) upgrade the copy to the new schema, (3) replay any changes happened on the original database during copy/build phase, and (4) finally switch active databases. This approach is very similar to the pattern described by Keller [26] who applied it in the migration of a legacy system. With IMAGO Dumitras and Narasimhan [27, 28] use blue-green deployment for their *parallel universe*: they reduce upgrade failures by isolating IMAGO the production system from the upgrade operations, and completing the upgrade as an atomic operation. QuantumDB, a tool created by de Jong and van Deursen [29], applies the expand-contract strategy (explained in Section VI) with blue-green deployment.

Hick and Hainaut [30] and Domínguez et al. [31] developed and used MeDEA: a tool that focuses on traceability of artifacts. MeDEA makes it possible to translate changes from a conceptual model of a relational database to schema changes in the actual database. Curino et al. [32, 33] worked on PRISM and PRISM++, a database administrator tool that calculates the SQL statements needed to upgrade a schema. While calculating those statements it can check for information preservation, backwards compatibility, and redundancy. These approaches solve the problem of analyzing schema changes and generating data conversion statements, something that is not part of the solution presented in this paper.

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

## IV. Event Store Upgrade Operations

An event store contains different event streams and events. An example is given: the event store of a *WebShop* application, shown in Figure 2. The two streams contain many events, but only two events per stream are shown as an example.

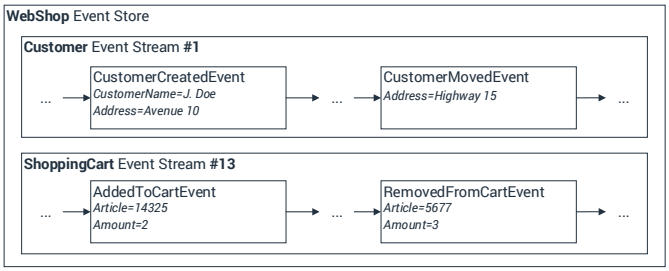


Figure 2. An example event store with different stream and event types.

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

Table I
THE EVENT STORE UPGRADE OPERATIONS, CATEGORIZED BY LEVEL AND COMPLEXITY.

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

An event store has a structure of three levels:

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

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

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

The event store upgrade operations are used to express how an event store version 1.0 can be transformed into version 2.0. These operations have the same purpose as the NoSQL schema evolution language proposed by Scherzinger et al. [18]: they give a common language to express the conversion of an event store. The full list of operations is shown in Table I. Two categorizations are applied: structure level and complexity. The operations on the store level are executed on one or multiple streams, the stream level operations convert one or more events within the same stream, and the event level operations convert a single event. The update of a stream is expressed by the stream level operations while the update of an event is expressed by one or more event level operations. Every level of operations is also categorized in basic and complex operations. Basic operations are seen as the foundational operations: they cannot be expressed by other operations. The complex operations can be built by combining several basic operations, making the categories with complex operations infinitely large.

The operations presented are agnostic of the business domain of the application and its functionality. The process of expressing the transformation in these operations should be done manually, because it should reflect the intent of the upgrade. Schema changes can be expressed by different sets of operations and these different sets have their own effects. An example is given: the *WebShop* application is upgraded to a new version, and part of the upgrade is a change in storing addresses. Figure 2 shows the old event definition: a single attribute for both street and number. In the new version, this should be stored in two separate attributes. This data conversion can be done in multiple ways and two possibilities are given:

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

Although both options transform the event store differently the two resulting versions of the *WebShop* application are functionally equivalent to the users of the system. However, the inner workings differ significantly. In the first solution the knowledge of the initial address property together with the values is removed. The conversion itself has changed the event

store, and now it appears that the events always contained the two separate attributes. The second solution retains the old addresses, and adds the split in two attributes as an event to this system. This conversion keeps the old events intact and does not remove information from the store. This example illustrates the need for requirements defined by stakeholders to guide the data conversion.

## V. Event Store Upgrade Techniques

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

**Multiple versions** - In this technique multiple versions of an event type are supported throughout the application. The event structure is extended with a version number as suggested by Betts et al. [34]. This version number can be read by all the event listeners, and they have to contain knowledge of the different versions in order to support them. In this technique the event store remains intact as old versions are never transformed. There are no extra write operations needed to convert the store.

**Upcasting** - Upcasting centralizes the update knowledge in an *upcaster*: a component that transforms an event before offering it to the application. Different than in the *multiple versions* technique is that the event listeners are not aware of the different versions of events. Because the upcaster changes the event the listeners only need to support the last version. This technique is suggested by both Betts et al. [34] and Axon Framework [35].

**Lazy transformation** - This technique also uses an *upcaster* to transform every event before offering it to the application, but the result of the transformation is also stored in the event store. The transformation is thus applied only once for every event, and on subsequent reads the transformation is no longer necessary. This technique is similar to the ones described by Sadalage and Fowler [21], Roddick [36], Tan and Katayama [37], and Scherzinger et al. [9].

**In place transformation** - A technique applied by many systems using a relational database. These systems convert the data by executing SQL statements such as `ALTER TABLE` (to alter the schema) and `UPDATE` (to alter the data). As described by Scherzinger et al. [9], NoSQL databases do not have such a possibility. In those cases a batch job is run that reads the data, transforms it, and writes the updated data back to the database. The documents in the database are updated by this job: adding, deleting, renaming properties, and transforming the values. This technique can be applied to event stores in the same manner.

**Copy and transformation** - This technique is similar to the one described by Callaghan [25] and Dumitras and Narasimhan [28]: it copies and transforms every event to a new store. In this technique the old event store stays intact, and a new store is created instead.

The event store upgrade techniques have their own strengths and weaknesses. To make this visible the techniques are judged on four quality characteristics from ISO/IEC 25010:2011 [38]: functional suitability, maintainability, performance efficiency, and reliability. The other four characteristics are regarded as not relevant for these upgrade techniques. Compatibility is a requirement for every upgrade system: it should be compatible with the overall system. End-users of the system will not interact with the upgrade system and thus is usability not relevant. The upgrade system is one of the components in the overall security, and therefore the security should not be different than in other components. Finally, portability is not regarded as a requirement for upgrade systems.

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

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

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

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

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

Table II
THE EVENT STORE TECHNIQUES COMPARED ON FOUR QUALITY CHARACTERISTICS.

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

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

## VI. APPLICATION AND DATA UPGRADE STRATEGIES

According to Humble and Farley [39] and Jansen et al. [40], deploying software involves three phases: *Prepare and manage*, *Installing*, and *Configuring*. In the first phase, the environment in which an application is deployed should be prepared and managed: both hardware and software dependencies should be in place. During the Installing-phase the application itself is deployed. The final phase, the Configuring-phase is used to configure the application and make it ready for use.

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

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

The simplest deployment strategy is to copy the new application onto the machine(s) replacing the older version. Brewer [41] refers to this approach as *fast reboot*. The time that it takes to bring down the application process, copy the new application, and starting the application process again is the downtime that is observed with this strategy. Its simplicity is its biggest selling point, but its biggest downside is that this strategy is not without downtime. Deployment strategies described by Pulkkinen [42] such as feature flagging, dark launching, and canary release are excluded from the list of discussed strategies, because they are specifically used to gain more knowledge about the users and/or (system) performance. Four strategies found in literature, suitable for upgrading an event sourced system, are discussed:

**Big flip** - This strategy, described by Brewer [41], uses request routing to route traffic to one half of the machines, while they other half is made available for the upgrade. The traffic is rerouted again when the first half is upgraded after which the second half can be upgraded. When all machines are upgraded the load balancer again can route the traffic to every machine. During the upgrade only half of the machines can be used to handle traffic.

**Rolling upgrade** - This strategy too uses some form of request routing to make sure that some machines do not receive requests. The machines in this strategy are upgraded in several upgrade groups defined by Dumitras et al. [43]. Because a small number of machines is being upgraded at a time, more machines are available to handle the traffic. However, the machines that are available are running mixed versions of the application: both those that are not yet upgraded and those that are already upgraded. This makes rolling upgrades complex, and the application should be able to handle these kinds of rolling upgrades.

**Blue-green** - Blue-green deployment is described by both Humble and Farley [39] and Fowler [44]. According to Humble [39] this is one of the most powerful techniques for managing releases. Every application is always deployed twice: a current version and either a previous version or a future version. One of the deployments is active at a given time, either the *green* slot or the *blue* slot. When the application is upgraded, the inactive slot is used to deploy the new version. Blue-green deployment can be done without downtime, as no traffic is going to the version that is upgraded. After the upgrade, the traffic can be rerouted to the upgraded slot, switching between blue and green. This strategy is similar to the *big flip* strategy, but reserves extra resources for the upgrade while the *big flip* strategy uses existing resources and thus limits the capacity during an upgrade.

Table III
COMBINATIONS OF TECHNIQUES AND STRATEGIES THAT RESULT IN ZERO DOWNTIME.

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

**Expand-Contract** - A strategy, also known as *parallel change*, described by Sato [45] in three phases. The first phase is the expand phase: an interface is created to support both the old and the new version. After that the old version(s) are (incrementally) updated to the new version in the migrate phase. Finally in the contract phase, the interface is changed so that it only supports the new phase This strategy is suitable for upgrading components that are used by other components. By first expanding the interface of the component depending components can be upgraded. When all the depending components use the new interface the old interfaces can be removed. This strategy is not applicable for application upgrades, however, it can be utilized in upgrading the database.

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

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

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

## VII. EVENT STORE UPGRADE FRAMEWORK

This section explains how the event store upgrade operation, techniques, and strategies form the event store upgrade framework that can be utilized in two distinct manners. Figure 3 shows the different event store upgrade operation, techniques, and strategies and their combinations.

The first row of Figure 3 shows the event store upgrade operations, the darker yellow identifies the category of operations that crosses event streams and cannot be executed run-time. The event store upgrade techniques are colored green, the darker elements identify schema evolution techniques, while the others are schema versioning techniques. The last two rows identify both application and data upgrade strategies. The arrows between single elements, or groups of elements, identify the valid combinations. The valid combinations are explained in more detail along with the utilization of the framework.

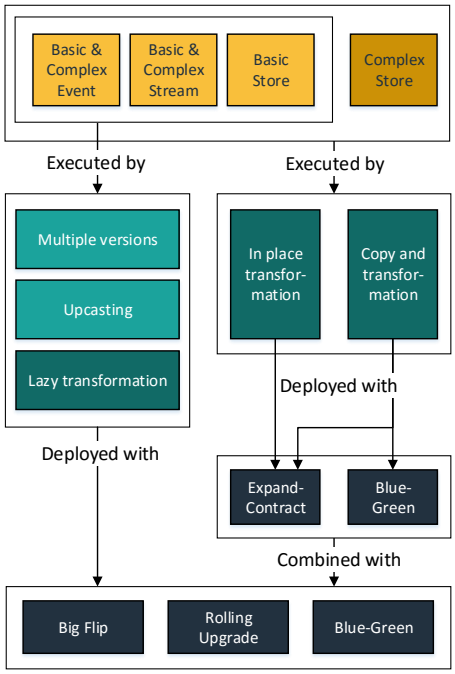


Figure 3. The Event Store Upgrade Framework.

The first utilization of the event store upgrade framework is a decision tree that supports the upfront design and implementation of an upgrade system for event sourced systems, presented in Figure 4. This tree shows the decisions that form the design and implementation of an upgrade system.

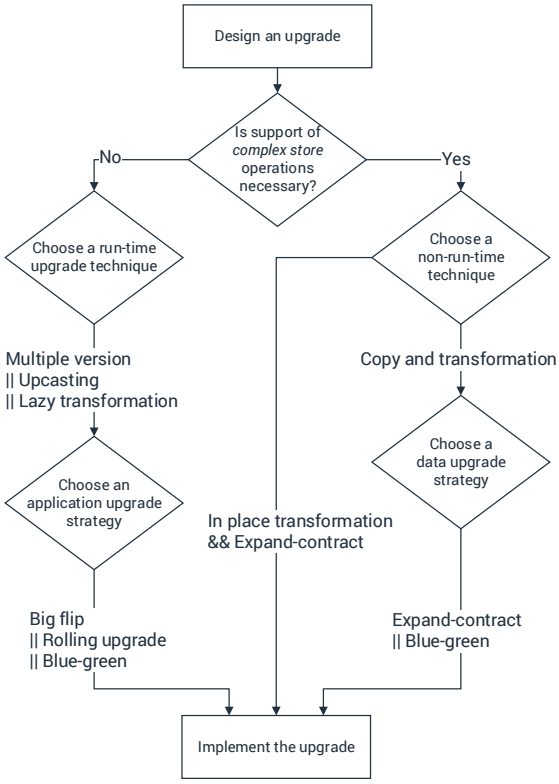


Figure 4. Decision Tree for the Design and Implementation of an Event Store Upgrade System.

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

The second utilization of the event store upgrade framework is a run-time decision making system. This system is implemented in the event store upgrade system, and is visualized in Figure 5. In this system the analysis of the event store upgrade operations that need to be executed is done at upgrade time. When the operations do not contain complex store operations, the system can apply the run-time technique in combination with the application upgrade strategy. If there are complex store operations the system can deploy the non-run-time technique with the data upgrade strategy, and then apply the application upgrade strategy. In this utilization, the choice for run-time technique, non-run-time technique, data upgrade strategy, and application upgrade strategy is made upfront. The system implements both a run-time and non-run-time technique that fits the requirements. The two techniques are completed with an implementation of a data upgrade and an application upgrade strategy. Having these implementations in the system allows for a fully automated upgrade system based on Figure 5.

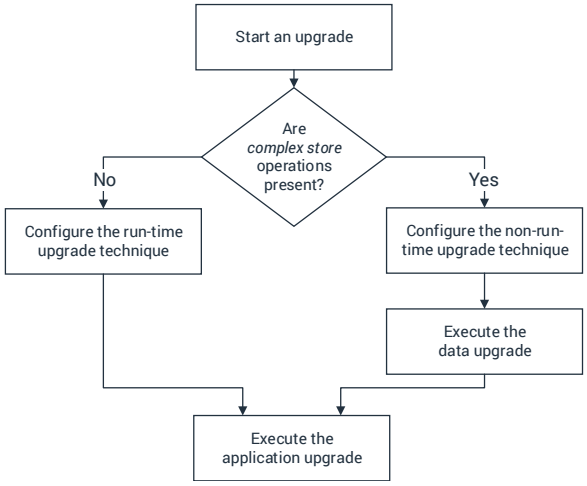


Figure 5. Decision Tree for an Automated Event Store Upgrade System.

## VIII. Evaluation

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

was longer than expected, because the query store needs to be rebuild as well.

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

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

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

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

**Frequency of Operations, and Business Requirements -** All interviewees agreed that the event store upgrade operations across the boundaries of event streams were not common. One of the interviewees stated "Complex event store operations, you will not see a lot. There is a exponential relation between the level and the times you encounter an operation." These operations cause the need for a data upgrade strategy, which is something that two interviewees found conflicting with the expected immutability of the event store. As a result of this discussion, the support of complex store operations became the first question in the event store upgrade framework. The interviewees explained that an event store upgrade system can be useful, even if it does not support these complex store operations, because they see other possibilities of solving these schema changes. The same holds for the operations that delete information from the store, and all interviewees suggested that deprecating or archiving of data is preferred over the actual deletion.

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

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

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

**Upfront Design and Prototyping -** The interviews show two sides to look at event sourced systems. Two out of three interviewees emphasized the importance of upfront design: by designing the event store, the streams, and the events with enough upfront thought, upgrades are less often necessary. One interviewee stated "Event sourcing needs a lot of upfront thought, which is hard to do with agile development." This line

[1]http://www.axonframework.org/

[2]https://github.com/pjvds/ncqrs

of thought is also seen in the application of *event storming*[3] in the design of event sourced systems. This design technique is applied to design the events in a system before implementation, and a good design is said to forestall some of the more complex event store upgrade operations, such as those on multiple event streams. The other interviewee stressed the importance of doing event store data upgrades to prevent the accumulation of conversion code, and thus found less value in defending technical immutability at all cost.

**Completeness and Usefulness -** The interviewees found the event store upgrade framework unanimously useful and complete. Interest in the end result was shown and encouragement was given to publish this material. One interviewee stated that "You are maybe the only one, which is having such an overview and also thought about edge cases, which I hope never to encounter."

## IX. Conclusion and Future Work

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

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

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

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

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

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

## Acknowledgment

**Acknowledgment** 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. The NEXT Platform is developed and maintained by AFAS Software. The authors also thank the three experts, Allard Buijze, Dennis Doomen, and Pieter Joost van de Sande, for their valuable experience and their willingness to contribute to this study.

## References

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] I. Neamtiu and T. Dumitras, "Cloud software upgrades: Challenges and opportunities," in *2011 International*

*Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2011, pp. 1–10.

[3] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[4] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.

[5] G. Young. (2010) CQRS and Event Sourcing. *visited on 2016-10-11*. [Online]. Available: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing

[6] U. Dahan. (2009) Clarified CQRS. *visited on 2016-10-11*. [Online]. Available: http://www.udidahan.com/2009/12/0

[7] G. Young. (2016) A Decade of DDD, CQRS, Event Sourcing - Domain-Driven Design Europe 2016. *visited on 2016-10-11*. [Online]. Available: https://www.youtube.com/watch?v=LDW0QWie21s

[8] M. Fowler. (2005) Event sourcing. *visited on 2016-10-11*. [Online]. Available: http://martinfowler.com/eaaDev/EventSourcing.html

[9] S. Scherzinger, M. Klettke, and U. Störl, "Cleager: Eager Schema Evolution in NoSQL Document Stores," in *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, ser. LNI, T. Seidl, N. Ritter, H. Schöning, K.-U. Sattler, T. Härder, S. Friedrich, and W. Wingerath, Eds., vol. 241. GI, 2015, pp. 659–662.

[10] D. Qiu, B. Li, and Z. Su, "An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 125–135, 2013.

[11] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[12] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[13] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, 2006.

[14] J. Kabbedijk, S. Jansen, and S. Brinkkemper, "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*, Irsee, 2012.

[15] N. Korkmaz, "Practitioners' view on command query responsibility segregation," Master Thesis, Lund University, 2014.

[16] G. Maddodi, S. Jansen, J. P. Guelen, and R. de Jong, "The Daily Crash: a Reflection on Continuous Performance Testing," in *ICSEA 2016, The Eleventh International Conference on Software Engineering Advances*, 2016, pp. 100–107.

[17] C. Jensen, C. Dyreson, M. Böhlen, J. Clifford, R. Elmasri, S. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. KÄfer *et al.*, "The consensus glossary of temporal database concepts—february 1998 version," *Temporal Databases: Research and Practice*, pp. 367–405, 1998.

[18] S. Scherzinger, M. Klettke, and U. Störl, "Managing Schema Evolution in NoSQL Data Stores," in *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy.*, T. J. Green and A. Schmitt, Eds., 2013.

[19] K. Saur, T. Dumitraş, and M. W. Hicks, "Evolving NoSQL Databases Without Downtime," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[20] M. Fowler. (2013) Schemaless Data Structures. *visited on 2016-10-11*. [Online]. Available: http://martinfowler.com/articles/schemaless/

[21] P. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.

[22] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber, "Understanding database schema evolution: A case study," *Science of Computer Programming*, vol. 97, no. P1, pp. 113–121, 2015.

[23] L. Meurice, C. Nagy, and A. Cleve, "Detecting and Preventing Program Inconsistencies under Database Schema Evolution," *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 262–273, 2016.

[24] A. Maule, W. Emmerich, and D. Rosenblum, "Impact analysis of database schema changes," *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 451–460, 2008.

[25] M. Callaghan. (2010) Facebook - Online Schema Change for MySQL. *visited on 2016-10-11*. [Online]. Available: https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/

[26] W. Keller, "The Bridge to the New Town - A Legacy System Migration Pattern," in *EuroPLoP*, 2000, pp. 261–268.

[27] T. Dumitras and P. Narasimhan, "No downtime for data conversions: Rethinking hot upgrades," Carnegie Mellon University, Pittsburgh, Tech. Rep., 2009.

[28] T. Dumitras and P. Narasimhan, "Why Do Upgrades Fail and What Can We Do about It?" in *Middleware 2009, ACM/IFIP/USENIX, 10th International Middleware Conference, Urbana, IL, USA, November 30 - December 4, 2009.*, ser. Lecture Notes in Computer Science, J. Bacon and B. F. Cooper, Eds., vol. 5896. Urbana, IL, USA: Springer, 2009, pp. 349–372.

[29] M. de Jong and A. van Deursen, "Continuous deployment and schema evolution in SQL databases," in *Proceedings of the Third International Workshop on Release Engineering*, Firenze, 2015.

[30] J.-M. Hick and J.-L. Hainaut, "Database application evolution: A transformational approach," *Data Knowl. Eng.*, vol. 59, no. 3, pp. 534–558, 2006.

[31] E. Domínguez, J. Lloret, A. L. Rubio, and M. A.

Zapata, "MeDEA: A database evolution architecture with traceability," *Data Knowl. Eng.*, vol. 65, no. 3, pp. 419–441, 2008.

[32] C. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the PRISM workbench," *PVLDB*, vol. 1, no. 1, pp. 761–772, 2008.

[33] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *VLDB J.*, vol. 22, no. 1, pp. 73–98, 2013.

[34] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices, 2013.

[35] Axon Framework. (2016) Reference Guide Axon Framework reference guide - Event Upcasting. *visited on 2016-10-11*. [Online]. Available: http://www.axonframework.org/docs/2.4/repositories-and-event-stores.html

[36] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383–393, jan 1995.

[37] L. Tan and T. Katayama, "Meta Operations for Type Management in Object-Oriented Databases," in *DOOD*, 1989, pp. 241–258.

[38] ISO/IEC 25010:2011, "Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," Geneva, CH, Standard, 2011.

[39] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Professional, 2010.

[40] S. Jansen, G. Ballintijn, and S. Brinkkemper, "A process model and typology for software product updaters," *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK, Proceedings*, 2005.

[41] E. A. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.

[42] V. Pulkkinen, "Continuous deployment of software," in *Proc. of the Seminar no. 58312107: Cloud-based Software Engineering*, 2013, pp. 46–52.

[43] T. Dumitras, P. Narasimhan, and E. Tilevich, "To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 865–876.

[44] M. Fowler. (2010) BlueGreenDeployment. *visited on 2016-10-11*. [Online]. Available: http://martinfowler.com/bliki/BlueGreenDeployment.html

[45] D. Sato. (2014) ParallelChange. *visited on 2016-10-11*. [Online]. Available: http://martinfowler.com/bliki/ParallelChange.html