AGH
University of Science and Technology in Krakow

Faculty of Computer Science, Electronics and Telecommunications

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

ANDRZEJ DĘBSKI, BARTŁOMIEJ SZCZEPANIK

SCALABLE ARCHITECTURE OF CLOUD APPLICATION BASED ON CQRS AND EVENT SOURCING

SUPERVISOR:
Maciej Malawski Ph.D

Krakow 2015

OŚWIADCZENIE AUTORÓW PRACY

Oświadczamy, świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym w załączniku), i nie korzystaliśmy ze źródeł innych niż wymienione w pracy.

.....

PODPIS

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektroniki, Telekomunikacji i Informatyki

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

ANDRZEJ DĘBSKI, BARTŁOMIEJ SZCZEPANIK

SKALOWALNA ARCHITEKTURA APLIKACJI W CHMURZE Z WYKORZYSTANIEM WZORCÓW CQRS I EVENT SOURCING

PROMOTOR:
dr inż. Maciej Malawski

Kraków 2015

Abstract

Cloud technologies, due to their capability of dynamic scaling of computing resources on demand, provide new opportunities for development of scalable applications. To fully benefit from these capabilities, the application architecture needs to be designed with the scalability as a main design objective. That enables developers to create low-latency solutions which handle millions of requests per second and adjust the resource usage to their current needs.

Due to the fact that this type of software is quite recent, we still do not have frameworks or architectural patterns widely accepted to be a standard in scalable distributed systems development. Moreover, new ideas are being proposed all the time and there is no consensus on any specific direction so far.

We did an extensive research of the current state of the distributed system design, focusing mainly on the concepts related to the event-driven architecture. This is one of the most popular approach amongst the all new findings we see today. We noticed a very particular interest in leveraging the concept of event sourcing (storing state as a sequence of events) in various forms. We became interested especially in its correlation with the Command-Query Responsibility Segregation principle (CQRS). We also asked whether several concepts from Domain-Driven Design, Reactive Manifesto and the actor concurrency model may be helpful for building the application based on event sourcing (ES) and CQRS. We decided to evaluate this architecture, especially focusing on the scalability in the cloud environment.

Cooperating with Lufthansa Systems GmbH & Co. KG, we developed a prototype flight scheduling application based on the CQRS+ES architecture. We designed it with scalability in mind by leveraging auxiliary concepts we had found during our research. We implemented the application with the Akka toolkit which is based on the actor model and supports event sourcing. Next, we deployed it to a cloud consisting of 10+ machines. Finally, we experimentally verified the scalability of both the write and the read parts of the CQRS+ES architecture.

In this study, we prove that it is possible to build a scalable application based on the Command-Query Responsibility Segregation and event sourcing patterns. We show that Domain-Driven Design and the actor model fits well to this architecture. Thanks to that, we expect the quick adoption of the CQRS+ES architecture by the industry as it provides many interesting advantages without sacrificing the performance.

Keywords: Scalability, CQRS, Event Sourcing, Domain-Driven Design, Reactive, Akka.

Acknowledgements

We would like to express our sincere appreciation to our supervisor Dr. Maciej Malawski for his invaluable support, guidance and patience.

Similarly, we would like to thank Stefan Spahr from Lufthansa Systems GmbH & Co. KG for his constant support, motivation and fruitful discussions.

This thesis could not happen without a Lufthansa Systems support we are grateful for. Special thanks to Dr. Dirk Muthig for the thesis proposal and the entire cooperation.

We cannot forget about Flexiant company and GWDG research centre as they shared computational resources with us allowing us to conduct performance tests.

Finally we would like to thank Dr. Daniel Żmuda, Maciej Ciołek and many other people for lots of suggestions and discussions.

This work was supported by EU PaaSage project (grant 317715).

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

This chapter lays out the general idea of the thesis. Section 1.1 describes the motivation for this work. Section 1.2 poses a main question that this thesis addresses. Section 1.3 states the goals of the thesis. Section 1.4 lists the requirements of the application which need to be implemented. Finally, Section 1.5 outlines the structure of the thesis.

1.1. Motivation

Nowadays, everyone is moving to the cloud and benefits from the economies of scale. The cloud provides more efficient resource utilization, easier infrastructure maintenance and lower costs relying in most cases on the standard hardware. The success of this model triggered a significant boost in the distributed computing field. The industry experience a ferocious but friendly fight between major companies which constantly supply the world with new ideas, concepts and frameworks for building distributed applications. However, there are still not many well-established patterns and solutions. That is a big issue when it comes to build applications today as one need to guess what direction he should take.

Cloud technologies, due to their capability of dynamic scaling of computing resources on demand, provide new opportunities for development of scalable applications. To fully benefit from these capabilities, the application architectures need to be designed with the scalability as a main design objective. That enables developers to create low-latency solutions which handle millions requests per second and adjust the resource usage to their current needs.

Due to the fact that this type of software is quite recent, we still do not have frameworks or architectural patterns widely accepted to be a standard in scalable distributed systems development. Moreover, new ideas are being proposed all the time and there is no consensus on any specific direction so far.

Lufthansa Systems GmbH & Co. KG (LSY) faced this problem too in the PaaSage project [1], which aims to create a platform based on a model driven engineering (MDE) approach. It facilitates development of applications deployed across multiple clouds. As LSY were responsible for an industrial use case, they selected their scalable

flight scheduling application as an example of an application that should demonstrate the scalability requirement. That opened the opportunity to experimentally evaluate several of the most promising concepts in distributed systems.

1.2. Problem statement

Together with Lufthansa Systems, we did an extensive research on the current state of distributed system design, especially focusing on the event-driven approaches. The results are presented in Chapter 2.

We became interested in the architecture based on the ***Command-Query Responsibility Segregation (CQRS)*** [2] and ***Event Sourcing (ES)*** [2] principles. Moreover, we found several other concepts like ***Domain-Driven Design (DDD)*** [3], the ***Reactive Manifesto*** [4] and the ***Actor Model*** [5] that may help building an application based on CQRS and ES patterns. All of these ideas are explained at length in Chapter 3.

We decided to evaluate this architecture. The most important question for us is the scalability of the CQRS+ES approach. That lead us to the following questions:

- Is it possible to build a flight scheduling application using the selected ideas?
- Is the architecture based on the these patterns horizontally scalable?

The answers on these questions should clear any doubts whether the discussed patterns are suitable for scalable cloud applications or not.

1.3. Goals of the thesis

Since the main objective is the evaluation of the CQRS+ES architecture scalability, we identified the following specific goals of this thesis:

Discuss the recent architectural patterns for building scalable systems

The goal is to discuss the recent architectural patterns for scalable systems, with the focus on the event-driven approach. This includes describing evaluated ideas in detail and presenting the related work.

Propose the architecture of the flight scheduling application using CQRS+ES

To combine all of the aforementioned ideas in a coherent architectural model for our particular use case with the horizontal scalability in mind.

Experimentally assess the scalability of the proposed solution

To implement the proposed architecture, deploy it in a cloud environment and validate it with a stress testing tool to evaluate if the application is capable of handling more requests with the increasing number of resources.

Deliver the industrial business use case for PaaSage project

The implementation of the flight scheduling application should be later used in the PaaSage project. It should be used as an acceptance test for the PaaSage-platform capabilities.

Fulfilling these goals should allow us to answer the questions posed in Section 1.2. Even though the last objective is not directly connected to the problem, it is definitely helpful in a way that the final implementation cannot end up to be unrealistic.

1.4. Requirements of flight scheduling applications

The application requirements were set by Lufthansa Systems GmbH & Co. KG. The application implements a flight scheduling service, with a business logic that is to not elaborate as the focus is put on architecture and performance evaluation.

The main subject of the application is a flight schedule which consists of *airplanes* and *rotations* (see Figure 1.1). A *Rotation* is an ordered list of *legs* which do not overlap. A *leg* is a directed relocation of an airplane between two airports at a given time. It is identified by a *flight designator* which consist of a carrier identifier, a flight number, an optional suffix, a departure airport and day of origin. *Airplanes* can be assigned to *rotations*.

The support for bulk loading of the schedule from SSIM files is also required. The SSIM file format is an industry standard for exchanging flight schedule information between airlines. As SSIM does not preserve rotations explicitly, support for loading leg to rotation assignment from a separate file is needed. During the schedule planning process no constraint should be forced except for not allowing for overlapping legs in rotations. However, when the planning is finished a schedule planner needs to check if it is correct. The application needs to validate the schedule by checking the following constraints:

- Consecutive legs have to share destination and origin airports what enforces ***rotation continuity*** (Figure 1.2).
- Every airport and airplane have their ***minimal ground times*** defined. The time between all pairs of consecutive legs has to be greater than both of the values (see Figure 1.3).
- Legs must have a ***unique flight designator***. It means that there are no legs in the schedule that shares the same day of origin, flight number and departure airport (see Figure 1.4).

No other validity checks should be performed, e.g. there is no need to check if the airplane is suitable for handling each leg in the rotation in terms of its capacity or range.

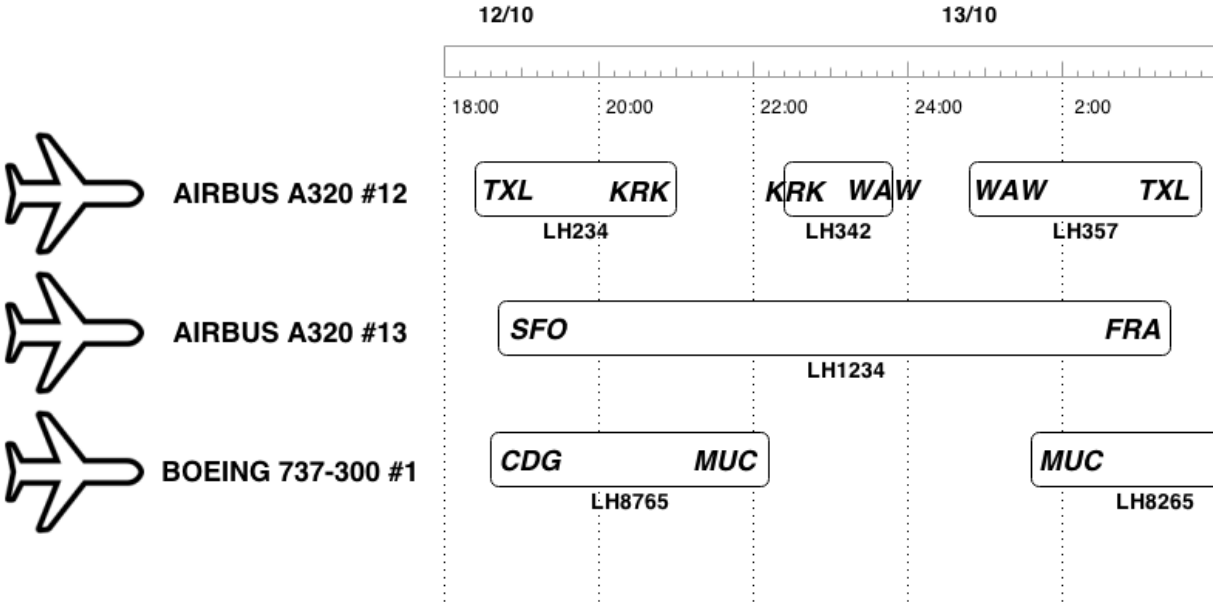


Figure 1.1: Example of a valid flight schedule. Rotations are displayed horizontally with their assigned airplanes. Legs are described with their origin and destination airports along with the flight designator placed below.

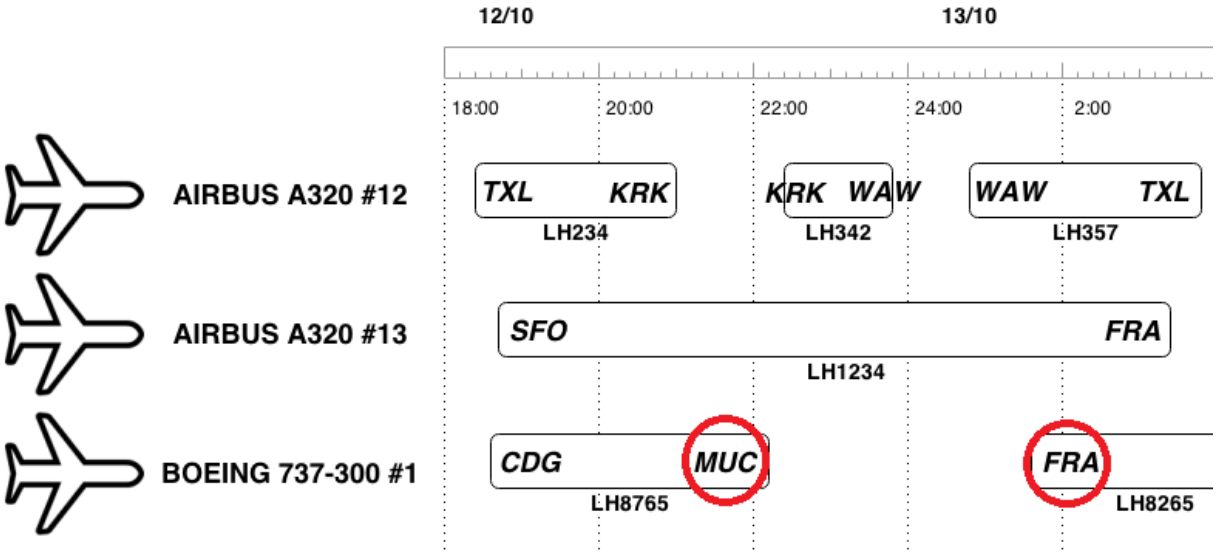


Figure 1.2: A flight schedule with a rotation that does not hold continuity constraint.

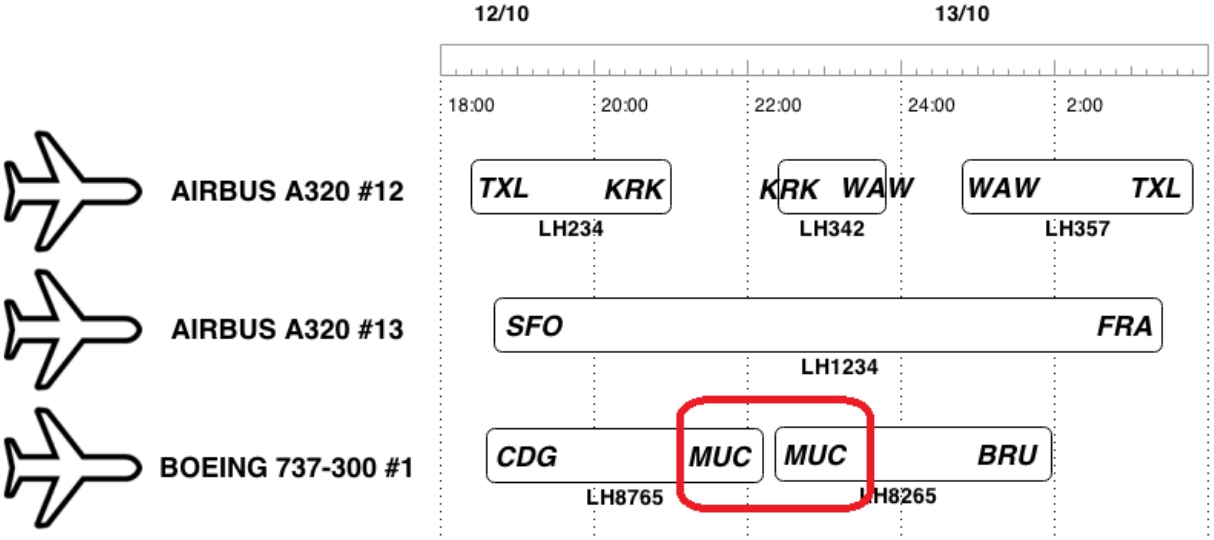


Figure 1.3: A flight schedule with two legs that do not hold minimum ground time constraint.

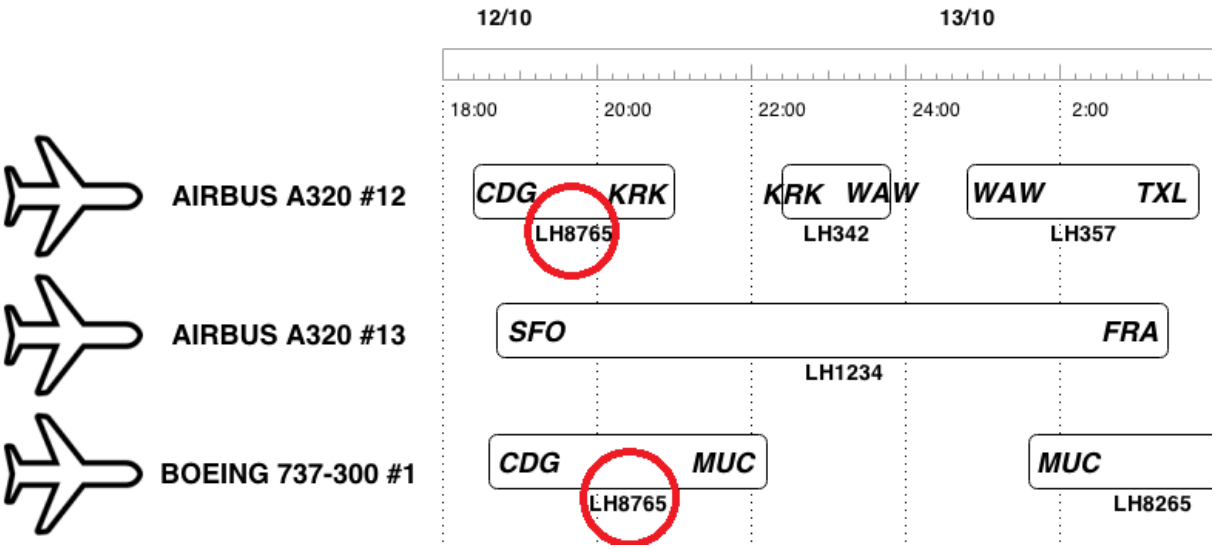


Figure 1.4: A flight schedule with two legs that do not hold unique flight designator constraint.

1.5. Outline of the Thesis

The thesis is organized as follows. Chapter 2 presents the newest trends in the event-oriented architectures. In Chapter 3 we thoroughly describe evaluated ideas and outline their pros and cons. Chapter 4 contains our proposal of the flight scheduling application architecture based on the chosen patterns. In Chapter 5 we described some of the challenges we faced during the implementation phase. Evaluation results and methodology are presented in Chapter 6. Finally, in Chapter 7 we included conclusions, lessons learned and future work.

1.6. Summary

In this chapter we presented the background, the motivation and the goals of this thesis. Nowadays, lots of new concepts appears constantly in the distributed system design. We found the CQRS and event sourcing concepts the most interesting ones and decided to evaluate it, focusing especially on the scalability. In order to achieve that, we want to design, build and experimentally assess the scalability of a flight scheduling application based on the CQRS+ES architecture.

2. State of the art of event-driven architectures

Event-driven architecture [6] is an approach for building loosely-coupled systems by exchanging events – the actual facts describing what happened to them. It covers many different aspects of computing like transaction processing, system integration and model building. In this chapter we describe current trends in this field.

In Section 2.1 we sketch out how the event-driven approach influences the transaction processing systems of today. Next, in Section 2.2 we discuss different consistency and ordering guarantees which are given by event-oriented systems nowadays. In Section 2.3 we lay out current trends in processing continuous streams of events. Section 2.4 shows approaches to building useful and persistent data representations from events. In the following Section 2.5 we explain how the industry leverage the idea to store all events that happened in the system. Section 2.6 describes newest logging and monitoring techniques which are deeply event-driven. Section 2.7 tells us about using events to integrate different systems. Finally, the last Section 2.8 presents how the event-oriented approach influence on contemporary user interface tooling.

2.1. Transaction processing

Event-driven programming became one of the most popular concurrency models recently. We see a lot of solutions based mostly on the reactor pattern [7], of which the most successful are the NodeJS framework [8], the Nginx web/proxy server [9] and the Netty framework [10]. The key factor of their success was the efficient processing of incoming events (requests) by effective thread manipulation. This model is now being adapted to current distributed systems and cloud computing trends. The Reactive Manifesto [4] notices that "today's demands are simply not met by yesterday's software architectures" and claims that a new "reactive architecture", based on reaction to events (message-driven) is the way to go.

There are a lot of attempts to model the transaction processing in this way. The Akka toolkit brought the actor concurrency model [5] to the JVM platform based on Erlang's actor model [11]. Actors are location-transparent, single-threaded entities which communicate by message-passing. Vert.x, a NodeJS clone for Java based on Netty,

introduced the concept of verticles, very similar to Akka actors [12]. LMAX presented the Disruptor [13] which allows for efficient enqueuing and processing millions of transactions per second on a single thread.

Interestingly, all of those attempts define fine-grained, single-threaded processing units and queues (a.k.a. mail-boxes) that store the requests to process. They react on events producing new events as a consequence. In fact they are all derivatives of the Staged Event-Driven Architecture (SEDA) [14]. Moreover, the processing units resemble entity concept proposed by Pat Helland in [15] as a way to achieve the "Almost-infinite scalability" and Eric Evans' concept of aggregate defined in Domain-Driven Design [3].

2.2. Causality and consistency guarantees

If a strict ordering between events is necessary in a distributed system, the standard consensus resolution methods may be used. Apart from the widely-known Paxos [16], there are newer and simpler algorithms available: Zookeeper Atomic Broadcast (ZAB) [17] and RAFT [18].

Thanks to Brewer's CAP theorem [19], we know we cannot guarantee high availability without relaxing the consistency guarantees. Eventually consistent [20] models are used instead of transactional updates of all models when events are published. Moreover, often ordering between closely occurring events is not preserved.

Concurrent access to eventually consistent data may result in the need of conflict resolution. This is why there is a big interest currently in a concept of Conflict-free replicated data types (CRDT) [21]. CRDT is a data type which always gets to the same state regardless of the ordering of applied events (so called strong eventual consistency). The simplest example is representing a counter as a list of increment events instead of a single integer field. The CALM theorem [22] generalizes this idea, stating that eventual consistency can be guaranteed by the logical monotonicity of the operations. Languages like Bloom [22] and Lasp [23] propose a new approach of writing distributed systems based on the CALM theorem conclusions, which are guaranteed to achieve strong eventual consistency.

Stronger models are being investigated. COPS [24] introduces a causal consistency model with convergent conflict handling which does not hinder scalability. Google Spanner [25] implements an externally-consistent model which guarantees linearizability of non-overlapping transactions. Also distributed databases like Dynamo [26] uses causal relationship, e.g. in the form of vector clocks to detect conflicting updates.

2.3. Stream processing

The MapReduce [27] computation model, presented by Google in 2004, was a revolution in the data science world and started the "Big Data" epoch. Now we observe a gradual focus shift from batch-oriented computations towards stream processing as it allows decreasing the time between getting the data and making use of it. It is

really important in cases such as news recommendation.

That caused a significant boost in the area of processing data using streams. Complex event processing [28] ideas were revived in a modern, distributed, cloud-ready fashion. Twitter presented Storm [29] which allows creating a custom stream processing topology in the shape of a directed acyclic graph. Similarly as in MapReduce the user creates only transformations (graph vertices) and the distribution and reliability is provided under the hood. The Yahoo! S4 [30] approach is very similar. Spark Streaming [31] is an attempt to unify batch-oriented and real-time computation models. The idea is to treat streaming computations as a series of deterministic micro-batch computations on small time intervals. Google's Millwheel [32] is a streaming system that focuses on fault-tolerant state building from processed events by using exactly-once delivery semantics and checkpointing.

The data science world is permeated with functional programming concepts. Spark [33] and DryadLINQ [34] provide abstractions that allow for similar data transformations like functional languages have, e.g. map, filter, fold. The same approach was successfully adopted in streaming systems like Storm or Spark Streaming. Twitter's Summingbird [35] is an attempt to create a similar data transformation abstraction shared both by batch- and stream-oriented processing. Reactive Extensions [36] is a .NET library which introduced a concept of observables which treat data streams like asynchronous data sequences and allows applying standard functional-style collection operators (LINQ) to them. Akka Streams [37] presents a similar approach on the JVM platform. There is also an initiative called Reactive Streams [38] aiming to unify these stream processing interfaces, focusing especially on back-pressure.

2.4. Model building

The Lambda Architecture [39] approach combines both batch- and stream-oriented computations within a single use case. The architecture is composed of three components: a batch layer for computing a precise model of the entire data, a speed layer approximating the impact of recent updates and a serving layer which answers queries based on both models. ElephantDB [40] and Druid [41] are databases crafted for handling a large number of updates in a streaming/batch-oriented manner and they are often used to merge the speed and the batch layer.

Command-Query Responsibility Segregation (CQRS) [2] is a concept which separates updates of the data and query models. The latter are built from the stream of events generated by the command processing part of the application. The Axon Framework [42] allows building applications based on this principle. Datomic [43] is a distributed database which is based on a similar idea: updates are processed by transactors which mutate the state in the database and reflect accordingly all derived query models and caches. The Change Data Capture concept, in which the source of events are updates to the relational database, allows synchronizing command and query models on a lower level. It is especially useful when dealing with legacy systems. This is provided for instance by LinkedIn Databus [44].

2.5. Event Sourcing

The idea to leverage the state/events duality and to store the state as a log of all updates was coined as event sourcing [2]. The Axon Framework [42] and Akka Persistence [45] allows event-sourcing entities in the application, however, the latter still does not provide a fully-functional support for projection building from stored events [46]. Eventuate [47] is a high-availability implementation of event sourcing in which conflicting events may occur. EventStore [48] is a distributed database crafted especially for storing events, performing complex event processing and building projections out of them.

LinkedIn presented a log-centric infrastructure [49] based on a similar idea as event sourcing. Events are gathered from all sources and stored for some time or even indefinitely. That eases fault-tolerant model building by providing at-least-once consumption model and ability to go back and replay old events. The core component of the infrastructure is Kafka [50], a durable, distributed log system similar to a write-ahead log in databases. Events are stored in replicated partitions and preserve strict ordering within a single one. Samza [51] is used for building models out of Kafka streams, locally to the data in rest. This approach was coined as Kappa Architecture [52] as an alternative to the Lambda Architecture. The big advantage is that there is only a single processing code as both real-time computation and reprocessing are done in the same way. Moreover, the model building borrows close-to-data paradigm from batch-oriented computations.

2.6. Logging and monitoring

Logging and monitoring are the most classic event processing use cases. The most popular approach for dealing with large amount of logs currently is to flush them locally to disks, periodically push it in a batch manner to the distributed log server, aggregate them and finally present in a useful representation. This approach was followed by Facebook's Scribe [53], Elastic Logstash [54] or Apache Flume [55]. Usually data is piped further to large data clusters like Hadoop [56] for analytics and/or to search servers like Elasticsearch [57]. Currently, there is a trend towards pull-based distributed log servers like Apache Kafka as they are less prone to event loss and provide consumer backpressure.

Google presented Dapper [58], a distributed tracing infrastructure for aggregating and correlating low-level service events in order to produce performance statistics and request flow visualizations. Twitter Zipkin [59] is its open source implementation. As the monitoring tools are more and more popular, new solutions for aggregating events in time series data appeared, e.g. InfluxDB [60] or Graphite's Whisper [61]. The aggregated data is usually presented by charting tools like Grafana [62] or Kibana [63].

2.7. System integration

The event-based type of integration between systems is very popular in the industry. It is achieved by using message-oriented middlewares in the form of standalone queues or Enterprise Service Bus (e.g. Mule [64]). Recently also distributed, persistent logs like Apache Kafka are becoming popular publish-subscribe solutions.

Gregor Hohpe and Bobby Wolf in their book Enterprise Integration Patterns [65] presented a set of common patterns for dealing with message/event-oriented systems. That introduced a standard vocabulary and was the reason of the fact that tools like Apache Camel [66] appeared, making designing and building event-driven systems easier.

2.8. User interface

User interfaces are event-based since the invention of the MVC pattern [67] in which a view reflects the changes in the model on the fly. Currently popular UI solutions like AngularJS [68], Backbone [69], JavaFX [70] or Windows Presentation Foundation (WPF) [71] are based on the Presentation Model pattern [72] and data binding [73] concepts which aim to decouple presentation logic from the actual UI components. This is achieved by making UI controls event-based and subscribing them to changes in the presentation model.

ReactJS [74] is a library for building real-time web applications focusing on efficient partial updates of the page based on received events. Facebook Flux [75] is an attempt to introduce dataflow programming paradigm for user interface building. State of the presentation model reacts on action events, and views are updated based on update events from state entities.

2.9. Summary

In this chapter we gave a brief overview of event-driven architectural patterns. Industry leaders like Google, Twitter, Microsoft or Facebook invested recently lots of their resources in the research on this field. That caused a significant boost and many new event-driven solutions and techniques for transaction processing, model building, stream processing or displaying user-interface have just appeared. What is more, the evolution speed of the field does not decrease and we should expect many more exciting findings to appear soon.

3. Evaluated architectural concepts

This chapter introduces architectural concepts and tools that we decided to evaluate in terms of scalability. A more comprehensive description of these concepts can be found in the referenced bibliography [2] [76] [77].

In Section 3.1 we present CQRS, an architectural pattern for improving application responsiveness. Section 3.2 describes the novel persistence solution called Event Sourcing. Section 3.3 explains the basic ideas behind the Domain-Driven Design approach for building and maintaining complex systems. In Section 3.4 the Reactive Manifesto guideline for building efficient applications is discussed. Finally, Section 3.5 lays out Akka toolkit description, the message-passing middleware built upon Reactive Manifesto ideas.

3.1. Command-Query Responsibility Segregation (CQRS)

CQRS originates from the Command Query Segregation pattern, a widely accepted pattern in the Object-Oriented Programming community proposed by Bertrand Meyer in 1988 [78]. It states that every method in a class should either be a query action returning data without any side-effects or a command that mutates the state and does not return anything.

Greg Young picked up this concept and applied it on the architectural level during his research on Distributed Domain-Driven Design for high performance systems in 2008 [79]. He admitted [79] that the concept is closely related to the Side-Effect Free functions presented by Eric Evans in his DDD book [3]. A year later he coined the Command-Query Segregation Principle term [80] to avoid confusion.

The CQRS (depicted on Figure 3.1) principle is a kind of Event-Driven Architecture [6] that separates an application into two disjoint parts. The first one is called a write, command or domain model. It is responsible for handling all state changing operations (commands), i.e. validating them, updating the application state and publishing events. The second one is called read or query model. It both answers user queries and updates its model based on received events from the write side. Nothing stops from creating multiple query models crafted for each application's use case separately.

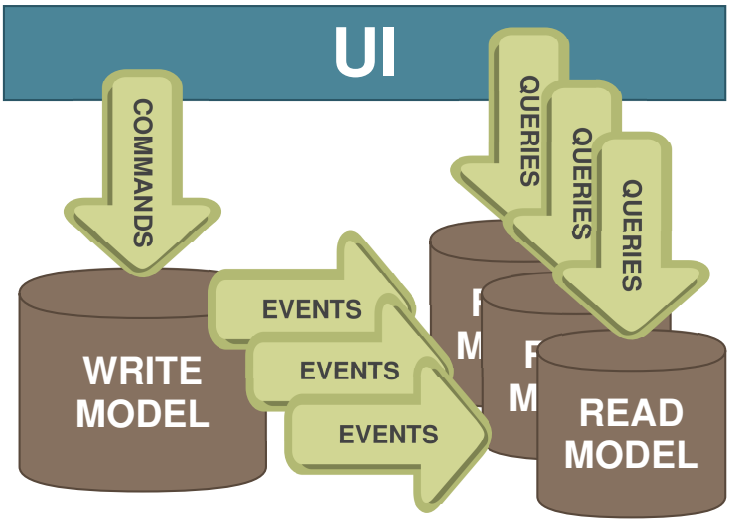


Figure 3.1: The diagram depicts the idea of CQRS. Write model handles all user commands, validate them and produces events. Those events are published to all read models which updates their query models and become ready to handle new queries.

CQRS pushes the ideas of secondary indexes and materialized views from relational databases to the architectural level and make them more appealing to the NoSQL world. In fact, a materialized view is the simplest CQRS implementation in a strictly consistent manner. However, most of the times the eventually consistent model is perfectly enough. It is better to make queries slightly more outdated (less than 10s usually) and gain better performance. Even having a relational database does not prevent users from dealing with outdated data as they load the page (query result), usually thinks a while and before they take an action the model may already be outdated.

Adhering to this pattern comes with many advantages:

- You can choose ***different databases for write and read models***. In many cases read to write operation ratios are greater than one or two orders of magnitude. That means you can select the most performant database for your query use case without losing advantages of your favourite (e.g. fully ACID, relational) database for state mutating operations. Sometimes there is no point in scaling the write model.
- Moreover, you can choose ***different databases for every query use case***. You can use for instance simultaneously ElasticSearch [57] to answer full-text search queries, Hadoop [56] for building recommendation model once a day and Riak [81] for counting page views.
- Decoupled ***models become focused and simple***. That means easier maintenance and better caching possibilities.
- Queries are inherently idempotent, so it is ***much easier to scale out a model without writes***.
- Eventually consistent models ***remove transaction complexity and performance overhead***. Commands are timely acknowledged by the write model without waiting for every model to update. Read models can consume events in their own pace, regardless of bursts of events.

Great benefits always come with a cost:

– You need to ***synchronize multiple, distributed data models***. It is not a trivial task if you want to make it efficient, correct and resilient (e.g. events may disappear, nodes may go down). Still not too many tools were created specifically for CQRS.

– A Multicomponent application with multiple types of databases results in ***deployment model that is harder to maintain*** and puts a way more work on devops. It may lead also to code duplication when two models are quite similar.

– ***Not every problem is easy to model*** in this way. For instance, it is hard to provide unique email semantic for users in the write model as it requires to query the read model with email index first before adding a new user. Eventual consistency makes it even harder.

CQRS is an interesting concept based on the old pattern and reinvented anew in the DDD community. The most distinctive feature is the ability to address different non-functional requirements for write and read operations, which comes with a cost of data synchronization. A more comprehensive description of the concept may be found in [2], [82].

3.2. Event Sourcing (ES)

Event sourcing is a term coined by Martin Fowler in 2005 [83] for event-oriented persistence in contrast to popular state-oriented methods. The idea is to record all changes that happened in the system and recreate the state based on all of those changes when it is needed.

The idea is not new at all and dominates in most of mature sectors like finance or law. They are operating with journals, ledgers, transaction registers and addendums on a daily manner for a reason. They cannot imagine destroying or changing any transaction data. Auditing is always strictly required in the software built for them. What is more interesting the pattern is also common in software. All databases that presents a state-oriented interface are in fact deeply event-oriented. They register transactions in a write-ahead log and update its state accordingly. It is not possible to achieve correctness without the log. Similarly, many distributed systems are based on the state machine replication or primary-backup models which use ordered event logs in their core [49].

In event sourcing, events are stored in a stand-alone ordered log directly (a.k.a. event store) and stay there indefinitely. The ordering is usually preserved on entity or table level to avoid efficiency problems. When the state is needed, it is recreated by state machines which reply all events in order (see Fig. 3.2). It's important to store events, not commands, as the business logic or external service responses may change over time and that final state could be different. That would also lead to replaying side-effects.

This approach to persistence has a lot of advantages:

– It gives a ***full history of what has happened*** to the system. The complete audit log is really important for business people, both from safety and analytics perspective.

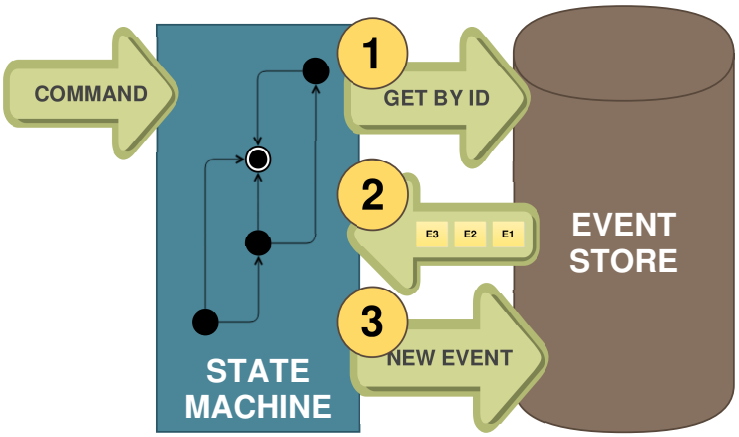


Figure 3.2: Event sourcing, depicted on the diagram, introduces a slightly different approach than the standard flow of dealing with entities in an enterprise world is: deserialize, mutate state, serialize. First the storage is asked for a list of all events for a given entity (1), then the brand new state machine applies all retrieved events (2). Finally the user command is validated and if was successful, new event is produced and stored (3).

- We get ***100% reliable audit log*** in contrast to most of the implementations where the state is changed in the database and then audit log is updated. When the system crashes in between and we did not harness complicated distributed transactions we may end up with inconsistencies.
- When events are first-class citizen in the system, it is ***easier to understand by business people***. Events represents real business facts they are familiar with.
- The ***semantic of data updates is not lost*** as it is often in the case of dealing with state. It is hard to reverse-engineer what happened to the system when we see only its current state.
- Events are only appended to the log and read sequentially. That enables ***efficient storage implementations*** without many disk seeks involved. Moreover, write-once read-many drives may be used for security reasons.
- It opens new possibilities for troubleshooting as we can ***recreate the application state that it had in any given time***.
- There is ***no object-relational impedance mismatch*** and ***no database model migrations*** needed as we store only events, not a state.

There are also some potential problems with event sourcing that one need to be aware of:

- Maintaining ***total order of application events creates a bottleneck*** for events writing. In most of the cases it is possible to have more fine-grained transaction units in the model, for instance an order per entity or entity type.
- The ***replay time increases with time*** as more and more events is being stored. A possible solution is to regularly create snapshots of the state to avoid replaying all events. If the state representation changes later, recreating the snapshots is not a problem.

– Storing all events may lead to ***running out of disk space***. It may be a problem if more than million events per second are generated. Usually it is true only for non-critical data like user clicks or ad impressions and we can store them only for a given time. Other option is to compact the data and leave only the last entry for a given entity, what is often enough to recreate the state.

– ***Event definitions may change*** during the application lifetime and we need to take care about their versioning. We can solve it by making the business logic aware of all versions of the event, writing converting chains to newer versions (upcasting) or by using a weak serialization format and handling missing attributes properly in the code.

Event sourcing is efficient only for dealing with state mutating operations in the system. For efficient querying we need to create a model derived from the log and that is why when using event sourcing CQRS is a must. It is not the same in the other way round, however, CQRS gains even more advantages when combined with event sourcing:

– You can ***add a new read model later***, long time after the system was created. It is hard to envision upfront how the data may be used.

– There is ***no need to persist the state***. You can cache the state in memory. When something breaks it is easy to recreate the state. Persistence may be treated only as an optional way for speeding up the replay process or handling more data than can fit into memory. And even then it is easier to manage - no backups, replication or model versioning needed.

– You ***can change freely the projection shape***, e.g. when business requirements are changed, technology stack is updated or a bug was found.

– ***Synchronization of read and write models becomes easier***. There is no need to keep both state and separate events for synchronization or setting up a sophisticated ETL process between databases.

– It is possible to see ***the read model state in any point in time***. It is especially useful for business people who introduce new analysis tools. They can see how their analysis would look like if they did it a year before.

Event Sourcing is a surprisingly common idea that predates computing for hundreds of years and prevalent in software engineering in many different shapes, such as state machine replication, audit logging, write-ahead log or database change capture. Instead of maintaining the application state in the database, immutable events are stored and the state is recreated from them when it is needed. We get a complete history of what happened to the system with the cost of recreating the state when we need the data and that is why it needs CQRS. When used together, those two patterns create a powerful tandem that makes the application maintenance easier and gives better performance.

3.3. Domain-Driven Design

Domain-Driven Design is a software development approach proposed by Eric Evans in 2003 [3] meant to deal effectively with complex and evolving systems. It defines both ***strategic patterns***, high-level guidelines for large system design and ***tactical patterns***, class-level building blocks for business logic modelling.

The core rules (strategic patterns) suggest i.a.:

- Close cooperation of the development team with ***domain experts*** to understand business processes which later is reflected in the software.
- Defining a common lingo between them called ***ubiquitous language*** which is then used across all artifacts, e.g. in codebase or documentation.
- Partitioning of the system into ***bounded contexts***, manageable and coherent pieces in terms of business logic. They are later mapped into independent software modules with their own data model and well-defined external interface called ***published language***.
- Prioritizing bounded contexts and focusing human resources on ***core subdomains*** of the business which gives competitive advantage. Considering of-the-shelf solutions for ***generic subdomains***.
- Building a ***domain model*** in the code that defines business logic events and behaviors using the ubiquitous language. It should be separated completely from any technical concerns.

Tactical patterns introduce a level of abstraction for building a domain model. It helps experts and developers to reason about the codebase in terms of business processes and behavior instead of classes and state. The most interesting ones are:

- ***Value object*** represents a set of attributes in the domain that are indifferentiable when all its properties are the same, e.g. date or account balance. Immutable implementation is suggested.
- ***Entity*** represents a concept with an identity.
- ***Domain event*** records actual facts that happened in the system.
- ***Aggregate*** defines a transactional unit in a system and protects its own invariants. A single transaction can involve only a single aggregate. It consists of entities and value objects. It accepts user commands, mutates its state and optionally produces domain events.
- ***Repository*** abstracts the implementation details of the storage. If stores and fetches aggregates.
- ***Saga*** defines a long running business process. In contrast to aggregates, it receives events and trigger commands.
- ***Application service*** integrates the infrastructural details with the domain model. Orchestrates repositories, sagas and aggregates. Handles user requests and transform them into domain commands.

DDD has a lot in common with CQRS+ES architecture. Domain model is a great fit for write model implementation and deprivation of all queries makes it even more focused on the system behavior and facts. Aggregates can be eventsourced using domain events they produce. They are also a fine-grained transactionality unit so totally ordered event log is not needed. Additionally, event sourcing makes persistence-agnostic implementation of the domain model a trivial task.

Summarizing, Domain-Driven Design is a software development methodology focused on expressive business logic models, communication, modularization and setting the right priorities. It integrates well with CQRS and event sourcing ideas.

3.4. Reactive Manifesto

Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson in the Reactive Manifesto [4] claim that 21st century expectations in terms of software cannot be fulfilled with 20th century techniques. They presented necessary aspects of the relevant architecture that fits to cloud-based deployments, guarantee 100% uptime, process petabytes of data and respond in milliseconds. Applications adhering to these guidelines are called Reactive Systems. According to the manifesto, systems built in this way are flexible, loosely-coupled and scalable. They give user a better, interactive experience and effectively deals with failures.

The Manifesto defines four basic traits characterising a modern architecture. The application may be called a Reactive System when it has these properties (see Figure 3.3):

– ***Message-driven*** as they are using asynchronous message-passing concurrency model. That introduces non-blocking communication, loose coupling, location transparency and back-pressure.

– ***Resilient*** to failure by leveraging replication, bounding failures within isolated components and delegating recovery process to other units.

– ***Elastic*** so it can use more or less resources based on the current workload. That requires no contention point and ability to distribute or replicate components.

– ***Responsive***, what means that all user requests are responded quickly. It is the primary goal of the reactive architecture achieved by the previous aspects.

The authors of the Manifesto claim that this should be enough to provide the best experience for end users. Additionally that helps building composable systems as the traits apply to all levels of scale.

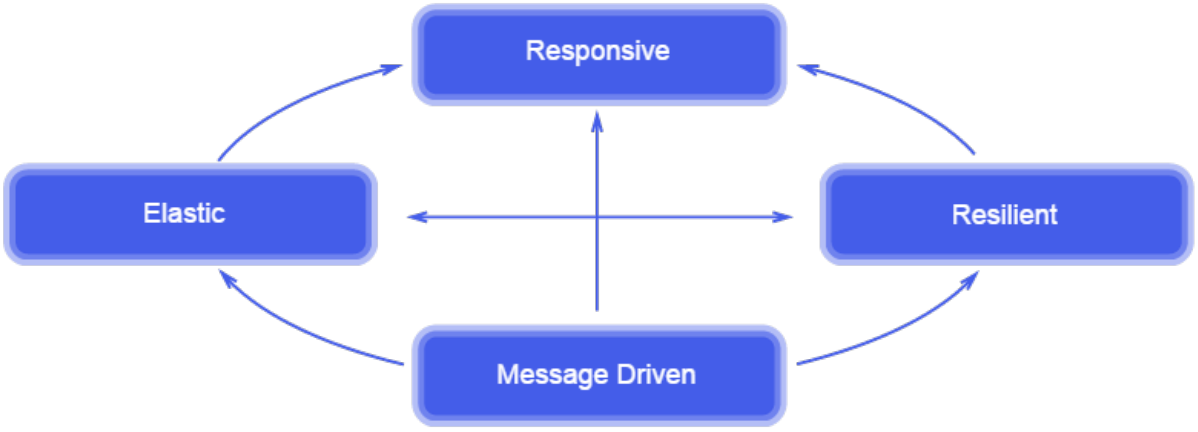


Figure 3.3: The diagram presents four traits of the Reactive System and their relationships. Responsiveness is an effect of applying all other aspects. Source of the diagram: http://www.reactivemanifesto.org/

3.5. Akka toolkit

The Akka toolkit [84] is a middleware simplifying the construction of Reactive Systems on the JVM platform. It was created by Jonas Bonér in 2009 and is now part of the Typesafe Reactive Platform. It exposes Java and Scala interfaces. Even though multiple concurrency models are supported, the actor model is a core. This model was invented in 1973 by Carl Hewitt [85] and was popularized by Erlang community.

The actor model is an implementation of the active object pattern. Execution units called actors are very fine-grained and lightweight in contrast to threads. The communication between them is ***message-based*** and ***asynchronous*** (see Listing 3.1). Actors do not share mutable state and hence synchronization is not needed. An actor serves a purpose of the unit of concurrency. It processes messages from its own mailbox one by one.

Akka is distributed by design. Actors are completely unaware of the deployment model. They send messages to ***location transparent*** references (see Listing 3.1) and never make locality assumptions. That introduces elasticity as it enables dynamic changes in the infrastructure like scaling up (adding more threads) and scaling out (adding more nodes) without modification.

Finally, resiliency is accomplished with the ***hierarchical actor systems***. The lifecycle of each actor is supervised by its parent. Supervisors decide how to remedy failures of their children, e.g. they can restart the actor or escalate the problem upwards in the hierarchy tree.

Akka is not a framework but a modularised toolkit. It consist of multiple building blocks, modules and plugins. The most important features are ***routers*** and ***clustering*** support. There are many routing strategies available, e.g. round-robin, consistent hashing, broadcasting. The most interesting is adaptive routing that takes into consideration CPU usage or pace of processing (mailbox sizes). Clustering support offers a membership service implemented with peer-to-peer gossip protocol. It provides automatic failure detection, notifies about cluster updates (see Listing 3.2) and supports routers which dynamically update their routing list based on who joined or left the cluster.

```
class MyActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}

val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor")

myActor ! 97
val futureResponse = (myActor ? 2).mapTo[Int]
```

Listing 3.1: The listing presents an example of the actor and its usage. It accepts an integer and replies back with the result of the computation. Actors reside in the actor system. Bang operator is used to send messages. Ask operator creates a future representing the result of the query in the asynchronous style. The example is taken from the Akka documentation.

```
class SimpleClusterListener extends Actor with ActorLogging {
  val cluster = Cluster(context.system)

  override def preStart(): Unit = cluster.subscribe(self,
      initialStateMode = InitialStateAsEvents, classOf[MemberEvent])
  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive = {
    case MemberUp(member) => log.info("Member is Up: {}", member.address)

    case MemberRemoved(member, oldStatus) =>
      log.info("Member is Removed: {} after {}",
          member.address, oldStatus)

    case _: MemberEvent => // ignore
  }
}
```

Listing 3.2: The listing presents an Akka actor which uses cluster membership service. When started, it subscribes for events of the interest which are later provided as messages. The example is taken from the Akka documentation.

```
class ExamplePersistentActor extends PersistentActor {
  override def persistenceId = "sample-id-1"

  var state = ExampleState()
  def updateState(event: Evt): Unit = { state = state.updated(event) }

  def numEvents = state.size

  def receiveRecover: Receive = {
    case evt: Evt                                 => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  def receiveCommand: Receive = {
    case Cmd(data) =>
      persist(Evt(s"${data}-${numEvents}"))(updateState)
      persist(Evt(s"${data}-${numEvents + 1}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
      }
    case "snap"  => saveSnapshot(state)
    case "print" => println(state)
  }
}
```

Listing 3.3: The listing presents the actor leveraging Akka persistence module. Persistence id differentiates events of different actors. Actor implementation is separated into two parts. The first handles commands, the latter one recreates the state based on stored events. The example is taken from the Akka documentation.

Akka Persistence is a module that makes actors durable. It allows storing incoming messages (command sourcing) or produced events (event sourcing) in the database and thus recreate the actor state when it fails. The process is transparent for the actor and resembles active record pattern. A persistent actor has a special identifier assigned (see Listing 3.3). When the actor is started, all messages with this identifier are retrieved from the database and replayed. There are a lot of plugins available for different storages and snapshots for quicker replays are also supported. There is also a concept of a view which subscribes to events of a given persistent actor. It is planned to enhance them in order to support CQRS [46].

There is also a lot of different interesting modules, i.a.:

- ***Akka Streams*** which implements the idea of Reactive Streams, a standard for asynchronous stream processing with non-blocking back pressure [38],
- ***Akka HTTP*** for exposing actors to the web via HTTP,
- ***Cluster Sharding*** for handling stateful actors that together consume more resources (e.g. memory) than fit on one machine,
- ***Distributed Publish Subscribe in Cluster*** for dynamic multicast communication.

All in all, the Akka toolkit introduces an actor-based concurrency model on the JVM platform. It adheres to the Reactive Manifesto guidelines as actors communicate in a message-driven way, location transparency makes them elastic and supervision hierarchy ensures resiliency. Akka provides many useful modules for dealing with routing, clustering, event sourcing and many others.

3.6. Summary

In this chapter we presented the CQRS and the event sourcing in detail, listing their their advantages, drawbacks and possible places where they can be successfully applied. Additionally we explained the Domain-Driven Design and the Reactive Manifesto, interesting guidelines that may help design application architecture based on CQRS and event sourcing. Finally, we presented the Akka toolkit as an example of the actor model and the event sourcing implementation.

4. Scalable application architecture based on CQRS

In this chapter we present the architecture of the application. As it is based on CQRS and event sourcing patterns, we divided it into write model (described in Section 4.1) and read model part (described in Section 4.2). Section 4.3 describes the event store architecture which connects both models.

Each section starts with the discussion about possible design decisions. Even though we bound ourselves to the CQRS+ES architecture we have still quite a few degrees of freedom. We refrained from using a complete CQRS framework as the only mature option at the time was the Axon Framework [42]. That would prevent us from using actor model and Akka toolkit we wanted to evaluate.

4.1. Write model

In order to design the command side of the application we needed to decide what consistency level we want to guarantee, how to scale out the command processing and finally what kind of caching strategy we want to choose to make the write model more responsive. The possible solutions to these issues are discussed in Table 4.1.

Pat Helland in his famous work “Life beyond distributed transactions” [15] introduced the idea of entity, a fine-grained transactional unit in a distributed system. It needs to fit on a single node, may be a subject of resource reshuffling when the application is scaled out. No atomic transaction can span over multiple entities. According to Helland, this approach gives a scale-agnostic programming abstraction and results in ‘almost-infinite‘ scalability. This description resembles both DDD’s aggregate and actor concept. We decided then to design aggregates as actors in the system. Fortunately the problem domain do not require strict ***consistency guarantees*** and it was possible to find fine-grained transactional units. We managed to discover two aggregate types:

– ***rotation*** consisting of a list of legs with the invariant that they do not overlap,

– ***airplane*** with optional rotation assigned.

Table 4.1: Objectives of the write model architecture along with considered means of implementation. The chosen solutions are underlined.

Objective	***Possible solutions***
Consistency guarantees	Strictly consistent model with transactions spanning over multiple entities.
	Fine-grained transactional units eventually consistent with each other.
Scalable processing	Optimistic, multi-master replication with efficient conflicts resolution.
	Processing distribution (sharding) using consistent hashing.
Decreasing latency	Caching events from event store.
	Persisting state snapshots in the event store.
	Caching recreated entities (event sourced state machines).

We made aggregates (actors) event sourced, i.e. they accept commands, validate them, produce events, persist them in the event store and finally switch to a new state. When fetched, a brand new instance is created and all associated events are replayed from the event store. Event ordering is maintained only within a single aggregate. Different aggregate instances are eventually consistent with each other what enables concurrent processing of their commands without any interference, locking mechanism and blocking.

We had two options of providing ***scalable processing***. The first option was to replicate processing and deal with conflicts like eventuate [47] does. In our use case it is impractical as we have very fine-grained transactional units. Partitioning of their processing (sharding) should be elastic enough. Aggregates are assigned to shards by consistent hashing of aggregate identifiers. Each write model node has a region service which knows the shard-to-node assignment and is synchronized with the master region service coordinating the shard assignment process. The command processing flow is presented on Figure 4.1.

We maintain a large number of partitions (shards), at least an order of magnitude more than the number of machines for write model deployment and assign multiple shards to each machine. That allows us to balance the load when a new node is added by transferring shards from each of the previous nodes. Similarly, when we want to deprovision a machine, we transfer all entities to other machines, partitioning them equally. In fact, we change only the shard assignment as every aggregate is persisted in a database and can be easily recreated on a new node.

To ***decrease the latency*** of recreating the aggregates, we considered several caching strategies. We could shorten the replay time either by periodically creating a snapshot of the state or by caching the events in memory. We think that the best option is to cache the recreated state that is ready to serve commands and to passivate it when we need space.

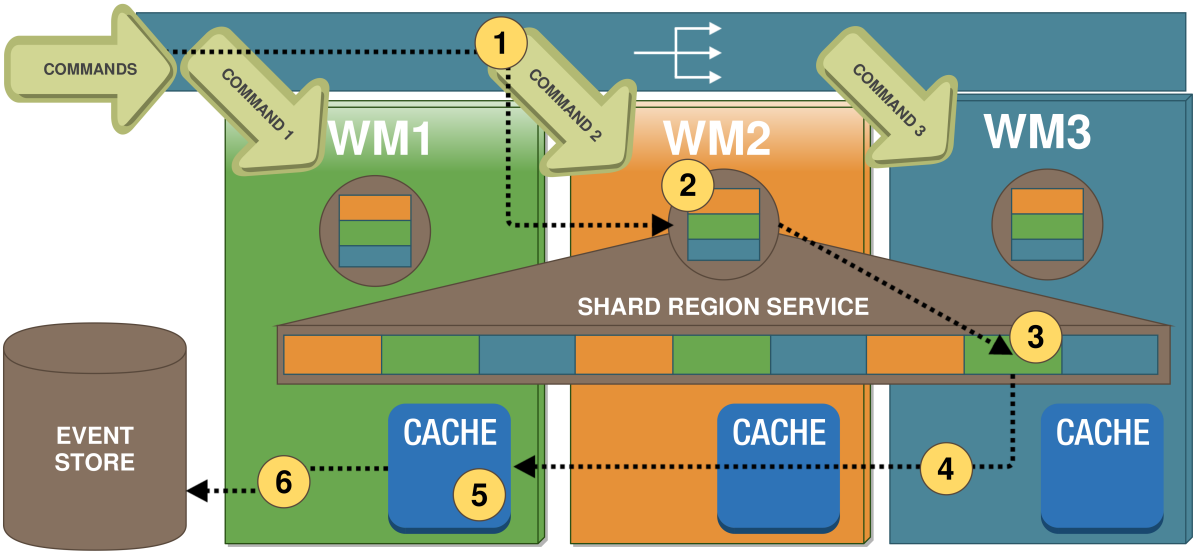


Figure 4.1: Scalable command processing is accomplished with the idea of sharding depicted on the diagram. Requests are dispatched by a round-robin load balancer (1) and hit the shard region service (2) on one of the write model nodes (WM1-3). The region service maintains the shard-to-node mapping with its peers on other nodes. It recognizes the shard the command belongs to (3) and dispatches it to a proper node (4). The responsible node looks up its cache (5) and either returns the cached aggregate or constructs a new instance using past events from the event store. Finally the command is applied, and the generated event is stored (6).

4.2. Read model

For the query part of the application we needed to make several design decisions too. We considered different data models, examined two approaches for rebuilding the model from scratch and finally decided how to scale out the processing (see Table 4.2).

In our case the only queries are the constraint checks which are executed against the loaded flight schedule (described in Chapter 1). These validations require quick graph traversal operations. We considered in-memory

Table 4.2: Objectives of the read model architecture along with considered means of implementation. The chosen solutions are underlined.

Objective	***Possible solutions***
Data representation	In-memory model
	Graph-oriented database
Place of state building during replay	At event store, only final state is transferred to read model
	At read model, all events transferred to read model
Scalable processing	Processing partitioning in a scatter-gather fashion
	Instance replication and round-robin routing

only model first, but finally we decided that a graph-oriented database is the best ***data representation***, as it allows handling of bigger schedules than the available memory of the node and to avoid recreating the state always from scratch. Fortunately, the model could fit into a single disk and that allowed us to get rid of the database distribution. Each read model node manages its own local database instance with a complete model.

When the read model is started, we always recreate it from scratch by replaying all events. We could have used state snapshots but the replay time was not a problem in our case. Due to the same reason we abandoned the idea of building the state close to the event data in rest, directly in the event store and transferring the final state only. We chose a simpler solution in which we transfer all events through the wire and ***build the state in the read model*** during the replay phase. When the replay is finished, the read model subscribes to new events and keeps the model up to date.

The ***scalable processing*** was achieved by replicating the instances of all read models and balancing the load in front of them. When there is a need to increase the processing power on the read side then a new instance of a read model is spawned. We avoid complicated model cloning due to the complete history of events that a new instance can ask the event store for and recreate the current state on its own. When the replay is over, the node joins the load balancer group to start handling requests and subscribes to new events to keep the model up-to-date.

4.3. Event store

Finally, we developed a simple architecture of the event store. We selected a consistency level that we want to guarantee and considered available solutions together with multiple designs combining a database with a durable message queue (see Table 4.3).

We could not achieve horizontal scalability without relaxing ***consistency guarantees*** on the query side. Bridging two models with an asynchronous event store comes with very important consequences. Firstly, the read model instances may slightly differ at a given time since updates are not synchronized. Secondly, when a new command is accepted by the write side there exists a time frame where the data contained in the read model nodes is slightly outdated. Fortunately these effects were acceptable in our use case.

We considered several ***event storage*** solutions. EventStore [48] was the only viable off-the-shelf event store available. It is an efficient and reliable implementation providing complex event processing capabilities. As these features were not required in our use case we decided to assemble a simpler alternative on our own. We rejected the idea of storing events only in a database because that would require either polling on the read side or using sophisticated database change capture tools like LinkedIn Databus [44]. On the other hand, using only a distributed log or persistent message queue would lead to inefficiency in the state recreation and consequently in command handling. Currently leading persistent queues like Kafka [50] do not provide so granular partitioning in which all events of fine-grained aggregates are collocated and easy to find.

Because of this we decided to use both a message queue and a column-oriented database. Every event is both stored in the database and pushed to the queue where it is retained for a while. That makes the read model more

resilient, it guarantees ***at-least-once*** delivery semantics and, most importantly, keeps new events coming to an instance during its initialization with historical events from the database.

This is to make sure that no event is missed during instance startup. The last question was ***which storage is the primary one***. As we did not strive for perfect resiliency in the case of the event store, we went with a simpler solution in which we first store events in the database and later push them to the queue. That may create a problem if a node crashes in between of such an operation. Storing events in the distributed log first and updating the database accordingly later would solve the problem, but also made the implementation more challenging.

Table 4.3: Objectives of the event store architecture along with considered means of implementation. The chosen solutions are underlined.

Objective	***Possible solutions***
Consistency guarantees	Strict consistency
	Eventual consistency
Events storage	Tailor made storage (e.g. EventStore)
	Distributed log (persistent message queue)
	Database only
	Database with a persistent message queue
Primary storage	Persistent message queue
	Database

4.4. Summary

The architecture we proposed in this chapter adheres to the Reactive Manifesto suggestions. We can distinguish all of the required traits:

- ***message-driven*** - commands trigger events, events trigger read model updates,
- ***elastic*** - adding new write model or read model instances results in better performance,
- ***resilient*** - losing an instance of a write or a read model does not prevent the system from working and the lost instance may be easily recreated,
- ***responsive*** - the best query model in terms of the performance was chosen, aggregates in the write model handles commands independently and are easy to cache.

The write model scalability was achieved by creating fine-grained transactional units which are distributed (sharded) in a cluster of machines. The read model was scaled out using the replication of instances. Finally, the event store was designed as a combination of a column-oriented database and persistent message queue.

5. Implementation challenges and choices to ensure scalability

In this chapter we describe our solutions to challenges we were faced with. We also motivate our technology choices and describe the most important implementation details. Section 5.1 presents the language we have chosen and its ecosystem. In Section 5.2 we discuss our load balancing solution. Section 5.3 shows our internal abstraction for writing applications using CQRS and DDD style. Section 5.4 contains the description of storage technologies used by the event store. In Section 5.5 we lay out details of the write model distribution. Next, in Section 5.6 we describe the event filtering which was needed to make our read model idempotent. The last Section 5.7 gives an explanation of the choice of the read model database.

5.1. Language and frameworks

We implemented the application using the Scala language [86]. This decision was driven mainly by our intention to use the Akka toolkit. Even though Akka provides a Java interface, we preferred to stick to the native one. The Scala syntax makes it more expressive and due to the interoperability it is still possible to take advantage of thousands of available Java libraries.

We used the Akka toolkit to implement most of the application behaviour. Actors are processing incoming requests both on the command and on the query side. They are also responsible for dealing with events: they push generated events to the event store (Akka Persistence module), and accept new events on the read side.

Actors are also responsible for handling the HTTP requests/response lifecycle. We expose the REST interface using the Spray [87] library which builds on top of Akka. We deserialize all commands and serialize responses to JSON format through a convenient utility provided by the library. Spray actors handle the HTTP communication and pass messages to other actors in charge of the business logic handling. When we send domain messages, e.g. commands created from HTTP requests, they are serialized by Akka using default Java serializer when they need to cross machine boundaries. However, all internal Akka messages, e.g. the clustering gossip protocol traffic is serialized more efficiently using Protocol Buffers [88].

5.2. Load balancing

The load is balanced by a single Nginx server in front of the read and write models. When a new node is started, it is registered in the load balancer. Originally we wanted to make use of Akka cluster-aware routers. It turned out to be a problem as that led to the design in which we handle HTTP requests orchestration (e.g. connection handling, serialization, deserialization) on a single server. That quickly became a bottleneck.

Using an external load balancer enabled us to delegate (de)serialization to the backend nodes and reduce the overhead of a TCP connection management as we can handle multiple HTTP requests from a single TCP socket. Without using this persistent connection pattern, the backend nodes spend a lot of time opening and closing TCP sockets if each client request comes from a different machine.

In the case of the read model we have single-hop routing. For the write model we can have two hops at most. We did not employed sticky sessions or any other sophisticated routing strategy. We simply evenly distribute commands to all write model nodes and rely on Akka cluster sharding module to deliver commands to the appropriate aggregates. We describe the sharding in Section 4.1 and Section 5.5

5.3. Domain-driven abstraction

Accordingly to the DDD guidelines, we decided to keep the business logic separate from the Akka actors and other infrastructural concerns. The business logic resides in plain, immutable classes implementing POJO interfaces. Every time the state changes, it is replaced by a new object of the aggregate created by the command handler. The aggregate objects are instantiated and used by generic aggregate handlers implemented as Akka actors. Each actor takes care of a single aggregate instance.

The core interface in our abstraction is `AggregateRoot` representing an aggregate. The `handleCommand` method validates the received commands and publishes events to change the aggregate state. The `applyEvent` method reflects the state of the aggregate when the event is published or replied from the store. An aggregate is immutable and the event handler returns a new state instead of modifying it. We also hide the aggregate lifecycle logic by introducing the `AggregateRootFactory` interface together with `AggregateRoot.Removed` null object. We present in Listing 5.1 an example of the business logic from our application which uses this abstraction.

Similarly, we introduced a `DomainView` interface for the read model side (see Listing 5.2). It accepts events to update the state and responds to queries. The infrastructure batches events for efficient updates of the model. If the view does not support it, events are simply consumed one by one. Read models are assumed to be idempotent. If needed, they can obtain event metadata from the `EventEvelope` wrapper, e.g. to filter out duplicate events.

```
case class Rotation(id: UUID, legs: List[FullyDatedLeg])
    extends AggregateRoot[Rotation] {

  override def handleCommand(eventBus: EventBus): CommandHandler = {
    case cmd: AddLegToRotation =>
      ensureNoIntersectingLeg(cmd)
      eventBus.publish(new LegAddedToRotation(cmd))

    case cmd: RemoveLegFromRotation =>
      ensureLegExists(cmd.legId)
      eventBus.publish(new LegRemovedFromRotation(cmd))

    case cmd: RemoveRotation =>
      eventBus.publish(new RotationRemoved(cmd.rotationId))
  }

  override def applyEvent: EventHandler = {
    case event: LegAddedToRotation =>
      val newFullyDatedLeg = FullyDatedLeg.fromLegAddedEvent(event)
      Rotation(id, sortByTime(newFullyDatedLeg :: legs))

    case event: LegRemovedFromRotation =>
      Rotation(id, legs.filter(_.id != event.legId))

    case event: RotationRemoved => removed
  }

  (...)
}
```

Listing 5.1: Rotation aggregate implementation using our domain-driven abstraction.

```
trait DomainView {
  def receiveEvent: PartialFunction[EventEnvelope, Unit]
  def receiveQuery: PartialFunction[DomainQuery, Any]

  def receiveBatch(envelopes: Seq[EventEnvelope]) = {
    envelopes.foreach(receiveEvent(_))
  }
}
```

Listing 5.2: Interface of the read model abstracting the event provisioning logic.

5.4. Event store

Apache Cassandra [89] and Apache Kafka [50] were chosen as the database and the persistent messaging queue respectively for the sake of an event store implementation. Akka Persistence along with the already available Cassandra plugin provided an event store interface for event sourcing purposes. Since the event store was not a bottleneck during our tests, we simplified the implementation into a single Kafka partition to store events and we used single-node deployments of both datastores.

In the time of working on the project, Akka Persistence was not capable yet of querying the event store for all events. We needed to implement our own Cassandra journal reader. A similar functionality is planned to be added to Akka Persistence soon [46].

5.5. Cluster sharding

We implemented the partitioning of the write model processing with the Akka cluster sharding extension. It introduces a concept of a shard which is a group of actors which are always present together on a single node. Entire shards are subject of rebalancing, not single actors. Each node has a `ShardRegion` actor which knows the mapping of the shards to the nodes and it routes messages appropriately. The shard assignment is maintained by `ShardCoordinator`, a single actor in the entire cluster. The uniqueness of this actor in the cluster is assured by a cluster singleton [90] feature of Akka. It is event sourced to resist failures. `ShardRegion` actors reach out to this actor when they do not know where to route the message. The responses are cached, so they do not need to involve the coordinator again during the dispatch process.

ShardRegions need to be provisioned with consistent correlation between messages, actors and shards. It comes in a form of two functions presented in Listing 5.3. The first extracts the persistence identifier of the recipient actor from the message, the latter groups those identifiers into shards. As actors are backing aggregates, that results in routing commands to appropriate, sharded aggregates. Our command IDs are UUIDs [91], so we simply bucket them into a predefined number of shards. It is suggested to use about ten times more shards than the expected number of machines, as there is no way to change it dynamically in production yet. We also enhanced the cluster sharding with the recognition of factory commands which creates new aggregates to handle them in a special way. They do not have a destination identifier, so we intercept those commands and enhance them with a generated one which ends up to be the identifier of a new aggregate ultimately.

Moreover, cluster sharding gives us a cache layer for aggregates. Aggregate actors remain in the memory after being recreated and there is no need to fetch events from the database when a next event arrive. They may be passivated on demand to avoid running out of memory. We decided to use the UUID implementation [91] from the standard Java library, even though it may cause performance issues due to the fact it is using a cryptographically safe random number generator.

```
ClusterSharding(system).start(
  typeName = "Counter",
  entryProps = Some(Props[Counter]),
  idExtractor = idExtractor,
  shardResolver = shardResolver)

val idExtractor: ShardRegion.IdExtractor = {
  case EntryEnvelope(id, payload) => (id.toString, payload)
  case msg @ Get(id)              => (id.toString, msg)
}

val shardResolver: ShardRegion.ShardResolver = msg => msg match {
  case EntryEnvelope(id, _) => (id % 10).toString
  case Get(id)              => (id % 10).toString
}
```

Listing 5.3: Using cluster sharding requires nothing more than provisioning mapping between messages, actors and shards and making actors persistent. The example is taken from the Akka documentation.

5.6. Event filtering

Our at-least-once event delivery semantics forces read models to be either idempotent or to filter out duplicate events. We created a utility class for filtering events. If we want to make the read model able to recover from the point it stopped consuming events instead of starting from the beginning each time, we need to persist the state of the filter. Moreover, this state should be persisted transactionally together with the read model updates. For that reason we left the persistence implementation of the filter to the user. The state of the filter is represented as a set of the last seen event sequence number for each aggregate.

5.7. Graph database

We decided to use Neo4j [92] as a graph database for the schedule validation read model. We tried out three available ways of writing the queries:

- ***Cypher language***, a declarative graph query language that allows for expressive and efficient querying and updating of the graph store provided by the Neo4j,
- ***Spring Data Neo4j*** [93], which enables using a template programming model popularized by the Spring [94] library amongst Java developers,
- ***native Java API*** provided by the Neo4j.

We ended up using the Java API both for persistence and for performing the feasibility checks. The performance of the Cypher language and the Spring Data Neo4j library were not satisfying and we rejected them, even though they are much easier to use in some cases. We also leveraged Neo4j ACID transactions to make the event filter state consistent with the schedule representation.

Each read model instance manages a separate Neo4j process in the embedded mode what simplified the deployment. The alternative mode is a standalone server instance. As the read model could fit on a single instance, we preferred to avoid the separation. Besides the simplicity, we get a better performance, because in the separate deployment we would need to communicate through REST protocol, possibly crossing network boundary.

5.8. Summary

In this chapter we described the most important implementation decisions: Scala language, Akka toolkit, Nginx, Apache Cassandra, Apache Kafka and the Neo4j database. We presented crucial features of our domain-driven framework which separates completely the business logic from the infrastructural concerns. We discussed some of its implementation challenges that affects scalability: aggregate sharding, event replay and event deduplication.

6. Scalability tests

This chapter presents the evaluation of scalability of our architecture and implementation. In Section 6.1 we describe the deployment model and the environment the application was running in during the tests, together with monitoring tools. Next, we show in Section 6.2 what actions we took to ensure that the results are not biased. Sections 6.3 and 6.4 present how we approached the evaluation and what results we got for each part of the application architecture respectively, as we decided to test those parts separately. In Section 6.5 we comment the results and draw conclusions from them. Finally, Section 6.6 lists some of the problems we faced during the tests.

6.1. Deployment and monitoring infrastructure

We deployed our application to the cloud environment provided by GWDG [95], a data processing research centre from Germany which takes part in PaaSage project together with Lufthansa Systems and AGH University of Science and Technology. The environment is based on the OpenStack [96] platform. The actual deployment of the components varied between the tests and is described in the respective sections. The Nginx load balancer was deployed in all cases on a separate 4 core, 4GB machine, similarly to the monitoring infrastructure residing on a single 4 core, 4GB machine with a HDD. The general overview of the deployment is depicted on Figure 6.1. All machines were located in the same cloud zone. We presents a more detailed description in the Appendix C. We were using the SBT [97] build tool for packaging of the application modules and managing the dependencies. The infrastructure setup and application deployment was automatized using Ansible [98] playbooks.

We leveraged several tools to monitor the application during testing. We were using Akka Clustering JMX endpoints to supervise the state of the cluster, JMX endpoints of the embedded Neo4j server to configure and monitor the database and native JVM JMX endpoints to see the memory layout, i.e. heap and garbage collection state. Kamon [99] was used to provide Akka-related metrics which allows us to see the current state of each actor system. It was also used to collect system metrics like CPU or memory usage thanks to kamon-system-metrics extension that uses the Sigar [100] library and to expose some custom metrics, e.g. number of rotations in the system. All metrics are gathered by using the statsd [101] collector and are persisted in the influxdb [60] time-series database. Finally, we displayed the metrics data in the form of real-time charts using the Grafana [62] dashboard. We used also the VisualVM [102] tool for ad-hoc inspection of metrics exposed by the JMX interface. During the testing we gathered over 20GB of metrics data grouped in 17,000 unique data series.

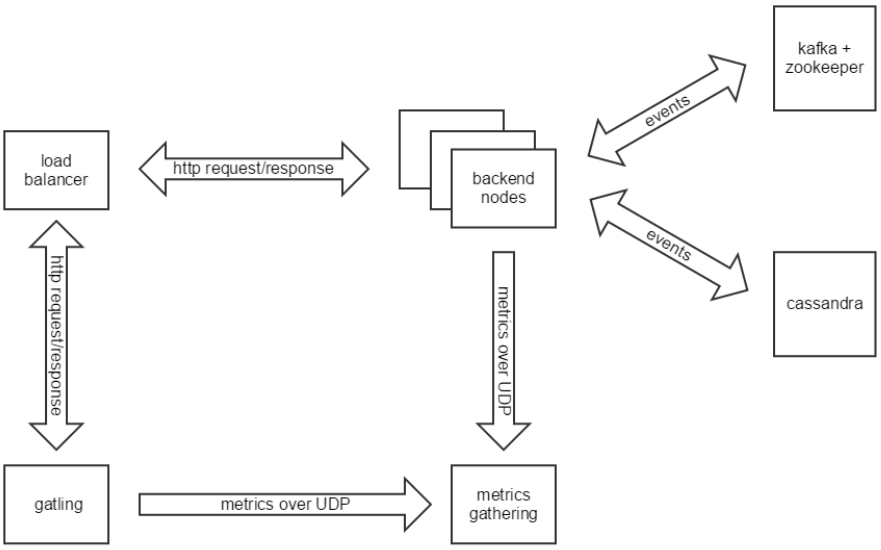


Figure 6.1: The diagram shows an overview of the application deployment in the cloud environment. Each box is a separate virtual machine. In the case of the read model scalability tests that changed slightly as we were running Kafka and Cassandra on the same VM.

6.2. Sanity of the load tests

We have chosen the Gatling [103] tool to perform the experiments and integrated it with the previously installed monitoring tools. That enabled to correlate load testing results (e.g. number of requests per second, response time) with the application resources behaviour (e.g. CPU usage, memory usage). For instance, thanks to that we were able to see if the load is being evenly distributed among the application instances. Figure 6.2 shows a sample of the Grafana dashboard in action during one of the load tests.

In order to assert the sanity of the experiments we carefully warmed up both the application and the load testing tool before each test. It is really crucial with any JVM applications, as the runtime environment is optimizing the bytecode during several thousands of the initial iterations. We explain the impact of the warming up in Figure 6.3. The warmup method varied between the tests and they are described in the respective sections. We also cared to warm up Kafka, Cassandra and the load testing tool before every run. We repeated each test case five times to obtain more accurate results. We restarted both the application and Gatling between every iteration. Moreover, we used the JHiccup [104] tool to detect any jitter from the underlying infrastructure that could result in biased results.

We decided to evaluate the scalability of read and write part of the application separately. We designed two different workloads, the first contains only queries and the latter executes commands alone.

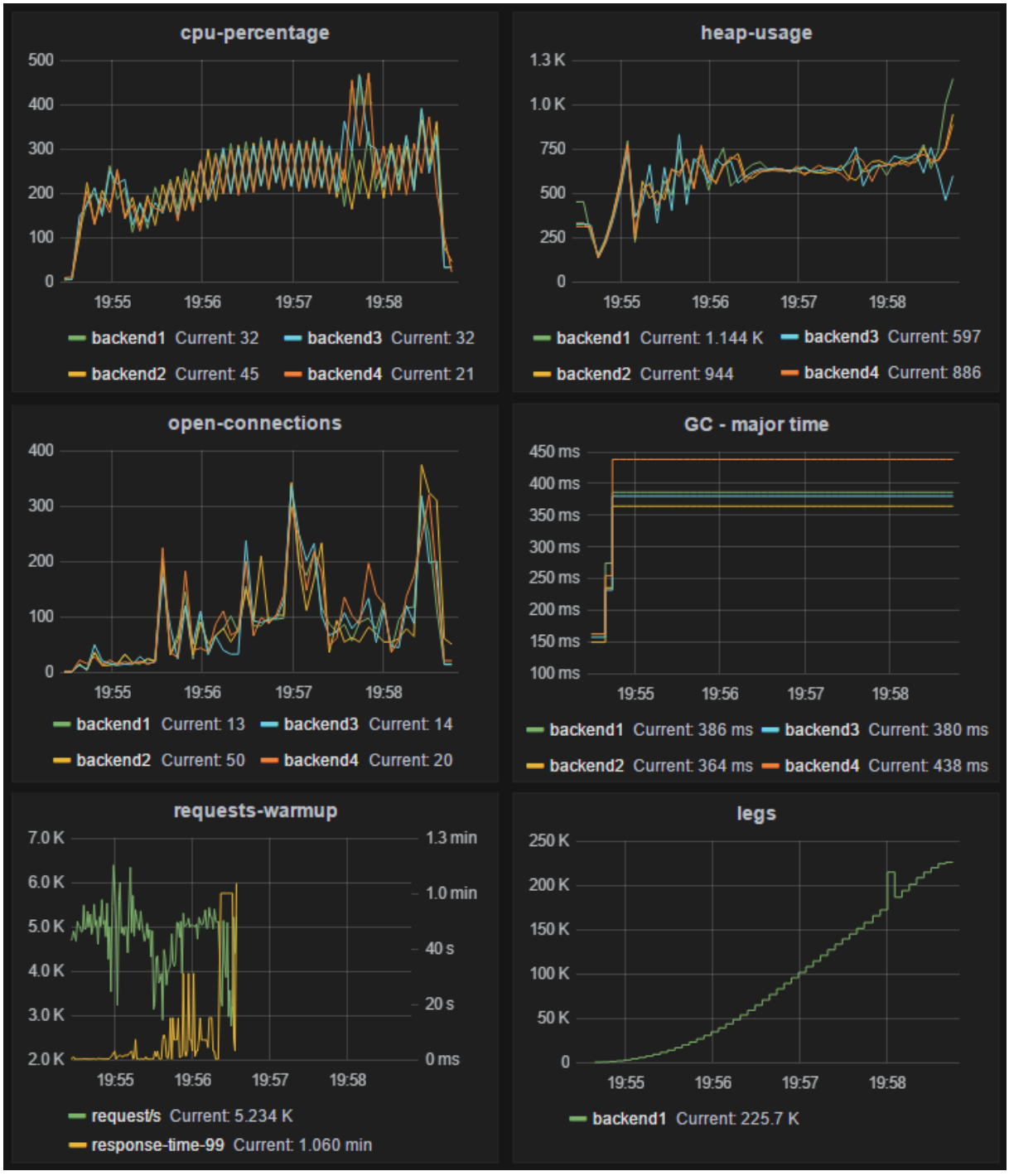
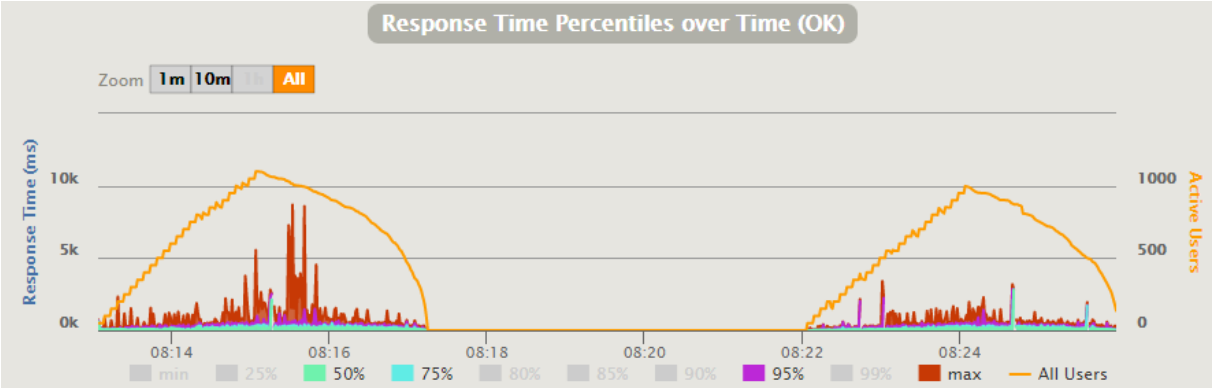
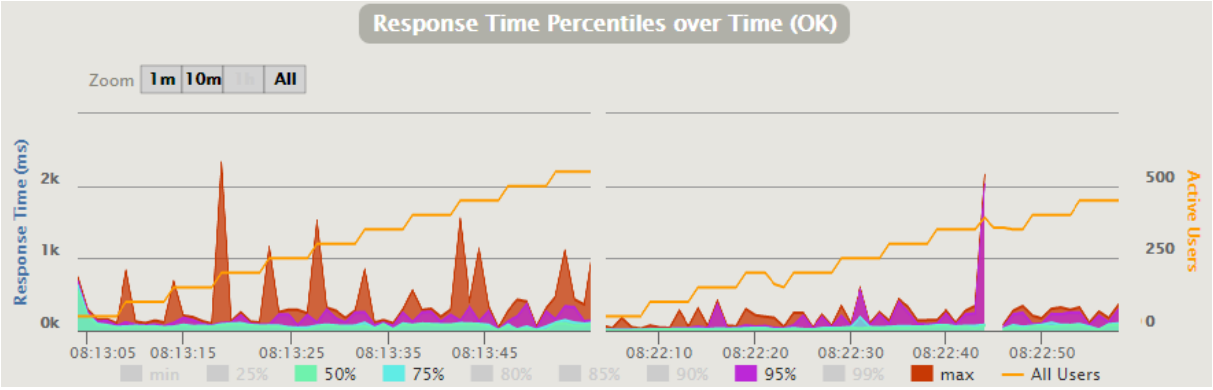


Figure 6.2: This is a fragment of the metrics dashboard captured during one of the write model scalability tests. We combined a variety of metrics from different sources like JVM metrics (e.g. CPU usage, memory usage, garbage collection), load testing metrics (requests per second, response time percentiles) and application metrics (e.g. total number of legs). The correlation of such data in time helped us with the investigation of problems and the verification of the correct behaviour under load. That was especially useful in conjunction with the separation of metrics coming from different application instances. In this example they are labeled as backend[1-4]. The full dashboard had 34 graphs in total.



(a) Two identical workloads run one by one on a fresh instance



(b) Zoom-in on a first 40 seconds of both runs

Figure 6.3: Here we show the warmup effect on the test results. In the top figure (a) we can see a single workload run. The first part (between 8:13 and 8:18) is a warmup phase, the latter (starting from 8:22) is a real test phase. We can see clearly the difference in response time between the two caused by the fact that during the first run the application was not warmed up. In the bottom figure (b) we see the beginning of each workload. The first seconds of the non-warmed up run reveal how big difference the warming up makes as the median of the response time drops four times just after a few seconds of receiving requests by the application.

6.3. Read model scalability evaluation

6.3.1. Evaluation method

Firstly, we decided to found the maximum sustainable request rate for a single instance of a model. In order to achieve that, we designed a workflow as follows: we start with 1 request/sec and ramp it up to the maximum number of requests/sec (e.g. 200 requests/sec) during 300 seconds, then we hold the maximum rate for 600 seconds. We treat the first part of the workload as a warmup phase and only the last 300 seconds are taken into consideration to calculate final results. The requests were distributed evenly between all the three types of the schedule validation checks: rotation continuity, minimal ground time and flight designator uniqueness.

We run this workflow with a goal of 125, 150, 175, 200, 225, 250 and 275 requests per second. For each of the values we repeated the test five times. We measured 75th, 95th, 99th and 100th (maximum) percentiles of the response time for every run. We also wanted to make sure that the CPUs, as the most demanded resource by the read model are heavily loaded.

After we learned the sustainable throughput of a single instance, we wanted to run the workload against the setups with multiple read model instances. We set the frequency of the requests respectively to the capacity of a single node multiplied by the number of instances. We wanted to see if the application can linearly increase its capabilities. If the latency does not change between the runs on the different setups, that will prove the linear scalability.

We reserved 12 machines with HDDs, 4 cores and 4GB of memory each, for running the read model instances. An additional machine of the same configuration was used to deploy the event store components: Kafka, Cassandra and Zookeeper (required by Kafka). The Gatling load testing tool was running on a single 4 core machine with 16GB memory.

6.3.2. Single instance capacity

We run the capacity test 35 times in total which is 9 hours of aggregate load and gathered 2.5GB of raw logs. The maximum throughput that we considered sustainable was achieved when the load testing tool was generating 175 queries per second. In this case, each of CPU cores was 75% utilized on average. All the results are charted in Figure 6.4 and presented in Table 6.1.

Request rate [req/s]	125	150	175	200	225	250	275
Average 75th latency percentile [ms]	51	51	51	56	847	1683	1612
Average 95th latency percentile [ms]	59	66	85	131	1383	2249	2086
Average 99th latency percentile [ms]	99	100	132	194	1625	2533	2626
Average maximum latency [ms]	380	429	295	410	2598	3723	3907
Average CPU utilization [%]	220	250	300	340	380	400	400
Number of all requests	37.1k	44.6k	52.0k	59.4k	66.7k	74.2k	81.6k
Number of failed requests	0	0	0	0	0	0	0

Table 6.1: The results of the capacity testing shows that the sustainable request rate correlates closely with the CPU utilization, as it is the most important resource for the read model. 100% means that a single core is fully utilized. As the machines were equipped with four cores processing units, 400% is the maximum possible utilization.

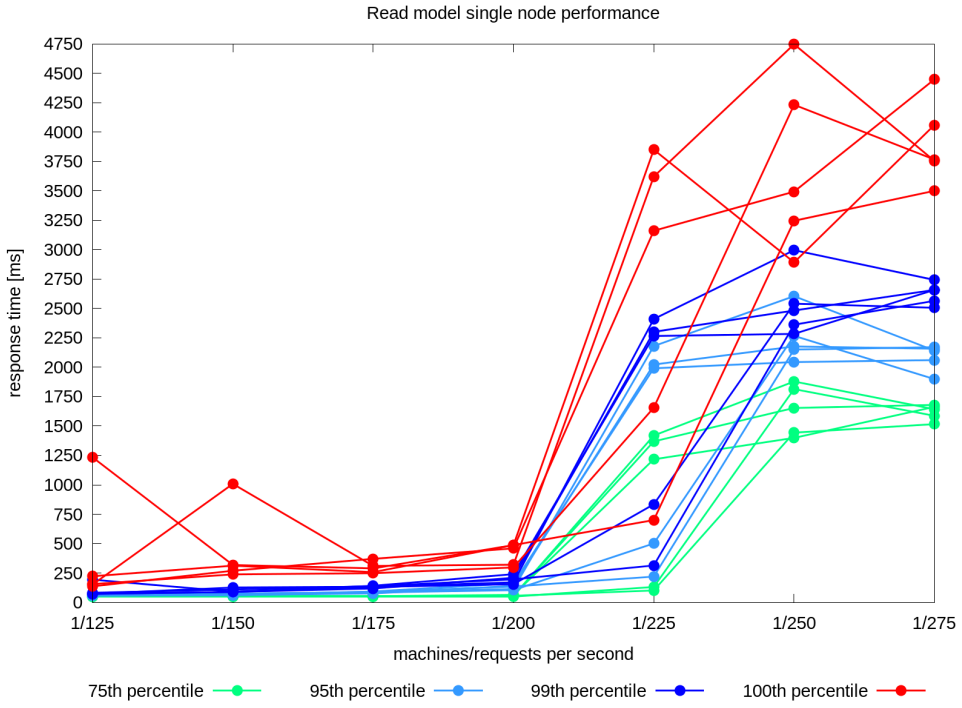


Figure 6.4: The figure presents the capacity testing results of a single read model instance. We can see that the query rate of 125-200 request/sec is acceptable but it changes for higher ones.

6.3.3. Read model linear scalability

Finally, we conducted the scalability tests. We measured the response time percentiles for setups with 1, 2, 4, 8 and 12 machines. Every machine was hosting a single read model instance. The tests were repeated several times in order to verify the reproducibility.

The results are shown in Figure 6.5 and also in Table 6.2. There is almost no difference between the request latencies, only the maximum response time (100th percentile) has greater variance which was in fact expected. It is extremely hard to bound the maximum response time, as even a single occurrence of any unexpected event like packet loss contributes to this metric. But 99% percent of all requests had a nearly identical latency upper bound in each run. That means the read model scales very well, in a linear fashion. We made sure that we selected the appropriate query rates by verifying the CPU utilization on each node. Every of the four cores on each node during the tests was 75% utilized nearly all the time.

Number of nodes (instances)	1	2	4	8	12
Query rate [req/s]	175	350	700	1400	2100
Average 75th latency percentile [ms]	52	51	51	50	51
Average 95th latency percentile [ms]	91	89	86	84	91
Average 99th latency percentile [ms]	145	139	137	134	145
Average maximum latency [ms]	318	416	519	446	481
Average CPU utilization [%]	300	300	300	300	300
Number of all requests	51.9k	104k	208k	416k	624k
Number of failed requests	0	0	0	0	0

Table 6.2: The table shows amongst others that the query generation frequency was well-suited as the CPUs were heavily loaded. 400% was the maximum possible utilization (4 cores $\times$ 100%).

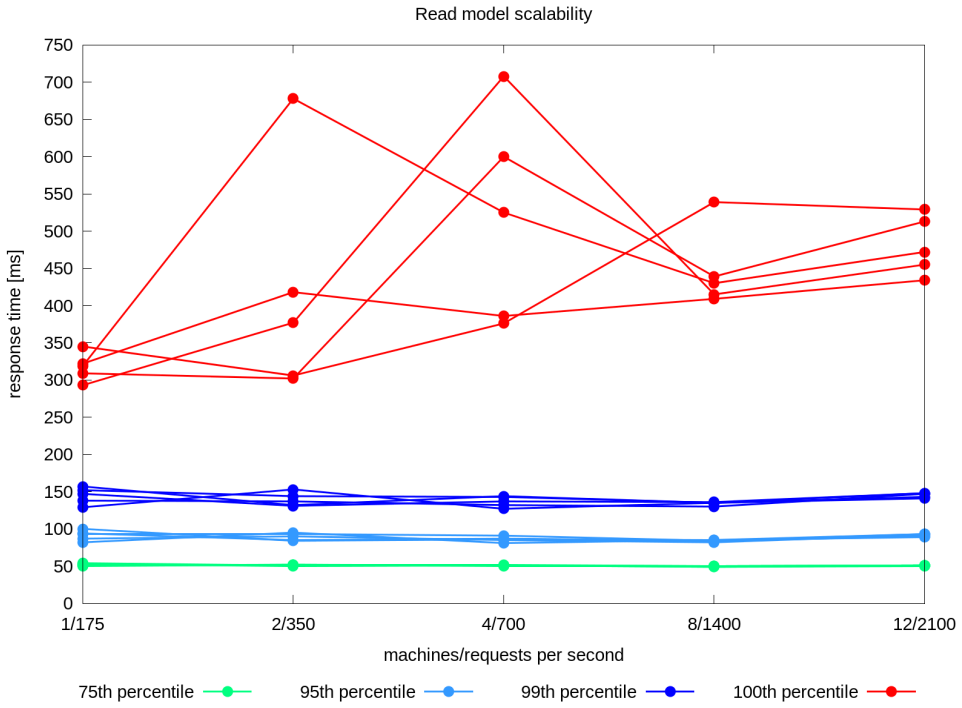


Figure 6.5: No difference in the request latency time proves the linear scalability of the read model. The slight increase for 12 nodes may suggest that for the greater number of nodes we could be limited by a single load balancer throughput. The figure was generated from 9GB of raw logs produced by 25 test runs lasting for 15 minutes each (over 6 hours of load testing).

6.4. Write model scalability evaluation

6.4.1. Evaluation method

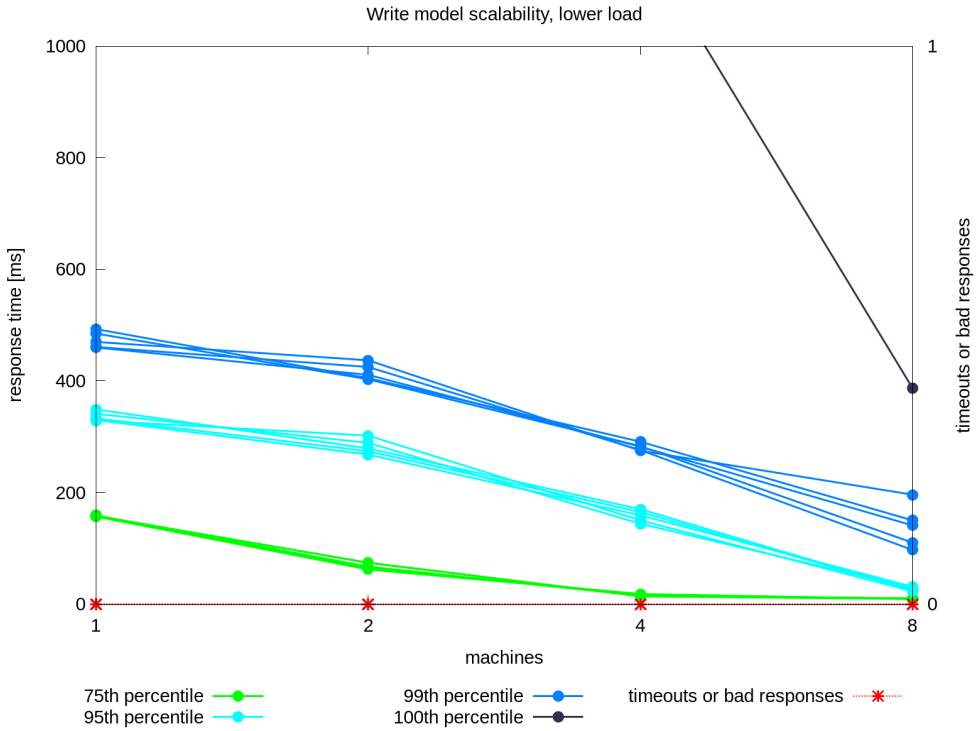
In order to evaluate the write model, we took a different approach. We decided to prepare a fixed command workload and run it against the setups with a different number of the write model instances. We wanted to choose a suitable frequency of requests to be able to saturate setups with one or two instances. We were interested how the response time, the CPU utilization and the number of time-outed requests changes when new instances are being added to the setup.

The workload is a simulation of multiple users performing the same scenario. First, they create a rotation, next they add several hundred legs to the rotation, then they create an airplane and finally they assign the rotation to the airplane. We ramped the number of users, adding 50 new users every 5 seconds until we started all of them. If any bad response occurred, the user scenario was interrupted. Figure 6.3 displays the progressive growth of active user sessions during one of our tests. In order to warm up the instances, we were running the same workload twice in every test case iteration separated by a five minute pause (see Figure 6.3). The first run was considered a warmup phase and only the latter was taken into consideration when the results where calculated.

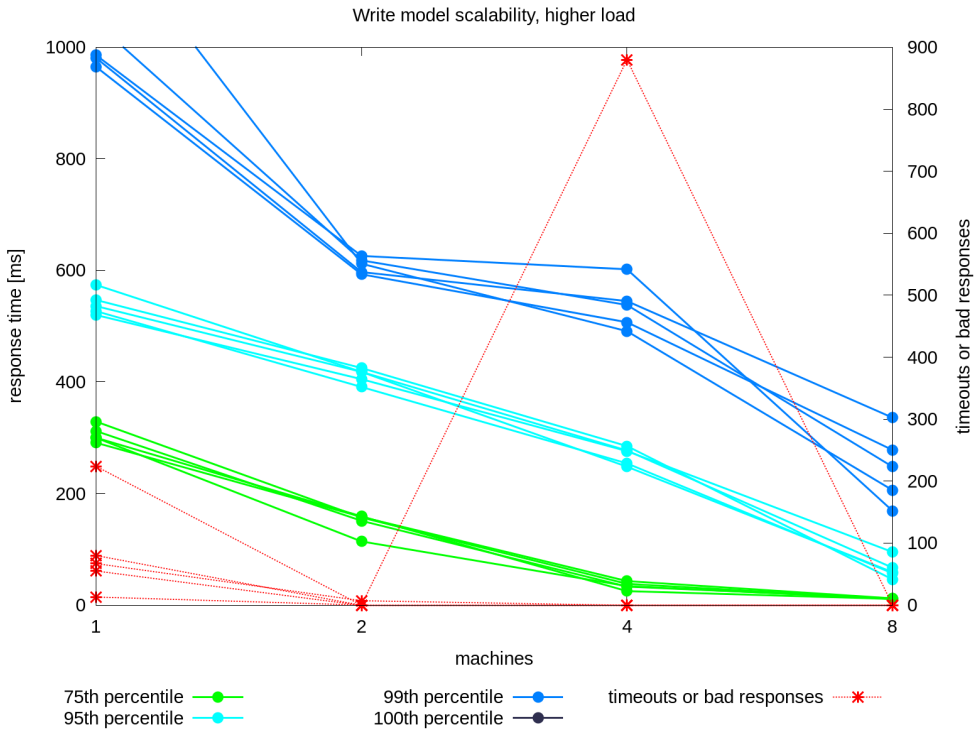
We reserved 8 machines with 4 cores and 4GB of memory each, for running the write model instances. In this case, we deployed event store on two 16 cores, 16GB machines. Kafka with Zookeeper were placed on the first one and Cassandra on the latter. Finally, we were generating the load on a single 4 core machine with 16GB memory.

6.4.2. Write model scalability

We present the results both in Table 6.3 and in Figure 6.6. We conducted the evaluation with two different scenarios: 1250 users adding 750 legs each and 1500 users adding 1000 legs each. We refer to them as lower and higher load scenarios, respectively. We can see that the response time of requests drops when instances are added, similarly the CPU utilization decreases linearly with the number of nodes. In the higher load scenario we can see timeouts indicating that the load is too big for the single instance. They disappear as the number of nodes increases.



(a) Results for slightly lower load



(b) Results for slightly higher load

Figure 6.6: Both figures prove the scalability of the write model architecture. Plot (b) depicts timeouts occurring in a single-instance deployment. As we add more nodes, the timeouts disappear. A deviation in the number of timeouts for a 4-node deployment visible on plot (b) was caused by an unrecognized, transient problem with the Cassandra database. It did not happen again in any other test so we treat it as a noise. We skipped the 100th percentile for better clarity.

Scenario (req/s)	lower load				higher load			
Number of nodes (instances)	1	2	4	8	1	2	4	8
Average 75th latency percentile [ms]	157	68	15	10	306	148	34	11
Average 95th latency percentile [ms]	336	282	158	26.2	541	411	268	64.8
Average 99th latency percentile [ms]	474	416	281	139	1042	609	537	247
Average maximum latency [ms]	2026	1397	1642	2300	15013	7254	9282	3355
Average CPU utilization [%]	400	350	300	200	400	350	300	200
Average command frequency [req/s]	4073	4444	4709	4528	3377	4154	4321	4220
Average number of all requests	941k	941k	941k	941k	1466k	1504k	1456k	1505k
Average number of failed requests	0	0	0	0	87.8	1.4	220	0

Table 6.3: The results proves the write model scalability. The table lays out the linear decline of the CPU utilization as more instances of the write model are used. The difference in average command frequencies comes from the excessive latency of some requests. A single user was simulated in a complete synchronous way, sending a next request after the previous was completed. Another reason is the fact that failed requests were terminating the user session and that is why higher load scenario has lower average requests rate even though it generated higher momentary load and more requests in total.

6.5. Conclusions from the tests

The tests we have conducted proved that the CQRS+ES architecture is scalable. Our implementation is able to keep the same response time of the read model queries under increasing load when the nodes where added respectively. We measured that a single instance of the read model handles at most 175 queries per second without getting into troubles. We managed to handle 2100 (12 * 175) requests per second using 12 instances of the read model keeping the same level of latency. That proves the linear scalability of the read model.

We are also able to decrease the overall latency of command requests in the write model and the CPU utilization by adding new instances of the model. The load we generated overwhelmed a single-instance deployment. Two instances were able to handle it with 87% of CPU utilization, but when we switched to 8 nodes, the utilization dropped to 50% and 99% of the requests were handled three times faster.

In the case of the read model, the slight increase of latency may indicate that we are close to the limit of a single load balancer. That means we would need to add a second level of load balancing (e.g. DNS-based) to be able to scale out even more. We did not test the write model scalability with more than 10 nodes but usually the real workloads have a ratio of queries to commands between 10 and 100. The scalability level we achieved is satisfactory for the flight scheduling application and we expect it will behave similarly for other applications of the architecture.

6.6. Encountered problems and their solutions

During the testing phase, we were faced with several issues impacting our test results. The first one was already mentioned in the Section 5.2. The setup with the Akka application-level load balancer was inefficient due to frequent opening and closing of HTTP connections. Next, we tried out HAProxy [105]. That helped significantly but in this case we encountered occasional, indeterministic problems with persistent connections limiting in the round-robin routing strategy. As we were not able to eradicate them, we switched to Nginx and the problem disappeared. We described the problem in more detail in Stack Overflow question [106]

We experienced how much the tuning of low-level JVM and operating system options influences the test results. Garbage collection (GC) pauses in the Gatling load-testing tool impacted the reported response time of the requests. Because of that, we struggled to minimize it as much as possible by experimenting with different GC options. Similarly, we experimentally chose the best heap and GC configuration for read and write model instances. Finally, we bumped onto a problem of connection socket limiting and need to increase the limit of allowed open descriptors. We list the detailed JVM, OS and Nginx configuration description in Appendix C.

The conducted stress tests brought out several implementation issues we were not aware of before. One of them was the contention caused by atomic counters in the Neo4j object cache. The profiling of the read model revealed that the threads spend most of the time competing for access to atomic counters used in the cache implementation. We ultimately ended up with turning this cache off. We described the problem in more detail on Neo4j user group [107].

6.7. Summary

In this chapter we presented the empirical evaluation results of the CQRS+ES architecture scalability. We deployed our application to a cloud environment consisting of 20 VMs and we set up monitoring tools. We load tested several configurations of the deployment varying in the number of read/write model running instances after meticulous assessment of the testbed sanity. We tested the read and the write side separately using different methods. For the read model we scaled the load and expected no change in the response time and for the write model we had a constant load and expected response time to improve. The results show that both the read and the write model scale very well.

7. Conclusions and future work

This chapter contains the summary of the thesis. In Section 7.1 we revisit the objectives defined in Section 1.3. Then, Section 7.2 presents the results of the study and conclusions. In Section 7.3 we discuss lessons learned. Finally, in Section 7.4 we outline possible ways to continue our research on CQRS+ES architectures.

7.1. Goals achieved

Let us revisit all of the thesis targets and summarize what we managed to do. Below we discuss each of the objectives of the thesis which were defined in Section 1.3 and explain how they were achieved.

✔**Discuss the recent architectural patterns for building scalable systems**

We provided a comprehensive state-of-the-art survey of the event-driven architectures. We presented all industrial attempts we are aware of to leverage CQRS and event sourcing concepts. Finally, we described in detail CQRS, event sourcing, domain-driven design, the reactive manifesto and the Akka toolkit.

✔**Propose the architecture of the flight scheduling application using CQRS+ES**

We designed the application based on the CQRS+ES architecture. We scaled out the write model using the idea of sharding. The read model was scaled out by leveraging the event-state duality and eventual consistency. We discussed the event store design which bridges the read and write models.

✔**Experimentally assess the scalability of the proposed solution**

We carefully tested the horizontal scalability of both the write and read part of the application. We deployed our application to the cloud environment, set up monitoring and load-testing infrastructure and finally got a positive answer to the main question about the architecture scalability.

✔**Deliver the industrial business use case for PaaSage project**

Our application was accepted by the project consortium as a demonstration of the capabilities of the PaaSage platform. We managed to successfully prove the scalability in a live demo session.

All of that means that we managed to successfully fulfill the objectives we set ourselves at the beginning. We are certain that it would not be the case if the thesis was done by a single person.

7.2. Results of the Study

The major goal of this thesis was the assessment of the CQRS+ES architecture scalability. We managed to successfully approach this problem. Firstly, by an appropriate design, next in the implementation of the flight scheduling application and finally, by an experimental performance evaluation.

The results presented in this thesis are favourable to the concepts related to event sourcing. The field research we conducted shows that there is currently an increasing interest trend in this area. We found many reasons arguing in favor of this approach and discerned the great advantages it brings. We got a very positive impression after our hands-on experiences. We learned that the actor model, Domain-Driven Design and the CQRS+ES architecture have a lot in common and together they create a great toolkit for development teams to build highly reliable and scalable applications which can easily handle complex business logic.

We conclude that the CQRS+ES architecture proves to be both a useful and a scalable solution for building event-driven business applications. Moreover the Akka toolkit and the actor model approach appear to be a very appropriate tool for putting the CQRS+ES idea into practice.

7.3. Lessons learned

Similarly to any emerging technique, the CQRS+ES approach still lacks widely adopted patterns and best practices. There are a lot of things you have to decide or invent yourself. There are also many problems that still prevent those solutions from becoming production ready in many cases, e.g. lack of support for event versioning in akka-persistence or the akka-clustering vulnerability to split-brains. On the other hand, we see how rapidly these tools evolve and in fact, the aforementioned problems should be resolved very soon [46].

We learned that performance measurement is not straightforward. The reason is not only the unintuitive statistical characteristics of the requests latency [108] but also the fact that distributed systems are composed of many unreliable parts: slow processes, failing disks, many virtual machines sharing the same CPU, sloppy network connections, etc. That makes the experiments sanity and proper results interpretation a non-trivial task. We know now that in this kind of environment the automatized deployment is crucial. There are too many components to manage them manually.

CQRS, like many other event-driven approaches, forces us to accept and understand weaker consistency models. We need to reset our minds in terms of consistency and replace a linear one with an eventual. It requires both to understand the business logic behind the application better and to become more careful as more things can go wrong (e.g. duplications, losses, retries). Moreover, we experienced the fact that distribution introduces a lot of complexity to the architecture design, no matter how good the design is.

7.4. Future work

We consider a handful of ideas for our further research. First of all, we would like to improve the event store we have created, especially focusing on its scalability and reliability. We want to try out different types of databases to store events, especially those with a support for publish-subscribe queues.

Next, we think about implementing different use cases using the domain-driven abstraction we have created in our project. We want to make sure it is generic enough to be useful. We consider implementing these interfaces with different tools, e.g. Axon with the EventStore database to compare them with our own solution. Moreover, we want to update the current implementation when the support in Akka Persistence for CQRS is done. This should simplify the framework significantly.

We are also interested in stream processing systems in the context of CQRS+ES architecture. Read model building and event filtering seems to be a perfect use case to apply this concept. We would like to evaluate causality ideas presented by COPS [24] and eventuate [47] to merge different streams of events in a sane manner. Finally, we want to significantly improve the model building speed leveraging the concept of doing that close to the data in rest, already used in EventStore [48] and Apache Samza [51].

Bibliography

[1] "PaaSage: Model-based Cloud Platform Upperware. Project website: http://www.paasage.eu/."

[2] 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, 1st ed., 2013.

[3] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 2003.

[4] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "The reactive manifesto." Published online: http://www.reactivemanifesto.org/, September 2014.

[5] C. Hewitt and H. G. Baker, "Actors and continuous functionals," in *Proceeding of IFIP Working Conference on Formal Description of Programming Concepts*, August 1977.

[6] G. Hohpe, "Programming without a callstack - event-driven architectures." Published online: http://www.enterpriseintegrationpatterns.com/docs/EDA.pdf, 2006.

[7] D. C. Schmidt, "Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Software Pattern Series, ch. 29, pp. 529–545, Reading, Mass.: Addison-Wesley, 1995.

[8] "Node.js. Project website: http://nodejs.org/."

[9] "NGINX. Project website: http://www.nginx.com/."

[10] "Netty. Project website: http://netty.io/."

[11] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[12] "Vert.x Core Manual - Verticles. Available online: `http://vertx.io/docs/vertx-core/java/#_verticles`."

[13] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," tech. rep., 2011.

[14] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 230–243, Oct. 2001.

[15] P. Helland, "Life beyond distributed transactions: an apostate's opinion," in *3rd Biennial Conference on Innovative DataSystems Research (CIDR)*, January 2007.

[16] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[17] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, (Washington, DC, USA), pp. 245–256, IEEE Computer Society, 2011.

[18] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[19] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, (New York, NY, USA), pp. 7–, ACM, 2000.

[20] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, pp. 20:20–20:32, Mar. 2013.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, (Berlin, Heidelberg), pp. 386–400, Springer-Verlag, 2011.

[22] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a calm and collected approach.," in *In Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR)*, pp. 249–260, January 2011.

[23] C. Meiklejohn and P. Van Roy, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, (New York, NY, USA), pp. 184–195, ACM, 2015.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 401–416, ACM, 2011.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 251–264, USENIX Association, 2012.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[27] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[28] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 2001.

[29] "Storm, distributed and fault-tolerant realtime computation. Project website: http://storm.apache.org/."

[30] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, (Washington, DC, USA), pp. 170–177, IEEE Computer Society, 2010.

[31] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 423–438, ACM, 2013.

[32] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, pp. 1033–1044, Aug. 2013.

[33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2008.

[35] "Twitter Inc. Streaming MapReduce with Summingbird. Available online: https://blog.twitter.com/2013/streaming-mapreduce-with-summingbird," September 2013.

[36] "Microsoft Open Technologies Inc. Rx - Reactive Extensions. Project website: https://rx.codeplex.com."

[37] "Typesafe Inc. Akka Streams documentation. Available online at: http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0/scala/stream-index.html."

[38] "Typesafe Inc. Reactive Streams." Published online: http://www.reactive-streams.org/, 2014.

[39] N. Marz, "How to beat the CAP theorem. Available online: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html," October 2011.

[40] "ElephantDB: Distributed database specialized in exporting key/value data from Hadoop. Project website: https://github.com/nathanmarz/elephantdb."

[41] "Druid: Open Source Data Store for Interactive Analytics at Scale. Project website: http://druid.io/."

[42] A. Buijze, "The Axon Framework. Project website: http://www.axonframework.org."

[43] "Datomic: The fully transactional, cloud-ready, distributed database. Project website: http://www.datomic.com/."

[44] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, S. Nagaraj, D. Zhang, L. Gao, J. Westerman, P. Ganti, B. Shkolnik, S. Topiwala, A. Pachev, N. Somasundaram, and S. Subramaniam, "All aboard the databus!: Linkedin's scalable consistent change data capture platform," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, (New York, NY, USA), pp. 18:1–18:14, ACM, 2012.

[45] "Typesafe, Inc. Akka Persistence documentation. Available online: http://doc.akka.io/docs/akka/snapshot/scala/persistence.html."

[46] "Typesafe, Inc. Akka Roadmap Update. Avaiable online: www.typesafe.com/blog/akka-roadmap-update-dec-2014," December 2014.

[47] M. Krasser, "The Eventuate toolkit. Project website: http://rbmhtechnology.github.io/eventuate," January 2015.

[48] G. Young, "Event Store: The open-source, functional database with Complex Event Processing in JavaScript.. Project website: https://geteventstore.com."

[49] J. Kreps, "The Log: What every software engineer should know about real-time data's unifying abstraction. Available online: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying," December 2013.

[50] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[51] "Linkedin Corp. Apache Samza. Project website: http://samza.apache.org."

[52] J. Kreps, "Questioning the Lambda Architecture. Available online: http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html," July 2014.

[53] "Facebook Inc. Scribe. Project website: https://github.com/facebookarchive/scribe/wiki," 2008.

[54] "Elasticsearch BV. Logstash: Collect, Parse, Transform Logs. Project website: https://www.elastic.co/products/logstash."

[55] "Apache Flume. Project website: https://flume.apache.org/."

[56] "Apache Hadoop. Project website: https://hadoop.apache.org/."

[57] "Elasticsearch BV. Elasticsearch: RESTful, Distributed Search & Analytics. Project website: https://www.elastic.co/."

[58] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," tech. rep., Google Inc., 2010.

[59] "Twitter, Inc. Distributed Systems Tracing with Zipkin. Available online: https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin," June 2012.

[60] "InfluxDB: An open-source distributed time series database with no external dependencies. Project website: https://influxdb.com/."

[61] "Whisper: a file-based time-series database format for Graphite. Available online at: https://github.com/graphite-project/whisper."

[62] "Grafana: An open source, feature rich metrics dashboard and graph editor for Graphite, InfluxDB & OpenTSDB. Project website: http://grafana.org/."

[63] "Elasticsearch BV. Kibana: Explore, Visualize, Discover Data. Project website: https://www.elastic.co/products/kibana."

[64] "MuleSoft Inc. Mule ESB. Project website: https://www.mulesoft.com/platform/soa/mule-esb-open-source-esb."

[65] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 2003.

[66] "Apache Camel. Project website: http://camel.apache.org/."

[67] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[68] "AngularJS — Superheroic JavaScript MVW Framework. Project website: https://angularjs.org/."

[69] "Backbone.js. Project website: http://backbonejs.org/."

[70] "Oracle Corp. Java FX. Project website: http://docs.oracle.com/javafx/."

[71] "Microsoft Corp. Introducing Windows Presentation Foundation. Available online: https://msdn.microsoft.com/en-us/library/aa663364.aspx."

[72] M. Fowler, "Presentation Model. Available online: http://martinfowler.com/eaaDev/PresentationModel.html," July 2004.

[73] "Data binding. Wikipedia definition available online: `https://en.wikipedia.org/wiki/Data_binding`."

[74] "Facebook, Inc. React: A JavaScript library for building user interfaces. Project website: http://facebook.github.io/react/."

[75] "Facebook, Inc. Flux: Application Architecture for Building User Interface. Project website: http://facebook.github.io/flux/."

[76] V. Vernon, *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st ed., 2013.

[77] D. Wyatt, *Akka Concurrency*. USA: Artima Incorporation, 2013.

[78] B. Meyer, *Object-Oriented Software Construction*. Upper Saddle River, NJ, USA: Prentice-Hall Inc., 1st ed., 1988.

[79] G. Young, "DDDD 10 - CQS. Available online: http://codebetter.com/gregyoung/2008/04/28/dddd-10-cqs/," April 2008.

[80] G. Young, "Command Query Separation?. Available online: http://codebetter.com/gregyoung/2009/08/13/command-query-separation/," August 2009.

[81] "Basho Technologies, Inc. Riak KV: Distributed NoSQL Database. Project website: http://basho.com/products/riak-kv/."

[82] Z. W. Hendrikse and K. Molkenboer, "A radically different approach to enterprise web application development." Published online: http://www.codeboys.nl/white-paper.pdf, January 2012.

[83] M. Fowler, "Event Sourcing. Available online: http://martinfowler.com/eaaDev/EventSourcing.html," December 2005.

[84] "Typesafe, Inc. Akka toolkit. Project website: http://akka.io/."

[85] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[86] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*. USA: Artima Incorporation, 1st ed., 2008.

[87] "Typesafe, Inc. Spray: Elegant, high-performance HTTP for your Akka Actors. Project website: http://spray.io/."

[88] "Google, Inc. Protocol Buffers: language-neutral, platform-neutral extensible mechanism for serializing structured data. Project website: https://developers.google.com/protocol-buffers/."

[89] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[90] "Typesafe Inc. Akka Cluster Singleton documentation. Available online: http://doc.akka.io/docs/akka/snapshot/scala/cluster-singleton.html."

[91] "Oracle Corp. Java 8 java.util.UUID class documentation. Available online: http://docs.oracle.com/javase/8/docs/api/java/util/UUID.html."

[92] "Neo Technology, Inc. Neo4j: the World's Leading Graph Database. Project website: http://neo4j.com/."

[93] "Spring Data Neo4j. Project website: http://projects.spring.io/spring-data-neo4j/ ."

[94] "Pivotal Software, Inc. Spring. Project website: https://spring.io/ ."

[95] "GWDG: IT in science. Institution website: `http://www.gwdg.de/index.php?&L=1`."

[96] "OpenStack: Open Source Cloud Computing Software. Project website: https://www.openstack.org/."

[97] "Typesafe Inc. SBT: The interactive build tool. Project website: http://www.scala-sbt.org/."

[98] "Ansible: Simple IT automation. Project website: http://www.ansible.com/."

[99] "Kamon: The Open Source tool for monitoring applications running on the JVM. Project website: http://kamon.io/."

[100] "VMWare Hyperic. Sigar: System Information Gatherer And Reporter. Project website: https://github.com/hyperic/sigar."

[101] "Etsy, Inc. Statsd: Simple daemon for easy stats aggregation. Project website: https://github.com/etsy/statsd."

[102] "Oracle Corp. VisualVM: All-in-One Java Troubleshooting Tool. Project website: https://visualvm.java.net/."

[103] "Gatling Project: Stress Tool. Project website: http://gatling.io."

[104] G. Tene, "jHiccup. Project website: https://github.com/giltene/jHiccup."

[105] "HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. Project website: http://www.haproxy.org/."

[106] "HAProxy: Not reusing connection - Stack Overflow question. Available online: http://stackoverflow.com/questions/32277981/haproxy-not-reusing-connections."

[107] "Neo4j object cache under high load and removal of object cache in 2.3.0 - Neo4j discussion group. Available online at: `https://groups.google.com/forum/#!topic/neo4j/SDUrl9O_260/discussion.`."

[108] G. Tene, "Understanding Latency: Some Key Lessons & Tools. Presentation available online: http://www.infoq.com/presentations/latency-lessons-tools," July 2014.

[109] "Gatling SBT plugin. Project website: https://github.com/gatling/gatling-sbt."

[110] "Typesafe, Inc. Akka 2.4.0-RC2 Released!. Avaiable online: http://akka.io/news/2015/09/04/akka-2.4.0-RC2-released.html," September 2015.

A. Source code of the DDD framework

We open-sourced the domain-driven abstraction implementation described in Section 5.3. The code is available on GitHub hosting service under the following URL: `https://github.com/cqrs-endeavour/cqrs-endeavour`.

We published the most important part of the application we have built. The framework reflects the architectural design of the write model, the read model and the event store that we introduced in Chapter 4. It includes the solutions to challenges we presented in Chapter 5. The unique feature of this abstraction is a complete separation of the business logic from Akka actors and other infrastructural concerns.

B. Division of work

The thesis is a result of our joint effort with a similar commitment share. We designed and implemented the application together. The state-of-the-art research was performed by Bartłomiej Szczepanik. Andrzej Dębski prepared the deployment and conducted the load testing.

In terms of the thesis text, Chapters 1, 2 and 3 were written by Bartłomiej Szczepanik. Chapters 5 and 6 come from Andrzej Dębski. Chapters 4 and 7 were prepared together.

C. Details of the testing infrastructure

We deployed our application to a cluster of virtual machines in a cloud environment. Every VM instance was running Ubuntu 12.04.4 64-bit with a 3.11.0-26-generic kernel version. Detailed information about the processing units of the underlying infrastructure can be found on Listing C.1. The operating system configuration in terms of resource management is presented on Listing C.2. Listing C.3 shows the network interface configuration of the virtual machines. The presented configuration applies to every virtual machine in our cluster.

The Nginx load balancing service was configured accordingly to the Listing C.4. That means it is reusing the connections to the backend nodes.

To run our Scala applications, the load generator, Kafka and Cassandra we used the standard Oracle Java™SE Runtime Environment (build 1.8.0_60-b27) with Java HotSpot™64-Bit Server VM (build 25.60-b23, mixed mode). We tuned a lot our applications to achieve the best performance. We used the following JVM options for the read model:

- -XX:+AggressiveOpts,
- -Xms3584M,
- -Xmx3584M,
- -XX:SurvivorRatio=12,
- -XX:NewSize=2560M,
- -XX:MaxNewSize=2560M,
- -XX:+UseParallelOldGC.

For the write model:

- -XX:+AggressiveOpts
- -J-Xms3200M

– -J-Xmx3200M

– -J-server

– -J-XX:+UseG1GC

– -J-XX:-HeapDumpOnOutOfMemoryError

And for the Gatling instance:

– -XX:+UseThreadPriorities,

– -XX:ThreadPriorityPolicy=42,

– -XX:+HeapDumpOnOutOfMemoryError,

– -XX:+AggressiveOpts,

– -XX:+OptimizeStringConcat,

– -XX:+UseFastAccessorMethods,

– -Xms3G (-Xms8G in the case of the write model tests),

– -Xmx3G (-Xmx8G in the case of the write model tests),

– -Djava.net.preferIPv4Stack=true,

– -Djava.net.preferIPv6Addresses=false,

– -XX:+UseG1GC.

Most of items of the last list comes from the gatling-sbt plugin [109]. We also experienced that the new G1 garbage collector from Java 8 is performing very well in the case of this load testing tool.

```
processor       : 3
vendor_id       : AuthenticAMD
cpu family      : 21
model           : 1
model name      : AMD Opteron 62xx class CPU
stepping        : 2
microcode       : 0x1000065
cpu MHz         : 2299.998
cache size      : 512 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall
                  nx mmxext fxsr_opt pdpe1gb lm rep_good nopl
                  extd_apicid pni pclmulqdq ssse3 cx16 sse4_1 sse4_2
                  popcnt aes xsave avx hypervisor lahf_lm cmp_legacy
                  svm cr8_legacy abm sse4a misalignsse 3dnowprefetch
                  osvw xop fma4 arat
bogomips        : 4599.99
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

Listing C.1: Detailed processing units information of machines we used for testing (/proc/cpuinfo content).

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31469
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65535
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31469
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Listing C.2: Configuration of OS resources on machines used for testing (ulimit -a command).

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
net.ipv4.tcp_max_syn_backlog = 4096
net.core.somaxconn = 4096
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
```

Listing C.3: Network interface configuration of machines used for testing (sysctl.conf).

```
keepalive 24;
keepalive_requests 600000;
worker_processes 4;
worker_connections 16000;
use epoll;
multi_accept on;
worker_rlimit_nofile 32768;
```

Listing C.4: Configuration of Nginx load balancer.

D. Akka Persistence Query

As we described in Section 3.5 and Section 5.4, at the time we were designing and implementing the application Akka Persistence was not yet adapted to handle the read part of our CQRS architecture. We implemented our own workarounds for this issue, for instance we query the Cassandra database directly to fetch the archived events.

Just before we finished writing the thesis, the release candidate of the new Akka 2.4 version was published at the beginning of September 2015 [110]. It introduces a large number of improvements and several new modules. From our perspective, the most important changes are:

- Promoting the Akka Persistence, Cluster Singleton and Cluster Sharding modules to be fully supported. They were treated experimentally so far.
- Improvement in Akka Persistence, e.g. better resiliency to failures, Persistent Finite State Machine (FSM) domain-specific language for creating persistent actors and support for schema evolution and migration of events.
- Improvement of Cluster Sharding, e.g. graceful shutdown and asynchronous shard allocation.
- Introduction of experimental Akka Persistent Query, a module responsible for covering the read side of CQRS.

We would have definitely benefit of these features if they had appeared earlier. Especially the Akka Persistence Query module is really interesting, as it solves the problems we needed to fix on our own. It exposes the Akka Streams API for journals which is designed in a very loose fashion to cover all types of possible underlying event stores. Journal plugins can decide what types of queries they want to support or how they want to execute them. For instance they can benefit from the streaming capabilities of the journal or fall back to polling if it is not the case. There are three types of queries available: `AllPersistenceIds`, `EventsByPersistenceId` and `EventsByTag`. The last one is a generic way to obtain partial streams, e.g. if only events from a single aggregate type are needed. Events are tagged using *Event Adapters* which also facilitate events schema migration. We present an usage example of the Akka Persistence Query on the Listing D.1.

We are very happy that we have chosen the concepts that serious companies are investing in and the technology that keeps being updated and extended with new features.

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val readJournal = PersistenceQuery(system).readJournalFor(JournalId)
val dbBatchWriter: Subscriber[immutable.Seq[Any]] =
  ReactiveStreamsCompatibleDBDriver.batchWriter

val readJournal = PersistenceQuery(system).readJournalFor(JournalId)

readJournal
  .query(EventsByTag("rotations"))
  .map(envelope => envelope.event)
  .map(convertToReadSideTypes) // convert to datatype
  .grouped(20) // batch inserts into groups of 20
  .runWith(Sink(dbBatchWriter)) // write batches to read-side database
```

Listing D.1: An example of using Akka Persistence Query to maintain a read model database in a batch fashion taken from the Akka 2.4.0-RC2 documentation. The example makes an assumption that the destination database exposes Reactive Streams [38] interface.

E. Publications and presentations

We present the list of publications related to the thesis which we co-authored together with our supervisor and engineers from Lufthansa Systems GmbH & Co. KG. Their full content is included below. As the second paper is still not finished, we enclose its draft. We plan to send this paper to IEEE Software journal soon.

- Bartłomiej Szczepanik, Andrzej Dębski, Maciej Malawski, Stefan Spahr, Dirk Muthig. *Towards a Scalable Architecture of Cloud Application Based on CQRS and Event Sourcing*. In: M. Bubak, M. Turała, K. Wiatr (Eds.), Proceedings of Cracow Grid Workshop - CGW'14, October 27-29 2014, ACC-Cyfronet AGH, 2014, Krakow, pp. 79–80 (2014).
- Bartłomiej Szczepanik, Andrzej Dębski, Maciej Malawski, Stefan Spahr, Dirk Muthig. *Scalable Reactive Architecture of Cloud Application Based on CQRS and Event Sourcing* (in preparation).

We have also presented our work at several venues to 150+ software developers, architects and researchers. We got immense number of thoughts and suggestions from them which helped us to improve the design of the system.

- *On the bleeding edge: Highly scalable, distributed, cloud-based architecture using Scala, Actor Model and RESTful Web Services*. Presentation for the AGH Department of Computer Science and AGH Cyfronet research center (Krakow, 3rd June 2014).
- *On the bleeding edge: Highly scalable, distributed, cloud-based architecture using Scala, Actor Model and RESTful Web Services*. Presentation of the progress and discussion with Lufthansa Systems software architects (Raunheim/Frankfurt, 30th June 2014).
- *Lufthansa Systems Flight Scheduling application: Scalable architecture with Akka and CQRS.* Presentation of the industrial use case provisioning progress for the PaaSage project in front of PaaSage board members with a live demo of the application scalability (Krakow, 10th September 2014).
- *Towards a Scalable Architecture of Cloud Application Based on CQRS and Event Sourcing*. Presentation at Cracow Grid Workshop 14' (Krakow, 28th October 2014).
- *DDD/CQRS/ES with Akka - Case Study*. Presentation at DDD-KRK community meetup in front of 80+ software developers (Krakow, 5th March 2015).

Thanks to the presentations and the immense number of discussions with many people we feel we have contributed to the dissemination of the CQRS and event sourcing concepts in the local community.

Towards a Scalable Architecture of Cloud Application Based on CQRS and Event Sourcing

Bartłomiej Szczepanik[1], Andrzej Dębski[1], Maciej Malawski[1], Stefan Spahr[2], Dirk Muthig[2]

[1] AGH University of Science and Technology
Department of Computer Science, al. Mickiewicza 30, 30-095 Kraków, Poland
[2] Lufthansa Systems AG, Frankfurt, Germany
emails: {bszczepa,adebski}@student.agh.edu.pl, malawski@agh.edu.pl,
{stefan.spahr,dirk.muthig}@lhsystems.com

Keywords: Cloud, Scalability, CQRS, Akka, Event-Sourcing, Domain Driven Design

1. Introduction: flight scheduling application in PaaSage

Cloud technologies, due to their capability of dynamic scaling of computing resources on demand, provide new opportunities for development of scalable applications. PaaSage platform [1], based on model driven engineering (MDE) approach, aims at facilitating development of applications that can be deployed across multiple clouds. To fully benefit from these capabilities, the application architectures need to be designed with the scalability as a main design objective. In this paper, we describe our experience with the prototype of a flight scheduling application that has been developed to meet this goal. The (commercial) application provides rich functionality for airline schedule planning tasks and needs to be designed as a responsive and resilient, cloud enabled application. The prototype implements only some basic business functionality of such a real scheduling application, but focuses more on scalability, elasticity and flexible deployment to demonstrate these new architectural styles.

2. The proposed architecture based on CQRS

We decided to conform to a quite new and promising concept called Command-Query Responsibility Segregation (CQRS) [2]. Although based on well-known patterns, by putting them together CQRS enables a number of advantages, including scalability. Up to our knowledge, there is currently no off-the-shelf CQRS framework enabling scalability. We decided to assemble a prototype of such a framework using the following building blocks: write model, a special variant of an event-bus, and read-models, each of them having different requirements. The architecture is shown in Fig. 1.

CQRS interplays with two other ideas that help building the write model: Event Sourcing - the refreshed version of well-known commit log pattern, now considered as a full-fledged persistence method, and Domain-Driven Design which defines Domain Events and Aggregate terms perfectly fitting to the previous concepts. Since the common factor of them are events, we went towards Event-Driven Architecture in a new flavour called Reactive Programming. Akka toolkit [3] was chosen, because it provides complete solutions for Event Sourcing (akka-persistence) also in a variant that scales (akka-cluster-sharding) and is fully event-driven. For the underlying event store the Cassandra database is used.

The read side of the application is loosely constrained by the architecture. Events from the other part are consumed on a regular basis, denormalized and stored in a model crafted for a specific use case to get the best responsiveness. In our case, the model is implemented using Neo4j database [4]. CQRS approach allows scaling read models by running multiple of them in parallel and balance the load, using Akka toolkit's clustering and routing features.

The most challenging part was the event-bus bridging the two parts of the system. The requirement is to provide a way to both publish new events and enable read models to read all the events from the start of the application. Moreover, the solution has to work efficiently in a distributed environment. We managed to implement it using a Kafka messaging system [5] by coupling it with Cassandra database.

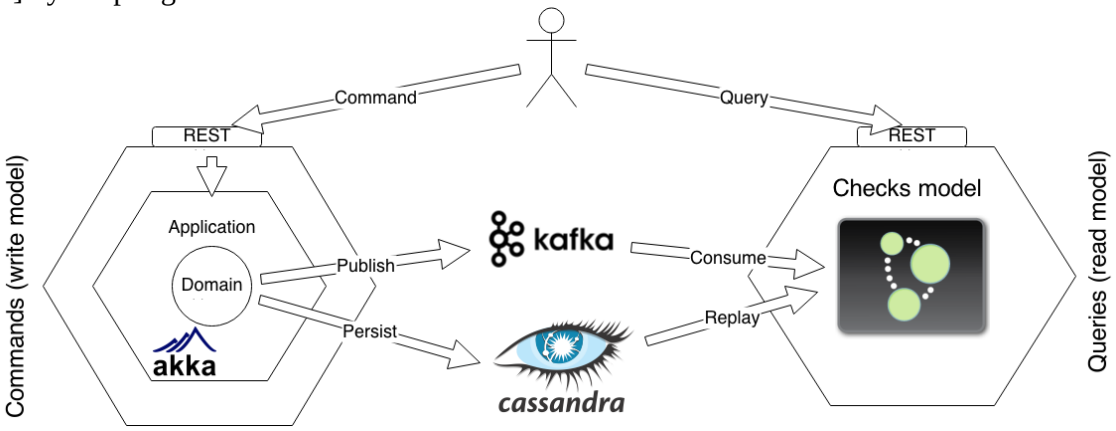


Fig. 1. Overview of the architecture based on separation of commands and queries.

3. Experience with prototype and performance results

In our preliminary tests we focused on proving that this architecture actually scales. We set up a cluster of several machines running the application on GWDG Open Stack infrastructure and load tested it using the Gatling tool [6]. As the load (HTTP requests) was constantly increasing we were adding new machines and new read models there. The tests results indicate the architecture scales really well: the read model side scales linearly, we managed to handle five more times of load using five more machines.

During the work we experienced many new technologies that are not production ready yet, however, they still make the implementation easier in comparison to the already settled up methods. The most problematic is the lack of good developer and monitoring tools for them. Nonetheless, we believe they are on a good way to get a foothold in the industry soon.

4. Conclusions and future work

The developed prototype of a flight scheduling application demonstrates that the architecture combining CQRS with Event Sourcing and Akka framework allows building a fully scalable application. The future work includes more performance tests, including write model scalability. Moreover, in line with the development of PaaSage platform we plan to use the prototype for evaluation of the model-based approach and its autoscaling capabilities.

Acknowledgments: This work was supported by EU PaaSage project (grant 317715).

References

1. The PaaSage project website: http://www.paasage.eu/
2. Martin Fowler, CQRS, 14 July 2011, http://martinfowler.com/bliki/CQRS.html
3. Akka toolkit website: http://akka.io/
4. Neo4j website: http://www.neo4j.org/
5. Apache Kafka website: http://kafka.apache.org/
6. Gatling tool website: http://gatling.io/

Scalable Reactive Architecture of Cloud Application Based on CQRS and Event Sourcing

Andrzej Debski, Bartlomiej Szczepanik, Maciej Malawski, *AGH University of Science and Technology, Poland,*
Stefan Spahr and Dirk Muthig, *Lufthansa Systems, Germany*

***Abstract*—Distributed system design flourishes. The industry competes in the arm race, producing constantly lots of new promising concepts like Command-Query Responsibility Segregation and Event Sourcing. We wondered if they have a chance to survive. In todays world, there is no point trying to convince software architects to a concept that does not scale well. Thus, we wanted to clear any doubts whether the discussed patterns have a future or not. We propose an architecture of the application using those patterns accordingly to Reactive Manifesto guidelines, implement it and experimentally answer the key question. Here we show that Command-Query Responsibility Segregation and Event Sourcing patterns are indeed scalable. That means that we should quickly see their adoption by the industry, as they provide many interesting advantages without performance trade-offs.**

Keywords*—*Scalability, CQRS, Event Sourcing, Domain-Driven Design, Reactive, Akka.

I. Introduction

CLOUD technologies, due to their capability of dynamic scaling of computing resources on demand, provide new opportunities for development of scalable applications. To fully benefit from these capabilities, the application architectures need to be designed with the scalability as a main design objective. That enables developers to create low-latency solutions which handle millions requests per second and adjust resource usage to their current needs.

Due to the fact that this type of software is quite recent, we still do not have frameworks or architectural patterns widely accepted to be a standard in scalable distributed systems development. Moreover, new ideas are being proposed all the time and there is no consensus on any specific direction so far. In this paper, we focus particularly on a movement called *Reactive Manifesto* [1], together with Command-Query Responsibility Segregation (CQRS) [2], a new architectural pattern for low-latency systems, and Domain-Driven Design (DDD) [4], a software design approach originating from the same community as CQRS.

The main goal of our work was to answer the question whether it is possible to implement a fully working, scalable CQRS application in a completely reactive fashion. We decided to build a prototype of a typical application using these ideas and the Akka toolkit [3] as a concrete technology. The prototype implements only some basic business functionality and focuses more on scalability, elasticity and responsiveness to demonstrate these new architectural styles.

As an example of a concrete problem we addressed a simplified version of flight scheduling application. Reactive and scalable application architectures are important aspects of airline related software. Beside the schedule planning software example there are other applications, like for flight operation, passenger handling or baggage tracing etc. Existing client/server or n-tier application stacks suffer mostly from (nearly) unbounded scalability available in today's available Cloud environments. The application maintains a schedule that is updated by planners and allows them to perform schedule validity checks when they are done. Schedule is comprised of rotations with assigned airplanes. A single rotation consists of one or more legs (flights). Each airport defines standard ground time which is the minimum time that airplane have to spend on ground between consecutive legs. To ensure the schedule validity, one can check if all legs in a rotation hold continuity property, do not violate standard ground times and flight numbers are not duplicated.

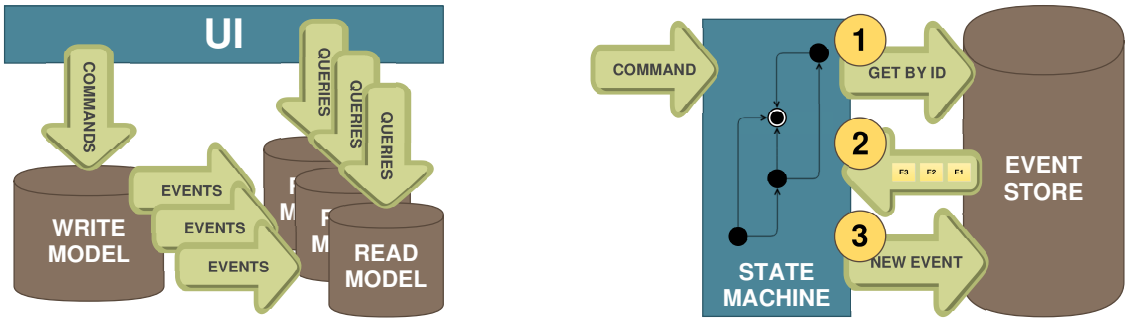
The main contributions of this paper are:

- we discuss the recent architectural patterns for scalable systems, with the focus on reactive applications,
- we propose a prototype architecture of a flight scheduling application using a novel reactive approach that combines CQRS with DDD principles,
- we evaluate experimentally the scalability of the proposed solution in a cloud environment.

II. Evaluated concepts

CQRS principle advises separation of operations mutating state (commands) from queries. That enables a lot of useful possibilities, e.g. ability to chose different database for write and read operations (see Fig. 1a). That means you can select the most performant one for the querying use case without losing advantages of original (e.g. relational) database for state mutation operations. Furthermore, you can optimize each of your query separately by maintaining several different read models at once. Those benefits come with a cost of synchronization of multiple data models.

CQRS plays really well with the **Event Sourcing** idea, the refreshed version of commit log pattern, well-known to database designers, now considered as a full-fledged persistence method. The system behavior is modeled in terms of facts (events) that happened and state machines, not just the state representation (see Fig. 1b). This enables us to abandon the cumbersome object-relational mapping and deliver fully persistence-agnostic model of the system. Moreover, we get for free a complete audit log of the system. It is especially useful when combined with the CQRS architecture. We can add new read models long time after we deploy a write model and still be able to make use of all user actions that happened so far.



(a) (b)

Fig. 1: The 1a diagram depicts the idea of CQRS. Write model handles all user commands, validate them and produces events. Those events are published to all read models which updates their query models and become ready to handle new queries. Event Sourcing, depicted on the 1b diagram, introduces a slightly different approach than the standard flow of dealing with entities in an enterprise world is: deserialize, mutate state, serialize. First the storage is asked for a list of all events for a given entity (1), then the brand new state machine applies all retrieved events (2). Finally, the user command is validated and if was successful, new event is produced and stored (3).

We can also always change our mind about the projections shape and simply recreate them from scratch. It is worth to notice that maintenance of such a commit log is possible with a storage providing append-only operations. Databases designed for event sourcing scenarios are called event stores.

Domain-Driven Design is a software development approach meant to deal effectively with complex and evolving systems. It defines both high-level guidelines for large system design (strategic patterns) and class-level building blocks for business logic modelling. The core rules suggest:

- total separation of business logic (domain model) from any technical concerns,
- cooperation of domain experts with the development team and
- definition of a common language between them (ubiquitous language) which is then used across all artifacts, e.g. in codebase or documentation.

Strategic patterns in DDD focus on a good partitioning of the system into manageable and coherent pieces in terms of business logic (bounded contexts). Those pieces are then prioritized by business value what enables effective architecture selection and human resources utilization. Tactical patterns introduce a level of abstraction that helps experts and developers to reason about the codebase in terms of business processes and behavior instead of classes and state. For example, the most interesting ones are:

- *aggregate* - defines a transactional unit in a system,
- *domain* event - records the actual facts that happened,
- *saga* - defines complex business processes.

Reactive Manifesto suggests a different approach for building scalable and responsive systems. It defines basic traits of a reactive application: elasticity, responsiveness, resiliency and message-driven approach. It suggests also techniques to achieve all of them. According to the manifesto it leads to more flexible, loosely-coupled, scalable and failure tolerant systems.

Akka toolkit helps building applications in the reactive approach. It is a middleware that enables communication between entities in the system using message-passing style. It implements the idea of the actor model [5] introduced many years ago by Carl Hewitt and popularized by Erlang community. The toolkit provides many useful capabilities like cluster membership service, location transparency of actors, different routing strategies, supervision of actors and an implementation of the reactive streams specification (Rx).

III. Application Architecture

We have divided our application into write model and read model accordingly to CQRS principle. Both parts are connected with an event store. Even though we bound ourselves to the CQRS+ES architecture we have still quite a few degrees of freedom. Table I presents the main design objectives and decisions that we analyzed.

The problem domain do not require strict **consistency guarantees**. That enabled us to divide the **write model** into small pieces of transaction scope, influenced by Pat Hellands idea of entities [6] (defined as a prerequisite for a good scalability) and DDD aggregate pattern. Aggregates, as we call them, are event sourced, i.e. they accept commands, validate them, produce events, persist them in the event store and finally transition to a new state. When fetched, brand new instance is created and all associated events are replayed from the event store. Event ordering is maintained only within a single aggregate. Different aggregate instances are eventually consistent with each other what enables concurrent processing of their commands without any interference, locking mechanism and blocking. We defined two aggregate types in our system along with their commands: rotation (a list of legs which do not overlap) and airplane (that handles a specific rotation).

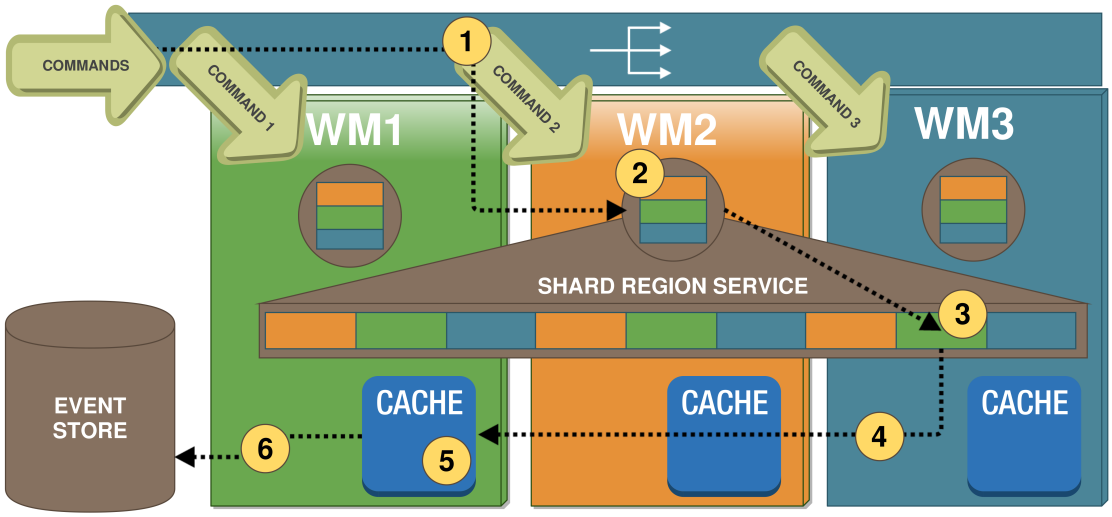


Fig. 2: Scalable command processing is accomplished with the idea of sharding depicted on the diagram. Requests are dispatched by a round-robin load balancer (1) and hit the shard region service (2) on one of the write model nodes (WM1-3). The region service maintains the shard-to-node mapping with its peers on other nodes. It recognizes the shard the command belongs to (3) and dispatches it to a proper node (4). The responsible node looks up its cache (5) and either returns the cached aggregate or constructs a new instance using past events from the event store. Finally, the command is applied, and the generated event is stored (6).

In order to enable **scalable processing** (to scale) out we used the idea of sharding. Instead of replicating the command processing units and dealing with conflict resolution we chose to partition the commands load using consistent hashing of aggregate identifiers. We maintain lots of partitions (a.k.a. shards), at least an order of magnitude more than the number of machines for write model deployment and we assign multiple shards to each machine. That allows us to balance the load when a new node is added by transferring shards from each of the previous nodes. Similarly, when we want to deprovision a machine, we transfer all entities to other machines, partitioning them equally. In fact, we change only the shard assignment as

TABLE I: Objectives of the application architecture along with considered and chosen means of implementation. The chosen solutions are presented in italics.

Objective	***Possible solutions***
Write model	
Consistency guarantees	Strictly consistent model with transactions spanning multiple entities.
	Fine-grained transactional units eventually consistent with each other.
Scalable processing	Processing replication with efficient conflicts resolution.
	Processing distribution (sharding) using consistent hashing.
Decreasing latency	Caching events from event store.
	Persisting state snapshots in the event store.
	Caching recreated entities (event sourced state machines).
Read model	
Effective data representation	In-memory model.
	Graph-oriented database.
Scalable processing	Processing partitioning in a scatter-gather fashion.
	Instance replication and round-robin routing.
Event store	
Consistency guarantees	Strict consistency.
	Eventual consistency.

every aggregate is persisted in a database and can be easily recreated on a new node. The command processing flow is presented on Fig. 2. To **decrease the latency**, we decided to cache aggregates created from replayed events in memory.

In our case the **read model** implements constraints checks that are executed against a configured flight schedule:

- rotation continuity - consecutive legs shares destination and origin airport
- minimal ground time - airplane spends required minimal amount of time grounded
- flight designator uniqueness - each leg has a unique flight identifier within a day

These validations requires quick graph traversal operations. We considered in-memory only model though finally we decided that graph-oriented database is the most **effective data representation**. All events that come to the read model from the event store are transformed to fit this model.

The **scalable processing** was achieved by replicating the instances of read models and balancing the load in front of them. Each node manages its own database instance with a complete model. When there is a need to increase processing power on read side then a new instance of a read model is spawned. We avoid complicated model cloning thanks to the complete history of events that a new instance can ask the event store for and recreate the current state on its own. When the replay is over, the node joins the load balancer group to start handling requests and subscribes to new events to keep the model up-to-date.

We could not achieve horizontal scalability without relaxing **consistency guarantees** on the query side. Bridging two models with an asynchronous **event store** comes with trade-offs. Firstly, read model instances may slightly differ at a given time since updates are not synchronized. Secondly, when a new command is accepted by the write side there exists a time frame in which data contained in read model nodes is slightly outdated. Fortunately these effects were acceptable in our use case.

We decided to create our own event store. In contrary to the most popular solutions, we did not design it from scratch and instead, assembled it from proven building blocks: a column-oriented database and a persistent message queue. Every event is both stored in the database and pushed to the queue where it is retained for a while (durable subscription pattern). That makes the read model more resilient, guarantees at-least once delivery semantic and, most importantly, holds new events coming to an instance being initialized with historical events from the database. This is to make sure that no event is missed during instance startup.

The architecture we proposed adheres to the Reactive Manifesto suggestions as it is:

- message-driven - commands trigger events, events trigger model updates,
- elastic - adding new write and read model instances results in better performance,
- resilient - losing an instance of a write or read model does not prevent the system from working and the lost instance may be easily recreated,
- responsive - the most performant query model may be designed, aggregates in write model are completely independent and easy to cache.

IV. Implementation details

The application was developed using Scala language and its ecosystem. Load balancing is implemented by combining routing and clustering capabilities of Akka toolkit in front of Akka actors. Spray toolkit is responsible for exposing REST endpoints and JSON serialization. We decided to keep business logic separate from Akka actors and other infrastructural concerns.

Apache Cassandra and Apache Kafka were chosen as database and persistent messaging queue respectively for the sake of event store implementation. Akka Persistence along with the already available Cassandra plugin provided event store interface for event sourcing purposes. Since the event store was not a bottleneck during our tests, we simplified the implementation into a single Kafka partition to store events and used single-node deployments of both datastores.

On the write side the sharding concept is covered by Akka Cluster Sharding module that builds on top of Akka Clustering functionality and requires Akka Persistence for state management. The module also enabled us to implement a simple cache layer as it maintains all retrieved aggregates in memory and allows to passivate them on demand. Neo4j database in the embedded mode has been placed on each read model instance. We also went for plain Java API since the performance of other data access layers were not satisfactory.

V. Scalability tests

We decided to evaluate the scalability of read and write part of the application separately. We designed two different workloads, the first contains only queries and the latter executes commands alone.

A. Read model scalability

Firstly, we decided to found the maximum sustainable request rate for a single instance of a model. In order to achieve that, we designed a workflow as follows: we start with 1 request/sec and ramp it up to the maximum number of requests/sec (e.g. 200 requests/sec) during 300 seconds, then we hold the maximum rate for 600 seconds. We treat the first part of the workload as a warmup phase and only the last 300 seconds are taken into consideration to calculate final results. The requests were distributed evenly between all the three types of the schedule validation checks: rotation continuity, minimal ground time and flight designator uniqueness.

We run this workflow with a goal of 125, 150, 175, 200, 225, 250 and 275 requests per second. For each of the values we repeated the test five times. We measured 75th, 95th, 99th and 100th (maximum) percentiles of the response time for every run. We also wanted to make sure that the CPUs, as the most demanded resource by the read model are heavily loaded.

After we learned the sustainable throughput of a single instance, we wanted to run the workload against the setups

with multiple read model instances. We set the frequency of the requests respectively to the capacity of a single node multiplied by the number of instances. We wanted to see if the application can linearly increase its capabilities. If the latency does not change between the runs on the different setups, that will prove the linear scalability.

We reserved 12 machines with HDDs, 4 cores and 4GB of memory each, for running the read model instances. An additional machine of the same configuration was used to deploy the event store components: Kafka, Cassandra and Zookeeper (required by Kafka). The Gatling load testing tool was running on a single 4 core machine with 16GB memory.

In order to recognize the single read model instance capacity, we run the capacity test 35 times in total which is 9 hours of aggregate load and gathered 2.5GB of raw logs. The maximum throughput that we considered sustainable was achieved when the load testing tool was generating 175 queries per second. In this case, each of CPU cores was 75% utilized on average. The results are charted in Figure 3.

Finally, we conducted the scalability tests. We measured the response time percentiles for setups with 1, 2, 4, 8 and 12 machines. Every machine was hosting a single read model instance. The tests were repeated several times in order to verify the reproducibility.

The results are shown in Figure 4. There is almost no difference between the request latencies, only the maximum response time (100th percentile) has greater variance which was in fact expected. It is extremely hard to bound the maximum response time, as even a single occurrence of any unexpected event like packet loss contributes to this metric. But 99% percent of all requests had a nearly identical latency upper bound in each run. That means the read model scales very well, in a linear fashion. We made sure that we selected the appropriate query rates by verifying the CPU utilization on each node. Every of the four cores on each node during the tests was 75% utilized nearly all the time.

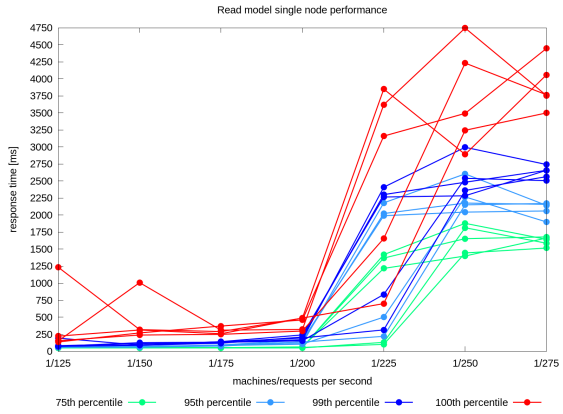


Fig. 3: The figure presents the capacity testing results of a single read model instance. We can see that the query rate of 125-200 request/sec is acceptable but it changes for higher ones.

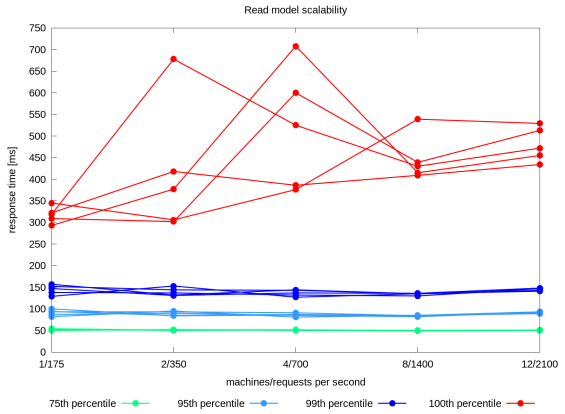


Fig. 4: No difference in the request latency time proves the linear scalability of the read model. The slight increase for 12 nodes may suggest that for the greater number of nodes we could be limited by a single load balancer throughput.

B. *Write model scalability*

In order to evaluate the write model, we took a different approach. We decided to prepare a fixed command workload and run it against the setups with a different number of the write model instances. We wanted to choose a suitable frequency of requests to be able to saturate setups with one or two instances. We were interested how the response time, the CPU utilization and the number of time-outed requests changes when new instances are being added to the setup.

The workload is a simulation of multiple users performing the same scenario. First, they create a rotation, next they add several hundred legs to the rotation, then they create an airplane and finally they assign the rotation to the airplane. We ramped the number of users, adding 50 new users every 5 seconds until we started all of them. If any bad response occurred, the user scenario was interrupted. In order to warm up the instances, we were running the same workload twice in every test case iteration separated by a five minute pause. The first run was considered a warmup phase and only the latter was taken into consideration when the results where calculated.

We reserved 8 machines with 4 cores and 4GB of memory each, for running the write model instances. In this case, we deployed event store on two 16 cores, 16GB machines. Kafka with Zookeeper were placed on the first one and Cassandra on the latter. Finally, we were generating the load on a single 4 core machine with 16GB memory.

We present the results in Figure 5. We conducted the evaluation with two different scenarios: 1250 users adding 750 legs each and 1500 users adding 1000 legs each. We refer to them as lower and higher load scenarios, respectively. We can see that the response time of requests drops when instances are added, similarly the CPU utilization decreases linearly with the number of nodes. In the higher load scenario we can see timeouts indicating that the load is too big for the single instance. They disappear as the number of nodes increases.

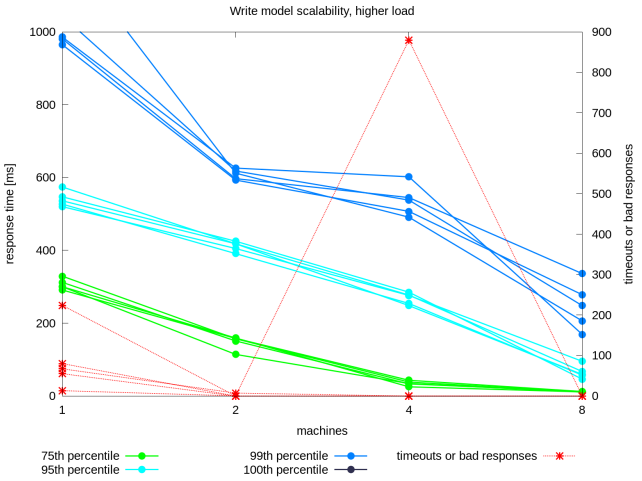


Fig. 5: The plot depicts timeouts occurring in a single-instance deployment. As we add more nodes, the timeouts disappear. A deviation in the number of timeouts for a 4-node deployment visible on plot was caused by an unrecognized, transient problem with the Cassandra database. It did not happen again in any other test so we treat it as a noise.

VI. Related work

There are several tools and undertakings revolving around CQRS and event sourcing ideas, but no comprehensive solution has been developed so far. In contrast to our use case, other solutions often focus on efficient and reliable event store implementation. EventStore [7] is a database crafted for event sourcing with Complex Event Processing capabilities. LinkedIn Databus [8] provides a stream of change capture events for a relational databases, what enables them to be used as event stores and allows adding read models to existing applications.

Axon Framework [9] is a robust framework for building scalable CQRS-based applications with optional event sourced persistence. Akka toolkit [3] currently does not cover all aspects of CQRS yet. However, Akka Persistence, a module for event sourcing applications, is planned to be extended to cover read side of CQRS [10].

The aggregate-oriented approach represented by those tools is not the only way to implement CQRS+ES architecture. Eventuate [11] enables event sourcing implementation with concurrent updates and conflicts resolution. Kappa Architecture [12] is a concept influenced by CQRS and event sourcing for designing stream processing applications that combines a distributed commit log (Apache Kafka) with a stream processing system (e.g. Apache Samza).

VII. Conclusion and future work

We managed to successfully design, implement and prove the horizontal scalability of the application based on CQRS+ES architecture. We derived a simple implementation of the event store and presented other endeavours in the industry revolving around those patterns. We successfully tried out Akka toolkit. It provides several out-of-the-box solutions for non-trivial problems like sharding and constantly is being enhanced with new features. Finally, we make use of DDD approach and find it very useful for building scalable, distributed systems.

Everything comes with a price. Eventual consistency requires developers to challenge their reasoning about control flow in the system. Distribution entails thinking about duplications, losses and retries. Finally, since the approach is still not widely adopted it lacks battle-tested tools, developer guides and best practices. Fortunately, we observe a big interest in CQRS and Event Sourcing patterns currently. New tools, frameworks, event stores and related concepts are popping out day by day and the situation may change quickly.

We consider a handful of ideas for our further research. We are especially interested in stream processing systems in context of CQRS+ES architecture. They may help providing stronger consistency guarantees (e.g. causal consistency) and building a reliable and scalable event store implementation. The latter option may require to create a robust event store benchmark and compare existing solutions.

References

[1] J. Bonér, D. Farley, R. Kuhn and M. Thompson (2014, Sep), *The Reactive Manifesto*, v2.0, [Online]. Available: http://www.reactivemanifesto.org

[2] 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*, 1st ed., Microsoft patterns & practices, 2013.

[3] Typesafe Inc., Akka toolkit, [Online]. Available: http://akka.io.

[4] E. Evans, *Domain-Driven Design: Tacking Complexity in the Heart of Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] C. Hewitt and H. Baker, *Actors and Continuous Functionals*, in Proc. of IFIP Working Conf. on Formal Description of Programming Concepts. August 15, 1977.

[6] P. Helland, *Life Beyond Distributed Transactions an Apostates Opinion*, Conf. on Innovative Database Research (CIDR) January 2007.

[7] G. Young, The Event Store database, [Online]. Available: http://www.geteventstore.com.

[8] S. Das, et al., *All Aboard the Databus!, LinkedIn's Scalable Consistent Change Data Capture Platform*, In Proc. 3rd ACM Symp. on Cloud Computing (SoCC 2012)

[9] A. Buijze, The Axon Framework, [Online]. Available: http://www.axonframework.org.

[10] Typesafe Inc. (2014, Dec), Akka Roadmap Update blog post, [Online]. Available: www.typesafe.com/blog/akka-roadmap-update-dec-2014.

[11] M. Krasser (2015, Jan) The Eventuate toolkit, [Online]. Available: http://rbmhtechnology.github.io/eventuate/.

[12] J. Kreps, Questioning the Lambda Architecture blog post, [Online]. Available: http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html.

Andrzej Debski is a Computer Science M.Sc. student at AGH University of Science and Technology in Krakow, Poland and a Software Engineer in AVSystem. He is mainly interested in distributed computing, functional programming, software engineering and domain-driven design. His prior professional experience includes working in IBM Poland and Sabre Airline Solutions.

Bartlomiej Szczepanik is a Computer Science M.Sc. student at AGH University of Science and Technology in Krakow, Poland and a Software Engineer in Akamai Technologies. His main scientific interest are highly distributed systems, domain-driven design, and productivity engineering. His prior professional experience includes internships in Sabre Airline Solutions and Google Inc. He lives in Krakow, Poland.

Maciej Malawski holds Ph.D. in Computer Science, M.Sc. in Computer Science and in Physics. He is an assistant professor and a researcher at the Department of Computer Science AGH and at ACC Cyfronet AGH, Krakow, Poland. In 2011-13 he was a postdoc and a visiting faculty at Center for Research Computing, University of Notre Dame, USA. He is coauthor of over 50 international publications including journal and conference papers, and book chapters. He participated in EU ICT CrossGrid, ViroLab, CoreGrid, VPH-Share and PaaSage projects. His scientific interests include parallel computing, grid and cloud systems, resource management, and scientific applications.

Stefan Spahr holds a Graduate Degree in Computer Science (Dipl.-Inform. FH). He works as a senior software architect for airline application software and Cloud solutions at Lufthansa GmbH & Co. KG in Berlin, Germany. Before his current job he worked as a software engineer, a development- and implementation-project manager and as a database expert in different departments of the company. He participates in the EU FP7 PaaSage projects as well as in the EU H2020 projects MUSA and BEACON. His main professional interests are Cloud computing architectures and related (emerging) technologies, domain driven design and distributed systems.

Dirk Muthig is the CTO & Innovations of Lufthansa Systems Hungaria Kft. and head of the team Production and Systems Design of Lufthansa Systems GmbH & Co. KG. He is responsible for innovations, standards and guidelines that fully shape the lifecycle of more than 20 major software products for the aviation industry. This product lifecycle includes service transition, which refers to the hand-over of a software system to the operating units. These products must be able to be operated in various settings that are heavily constraint by customer-specific IT infrastructures already existing. Dirk is with Lufthansa Systems for nearly six years. Before he headed the division Software Development at the Fraunhofer Institute for Experimental Software Engineering (IESE) and thus he has intensive experience with all kinds of research projects, as well as with bridging the gap between research and practice. His main research topics have been software product lines, system architectures, and service- or component-based development. He is also the chair of the software product line hall of fame that selects and presents successful industrial case studies on the website of the Software Engineeering Institute (SEI) in Pittsburgh, USA. Dirk has more than 100 publications (listed by the Fraunhofer Publica, see http://publica.fraunhofer.de/starweb/pub09/index.htm).