# In Search for a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing Case Study

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

***Abstract*—As cloud infrastructures are gaining popularity, new concepts and design patterns such as Command-Query Responsibility Segregation and Event Sourcing promise to facilitate development of scalable applications. Despite recent work and the availability of many blogs and tutorials devoted to these topics, there are still few reports on existing implementations in real-world use cases which would provide experimental insight into their scalability. To bridge this gap, we have developed an architecture which exploits both patterns in accordance with Reactive Manifesto guidelines, and implemented a prototype interactive flight scheduling application to experimentally answer the key question concerning scalability. Our performance evaluation in a cloud environment consisting of 15 VMs demonstrates the horizontal scalability of the Command-Query Responsibility Segregation and Event Sourcing patterns, observed independently for read and write models of the application. We present in detail how to assemble this type of architecture from smaller building blocks, first on a conceptual level and later with specific technologies including Akka, Cassandra, Kafka and Neo4J. Our reference implementation is available as an open-source project. We conclude that the evaluated concepts provide many interesting advantages without compromising performance, and thus we can expect to witness their rapid adoption by the industry.**

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

## I. Introduction

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

In this paper we focus specifically on a movement called the *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 is to determine whether it is possible to implement a fully working, scalable CQRS application in a completely reactive fashion. We have decided to build a prototype application using these ideas, focusing on the Akka toolkit [3] as the implementation technology. The prototype provides a subset of the functionality of a real-world application and focuses on scalability, elasticity and responsiveness as a means of demonstrating these new architectural styles.

Our example is based on the interactive flight scheduling application from Lufthansa Systems (LSY). Reactive and scalable application architectures are an important concept for airline-related software. In addition to schedule planning, they can be applied e.g. to flight operation, passenger handling, baggage tracking and so on. Existing client/server or n-tier application stacks cannot fully take advantage of (nearly) unbounded scalability offered by today's cloud environments.

Our application has particular characteristics that make it a good target for CQRS and Event Sourcing. While being interactive it also requires some complex processing in the background. The application maintains a schedule for a whole airline. Under normal conditions, planning covers a single season (which is 6 months); however longer periods (e.g. one year) are also widely used. For reporting or analytics it may be useful to consider even longer periods, such as 5 or 10 years. The corresponding volumes of data must be handled by the application with reasonable response times. Depending on the size of the airline, one season can include hundreds of airplanes and airports, along with several thousand flight events, each of them connecting two airports. The schedule is interactively updated by human planners and allows them to perform on-line validation checks. To simplify matters we implemented only a small subset of the business functionality of a real scheduling system.

A 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 an airplane has to spend on the ground between consecutive legs. To ensure schedule validity, the system must check, at every update, if all the legs in a rotation hold the continuity property, do not violate standard ground times and flight numbers are not duplicated. These *checks* are complex graph queries that need to be performed on a node schedule graph with approximately 200 000 nodes and a minimum memory footprint of 450 MB. This implies high load on the back-end database, which, in turn, greatly impacts performance and scalability.

The main contributions of this paper are as follows:

- we survey recent architectural patterns for scalable systems with focus on reactive applications,

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE

- we demonstrate how we went about development of a flight scheduling application prototype using a novel reactive approach that combines CQRS with DDD principles, together with an open-source implementation of the key components,
- we experimentally evaluate the scalability of the proposed solution in a cloud environment.

Starting with the prototype flight scheduling application (which is proprietary and owned by LSY), we extracted the generic CQRS framework and made it available as an open-source project [12]. We believe that it can be of interest to professionals working on scalability problems. Moreover, we consider the discussion presented in this paper to be of a general nature, since many similar applications can benefit from CQRS and Event Sourcing concepts presented in Section II, the architecture described in Section III, implementation technologies (Section IV), our evaluation methodology detailed in Section V as well as the outline of related work provided in Section VI.

## II. Evaluated concepts

The **CQRS** principle advises separating operations that mutate state (commands) from queries. This creates useful possibilities, e.g. the ability to choose different databases for write and read operations (see Fig. 1a). Developers can select the most performant alternative for queries without sacrificing the benefits of the original (e.g. relational) database for state mutation operations. Furthermore, each of the queries can be optimized separately by maintaining several different read models at once. These benefits come with a cost associated with synchronization of multiple data models and the underlying storage.

CQRS meshes well with the **Event Sourcing** idea, an advanced version of the commit-log pattern, well known to database designers and now considered a mature method of managing persistence. System behavior is modeled in terms of facts (events) happening in the system and state machines, not just the state representation (see Fig. 1b). This enables abandonment of the object-relational mapping to deliver a fully persistence-agnostic model of the system. Moreover, this approach automatically provides a complete audit log of the system. It is especially useful when combined with the CQRS architecture. New read models can be added a long time after deployment of the write model, taking into account the events stored thus far. The user can change the shape of projections (i.e. how a stream of events is converted into a structural representation) and simply recreate them from scratch. Such a commit log may rely on storage which supports append-only operations, usually called an event store.

**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 latter are called tactical patterns and introduce a level of abstraction that helps experts and developers reason about the codebase in terms of business processes and behavior instead of classes and state. For this study the most interesting elements are:

- *aggregate* - defines a transactional unit in a system,
- *domain* event - records the facts (events) that happened in the system.

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

## III. Application architecture

We have divided our application into a write model and a read model accordingly to the CQRS principle. Both parts are connected with an event store. Even though we have committed ourselves to the CQRS+ES architecture, we retain a substantial number of degrees of freedom. Table I, which does not attempt to provide a complete reference to all possible options, presents the main design objectives and possible solutions that we analyzed.

### A. Write model

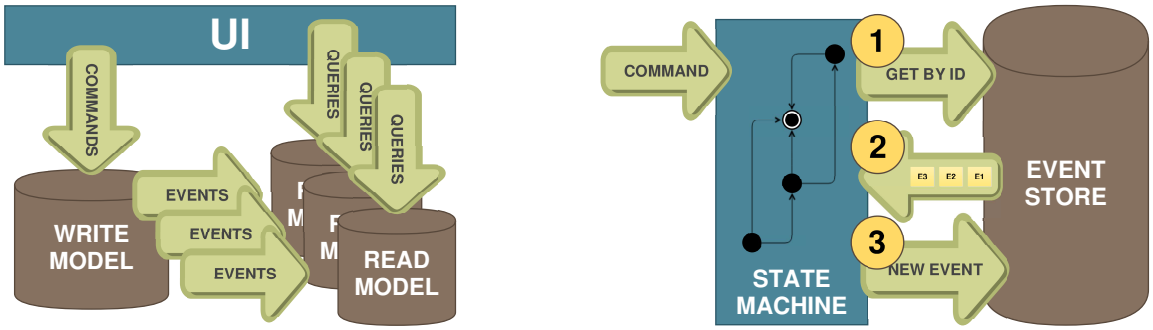
The flight scheduling problem domain does not require strict **consistency guarantees**. This enables us to divide the **write model** into smaller transactions, influenced by Pat Helland's idea of entities [6] (defined as a prerequisite for good scalability) and the DDD aggregate pattern.

An aggregate is a group of application entities that cannot be mutated independently. Conversely, all the operations involving multiple aggregate instances are performed in an independent and eventually consistent manner. That means the smaller the aggregates, the higher the concurrency level and the lower the cost of performing a single transaction, but also the less powerful invariants the system can hold with strong consistency.

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

| *Objective* | *Considered 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 conflict resolution. |
| | *Processing distribution (sharding) using consistent hashing.* |
| Decreasing latency | Caching events from event store. |
| | Persisting state snapshots in 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.* |

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE




(a) (b)

Fig. 1. The 1a diagram depicts the idea of CQRS. The Write Model handles all user commands, validates them and produces events. These events are published to all Read Models which update their query models and become ready to handle new queries. Event Sourcing, depicted in the 1b diagram, introduces an approach which differs slightly from the standard flow of dealing with entities in an enterprise world (i.e. deserialize, mutate state, serialize). In Event Sourcing, the storage component is first asked for a list of all events for a given entity (1). Subsequently, the brand-new state machine applies all retrieved events (2). Finally, the user command is validated and, if successful, a new event is produced and stored (3).

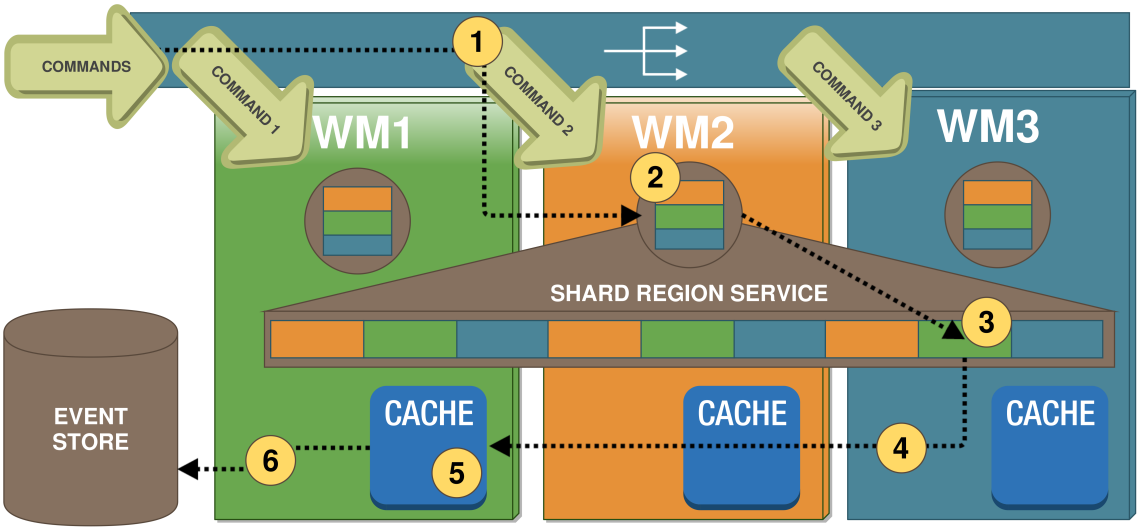



Fig. 2. Scalable command processing is accomplished with the idea of sharding depicted in the diagram. Requests are dispatched by a round-robin load balancer (1) and hit the shard region service (2) on one of the write model nodes (WM1-3). The region service maintains the shard-to-node mapping with its 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).

One of the examples from our system is a rotation aggregate. A rotation is a list of flight legs in a schedule for a single airplane. That requires the legs to be consecutive and not to overlap. We defend these invariants by enclosing all of the rotation legs in a single aggregate.

Aggregates in the system 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, a brand new instance is created and all the 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, which enables concurrent processing of their commands without any interference, locking mechanism or blocking.

In order to enable **scalable processing** (to scale out) in the write model, we use the idea of sharding. Instead of replicating the command processing units and dealing with conflict resolution, we choose to partition the commands load using consistent hashing of aggregate identifiers. We maintain a large number 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. This allows us to balance the load when a new node is added by transferring the shards from each of the previous nodes. Similarly, when we want to deprovision a machine, we transfer all the aggregates to other machines, partitioning

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE

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. The same process is used for fault tolerance, when we discover that a node in the cluster is not available. The command processing flow is presented in Fig. 2. To **decrease the latency**, we cache aggregates created from replayed events in memory.

### B. Read model

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

- rotation continuity – consecutive legs share arrival and departure airports,
- minimal ground time – the airplane spends the required minimal amount of time on the ground,
- flight designator uniqueness – each leg has a unique flight identifier within a day.

These validations require quick graph traversal operations. We initially considered an in-memory-only model, but finally decided that a graph-oriented database is the most **effective data representation** due to the types of queries and checks in our model. All events processed in the read model from the event store are transformed to fit this model.

**Scalable processing** is achieved by replicating the instances of read models and balancing their load. Each node manages its own database instance with a complete model. When there is a need for more processing power on the read side, 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 in order to recreate the current state on its own. When the replay is over, the node joins the load balancer group to begin handling requests, and subscribes to new events to keep the model up to date.

### C. Event store

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 any 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 deemed irrelevant in our use case, and are typically considered acceptable in use cases which can handle eventual consistency [13].

We decided to create our own event store. In contrast 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 stored in the database and pushed to the queue where it is retained for a while (durable subscription pattern). This makes the read model more resilient, guarantees at-least-once delivery semantics and, most importantly, holds new events dispatched to an initializing instance which is still processing historical events from the database. As a result, no event is lost during instance start-up.

### D. Architecture – summary

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; the event store is built on top of scalable datastores,
- 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; the event store uses datastores with enabled replication,
- responsive – the entire system resolves the user queries quickly due to the selection of the most performant query models that could be designed; aggregates in the write model are completely independent and easy to cache, which improves command resolution performance; the event store is highly optimized for append-only operations and fetching events for a single aggregate.

## IV. Implementation details

As the main technology for building the application in the reactive approach we used the Akka toolkit. The toolkit provides middleware which facilitates communication between entities in the system in a message-passing style. It implements the actor model [5] introduced many years ago by Carl Hewitt and popularized by the Erlang community. The toolkit provides many useful capabilities such as cluster membership service, location transparency of actors, different routing strategies, supervision of actors and implementation of the reactive streams specification.

The application was developed using the Scala language and its ecosystem. Load balancing is implemented by combining routing and clustering capabilities of the Akka toolkit in front of Akka actors. The 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 the database and persistent messaging queue respectively in the context of event store implementation. Akka Persistence, along with the corresponding Cassandra plugin, provide an event store interface for event sourcing purposes. Since the event store was not a bottleneck during our tests, we simplified its implementation, using a single Kafka partition to store events and running single-node deployments of both datastores.

On the write side the sharding concept is covered by the 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 their on-demand passivation. A Neo4j database (in the embedded mode) has been deployed in each read model instance. We also decided upon a plain Java API since the performance of other data access layers was unsatisfactory.

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE

## V. Scalability tests

We decided to evaluate the scalability of the read and the write part of the flight scheduling prototype separately. We designed two different workloads: the former contains only queries while the latter only executes commands. Read and write model instances were deployed to an OpenStack cloud consisting of twelve quad-core, 4GB VMs with HDDs matching the requirements of the test scenarios. We equipped this setup with the Nginx load balancer, deployed on an identical, independent VM and using a round-robin algorithm. The event store was deployed on two 16-core, 16GB VMs with HDDs. The first of these hosted Kafka with Zookeeper while the latter provided hardware backup for Cassandra. Note, however, that for the read model tests we simplified the event store setup to a single VM. All VMs were located in the same cloud zone. The load was generated using the Gatling tool running on a single quad-core machine with 16GB memory. One additional machine with a quad-core CPU, 4GB of RAM and 100 GB HDD was used to gather metrics from the application and the Gatling tool.

A complex setup of instrumentation and monitoring tools was prepared to gain insight into the system behavior during the evaluation. The application nodes and the Gatling tool produce metrics that were sent either to StatsD (which, in turn, forwarded them to InfluxDB – this includes application metrics and performance data gathered by Kamon) or dispatched directly to InfluxDB (Gatling). The following metrics were gathered:

- CPU and memory – using the Kamon system-metrics module that provides data readily available through JVM APIs, and the native sigar library for OS-level metrics,
- GC time – both old and new generation. We used G1GC for both Gatling and the application, since experiments revealed that this approach produced the most favorable response time characteristics. Response times reported by Gatling were taken into account, along with analysis of GC logs,
- application-level metrics – number of legs, rotations and airplanes,
- metrics from Gatling concerning request response times and active users,
- metrics from the HTTP server (spray-can), e.g. the number of open connections and the number of pending HTTP requests.

We did our best to sanitize our results, e.g. by using jitter detection tools for JVM, by repeating each test five times and by carefully warming up all JVM-based components: application instances, data stores and the load testing tool.

### A. Read model scalability

We decided to find the sustainable throughput for a single read model instance. In order to do so, we began by measuring the response time and CPU utilization. Then we checked if a deployment consisting of several machines was able to handle the corresponding load increase. We set the frequency of requests in proportion to the capacity of a single node multiplied by the number of instances. We wanted to find out if the application could linearly increase its capabilities. If the response time does not change between runs on the different setups, this would prove linear scalability. We tuned the cache configuration of the read model database in such a way that most of the processing required using disk I/O and not only RAM. This emulates a scenario when data is too large to fit in memory, and processing read-model queries saturates a single node even at a relatively low rate of 200 requests per second. This, in turn, allowed us to carry out all the tests using a single machine for load generation.

The workflow was designed as follows: we start with 1 request/sec and ramp it up to the maximum number of requests/sec (e.g. 200 requests/sec) over a period of 300 seconds. We then maintain this maximum rate for 600 seconds. Only the last 300 seconds are taken into consideration when calculating final results. The requests were distributed evenly among all three types of schedule validation checks.

The maximum throughput which we considered sustainable was achieved when the load testing tool was generating 175 queries per second, as shown in Fig. 3a. In this case, each CPU core was 75% utilized, on average. Having determined the sustainability threshold, we conducted scalability tests for setups with 1, 2, 4, 8 and 12 read models on separate VMs. The results are shown in Fig. 3b.

There is almost no increase in response time – only the maximum (100th percentile) exhibits greater variance, which was not unexpected. 99% of all requests had a nearly identical response time upper bound in each run. That means the read model scales very well, in a linear fashion. We double-checked that we had selected the appropriate query rates by verifying the CPU utilization on each node. Each one of the four cores on each node was 75% occupied nearly all the time during tests.

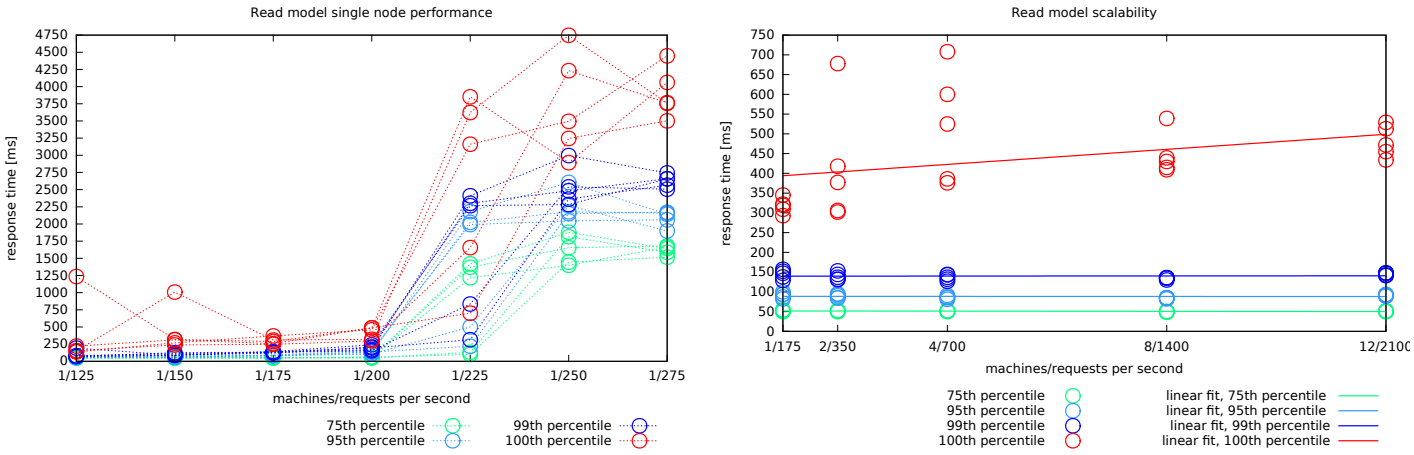
### B. Write model scalability

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

The workload consisted of 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 up the number of users, adding 50 new users every 5 seconds. We ran the same scenario twice for each test case, separated by a five-minute pause, and only the latter scenario was taken into account when collating results.

Results are summarized in Fig. 4. The response time for 99% of requests drops when instances are added, with a threefold decrease between 2 and 8 nodes. We can see timeouts indicating that the load saturated the single instance setup. These disappear as the number of nodes increases. Furthermore,

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE




(a) (b)

Fig. 3. Plot (a) presents the capacity testing results of a single read model instance. Each individual run is shown as a separate data series. We can see that query rates of 125-200 request/sec are acceptable, but higher rates result in rapidly increasing response times. Plot (b) presents the results of the load test for an increasing number of nodes. Each line shows a linear fit computed for each percentile. We observe no increase in the response time for 99% of the requests, which proves linear scalability of the read model.

the CPU utilization decreased linearly with the number of nodes. Two instances were able to handle the load with 87% CPU utilization, but when we switched to 8 nodes, utilization dropped to 50%.

## VI. Related work

There are several tools and projects built 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 an efficient and reliable event store implementation only. For example, 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 relational databases, which enables them to be used as event stores and allows adding read models to existing applications.

The Axon Framework [9] is a robust framework for building scalable CQRS-based applications with optional event sourced persistence. Akka toolkit [3] provides extensive support for event sourcing (e.g. rebuilding actor state from persisted events, pluggable event stores etc.), but full CQRS coverage is still in the experimental phase. Eventuate [11] is similar to Akka, but unlike the aggregate-oriented approach that we used it provides event sourcing with concurrent updates and conflict resolution. Recently, Akka authors have rolled out Lagom [10], a full-fledged microservice framework based on the CQRS+ES architecture. It handles persistence in a similar way to our prototype application, but in contrast to our toolkit approach it enforces the entire application structure, up to the definition of REST endpoints. Despite the high frequency of publishing new tools influenced by CQRS+ES architecture, there have been no in-depth performance studies of this approach we are aware of.

Currently we observe significant industrial interest in various stream processing technologies, from simpler, local solutions like Reactive Streams [14] to full-fledged, distributed processing systems like Twitter Storm [15]. Due to the fact that one of the core parts of the CQRS+ES architecture is event log processing, stream processing tools are frequently chosen for its implementation. In the majority of cases they help construct read models, especially complex ones, e.g. a machine-learned recommendation model based on customers' transactions. We did not use this approach in our application as the read model was simpler and model construction would not benefit from distribution capabilities. There are some attempts to base the entire CQRS+ES architecture (not just its read part) on a stream processing system. CQRS-server [16] defines the write-side command processing as a data-flow workflow. The community gathered around Apache Kafka promotes designing systems to first store a log of updates and then process them into materialized views, which closely resembles CQRS and Event Sourcing. Processing is handled by a commit-log-aware stream processing tool such as Apache Samza [17] or Kafka Streams [18]. This approach is referred to as the Kappa Architecture [19].

## VII. Conclusion and future work

As we can see, it is possible to successfully design and implement the flight scheduling application prototype on the basis of a CQRS+ES architecture. We learned that the actor model and DDD building blocks are very helpful in designing a scalable, distributed architecture. Our experience in assembling the CQRS+ES architecture is embodied by an abstract framework which is available as an open-source project [12]. It builds upon the Akka toolkit and contains a simple implementation of the event store.

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE

We proved that the CQRS+ES architecture is horizontally scalable. We observed no change in the response time of the read model when scaling the load and the number of instances respectively. Additionally, we experienced lower response times and reduced resource consumption when scaling the write model under constant load. We proved that actor model approach is a good match for this type of application architecture.

Everything comes at a price. Eventual consistency requires developers to challenge their reasoning about control flow in the system. Distribution entails thinking about duplications, losses and retries. The speed at which aggregates and views can be built from scratch by replaying events degrades slowly with time and may require mitigation in the form of unwieldy snapshots. Finally, since the approach is still not widely adopted, it lacks field-proven tools, developer guides and best practices. For instance, event versioning may be solved in multiple ways (rewriting, upcasting, version-aware consumers, etc.) and migration handling tools are rather scarce in the relational database world. Fortunately, we continue to observe great interest in CQRS and Event Sourcing patterns. New tools, frameworks, event stores and related concepts are popping up day by day and the situation may change quickly.

From the industrial perspective, the concept of CQRS+ES looks very promising as a means to develop production-grade software. LSY have decided to build new cloud-native applications based on the results of this research and the architecture used in our prototype.

Several ideas have been singled out for further research. We are especially interested in stream processing systems in the context of the CQRS+ES architecture. They may help provide stronger consistency guarantees (e.g. causal consistency) and build a reliable and scalable event store implementation. The latter option may be required when creating a robust event store benchmark and comparing existing solutions. Most notably, the scalability which we proved is a prerequisite for achieving automatic scaling (self-scaling) of the application, is one of the key features of cloud platforms. We intend to take advantage of this fact in the near future.

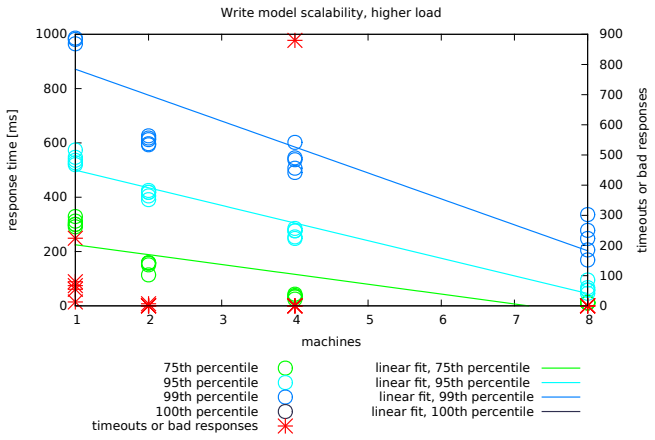



Fig. 4. The plot depicts response times and timeouts occurring in a multiple-instance write model deployment. As we add more nodes, the timeouts disappear. A deviation in the number of timeouts for a 4-node deployment visible on the plot was caused by an unrecognized transient problem with the Cassandra database. As it did not recur in any other test, we treat it as noise.

## ACKNOWLEDGMENT

The presented work is supported by the EU FP7-ICT PaaSage project (317715), Polish grant no. 3033/7PR/2014/2 and AGH grant no. 11.11.230.124.

## 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] Lightbend 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] Lightbend Inc., Lagom, [Online]. Available: www.lightbend.com/lagom.

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

[12] A. Debski, B. Szczepanik, (2016, Jan) Proof-of-concept of CQRS/ES framework, [Online]. Available: https://github.com/cqrs-endeavour/cqrs-endeavour.

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE

[13] W. Vogels, *Eventually consistent*, Commun. ACM, vol. 52, no. 1, p. 40, Jan. 2009.

[14] Reactive Streams, [Online]. Available: http://www.reactive-streams.org/.

[15] Apache Storm, [Online]. Available: https://storm.apache.org/.

[16] CQRS Server, [Online]. Available: https://github.com/Yuppiechef/cqrs-server.

[17] Apache Samza, [Online]. Available: http://samza.apache.org/.

[18] Kafka Streams, [Online]. Available: http://docs.confluent.io/3.0.0/streams/.

[19] 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 the 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 for IBM Poland and Sabre Airline Solutions.

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

**Maciej Malawski** holds a Ph.D. in Computer Science along with an M.Sc. in Computer Science and Physics. He is an assistant professor and 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 the Center for Research Computing, University of Notre Dame, USA. He is the coauthor of over 50 international publications, including journal and conference papers and book chapters. He participated in EU ICT Cross-Grid, 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 a database expert at different departments of the company. He participates in the EU FP7 PaaSage projects as well as in the EU H2020 MUSA and BEACON projects. 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 Production and Systems Design team 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 handover of a software system to the operating units. These products must be able to be operated in various settings that are heavily constrained by customer-specific IT infrastructures already existing. Dirk has been with Lufthansa Systems for nearly six years. Before he headed the Software Development division 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 ).

Digital Object Identifier 10.1109/MS.2017.265095722 0740-7459/$26.00 2017 IEEE