Bachelorarbeit im Studiengang Medieninformatik

How does an Event Sourcing architecture compare to CRUD systems with an independent audit log, when it comes to scalability, performance and traceability?

Vorgelegt von Lukas Karsch

an der Hochschule der Medien Stuttgart am 02.03.2026
zur Erlangung des akademischen Grades eines Bachelor of Science

Erstprüfer: Prof. Dr. Tobias Jordine

Zweitprüfer: Felix Messner

Ehrenwörtliche Erklärung

Hiermit versichere ich, Lukas Karsch, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: "How does an Event Sourcing architecture compare to CRUD systems with an independent audit log, when it comes to scalability, performance and traceability?" selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Lukas Karsch, 02.03.2026

Contents

List of Figures

List of Tables

Listings

Glossary

Anemic Domain Model The objects describing the domain only hold data, no logic. 5, 6

API API stands for *Application Programming Interface*. It describes the public interface of a module or service, often exposed over a network. 17, 24

Atomicity Atomicity means that an action is either fully executed or not at all. Atomic operations make sure the application is not left in an invalid state [1, p. 10]. 7

Contract Test A contract test verifies that services implement a shared interface by testing their interactions against an explicitly defined contract. 15

Docker Open platform for developing, shipping and running containerized applications. 31, 44, 45, 47, 48

Dockerfile A text document containing a series of instructions used to assemble a Docker image. 31

GPath Expression A Groovy-based path language for navigating and querying nested object graphs (such as JSON or XML) using concise, expressive selectors and closures [2]. 30

Groovy A dynamic JVM language that extends Java with concise syntax and powerful features such as closures, making it well suited for scripting, DSLs, and test code [3]. 30

HTTP HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 [4]. 3, 4, 17, 29, 30, 33, 34, 49

Median (P50) The middle value in a sorted list of response times, representing the "typical" delay experienced by users. It indicates that exactly 50% of requests are served faster than this threshold; the other 50% are slower. 18, 20

Percentile A statistical measure indicating the value below which a given percentage of observations in a group of data falls. For example, the n-th percentile is the threshold where n percent of requests are faster than that specific value. 17, 19, 20

REST REST stands for *Representational State Transfer*. It is an architectural style for distributed hypermedia systems. 3

REST Assured A library for testing HTTP servers. 29, 34

Rich Domain Model Objects incorporate both data and the behavior or rules that govern that data. 5, 6

Tail Latency The response times observed at high percentiles (such as P95, P99, or P99.9), representing the slowest requests in a distribution. These "outliers" are critical to monitor because they often affect the most data-intensive operations or represent the worst-case user experience. 18

Testcontainer Testcontainers are a way to declare infrastructure dependencies as code using Docker [5]. 34, 45

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 7, 8

BASE Basically Available, Soft State, Eventual Consistency. 8

CQRS Command Query Responsibility Segregation. 1, 7, 9, 15, 17, 25, 27, 41, 49

CQS Command And Query Separation. 7

CRUD Create Read Update Delete. 1, 6, 7, 10, 15, 17, 24

DAO Data Access Object. 4

DDD Domain Driven Design. 5, 25, 26

DSL Data Specific Language. 30

DTO Data Transfer Object. 8

ES Event Sourcing. 1, 15, 17

HATEOAS Hypermedia as the engine of application state. 4

HTML HyperText Markup Language. 3, 4

JMX Java Management Extensions. 30

JPA Jakarta Persistence API. 24, 35–37

JPQL Java Persistence Query Language. 24

JSON JavaScript Object Notation. 3, 4, 19, 24, 41

ORM Object-relational Mapper. 24

POJO Plain Old Java Object. 25

RPS Requests Per Second. vi, 17, 19, 20, 46, 50

RS Research Question. 1, 13, 15, 17, 21

SLA Service Level Agreement. 18

SLO Service Level Objective. 18, 49

SSH Secure Shell. 48

URI Uniform Resource Identifier. 3

VM Virtual Machine. 18, 44, 45, 48

VU Virtual User. 32, 48

WWW World Wide Web. 3

XML Extensible Markup Language. 4, 10, 24

Chapter 1

Introduction (TODO)

1.1 Motivation (TODO)

1.2 Research questions

This thesis provides a quantitative and qualitative comparison between Event Sourcing and traditional CRUD architectures. The primary research question is: **"How does an Event Sourcing architecture compare to CRUD systems with an independent audit log regarding performance, scalability, flexibility and traceability?"**

To provide a comprehensive answer, the following three sub-questions are addressed:

1. **Performance and Scalability:** How do CRUD and ES-CQRS implementations perform under increasing load, and what are the resulting implications for system scalability and resource efficiency?
2. **Architectural Complexity and Flexibility:** What are the fundamental structural differences between the two approaches, and how do these impact the long-term flexibility and evolution of the codebase?
3. **Historical Traceability:** To what extent can CRUD and ES-CQRS systems accurately and efficiently reconstruct historical states to satisfy business intent and compliance requirements?

The individual findings from these Research Questions (RSs) are combined in the conclusion to provide a holistic answer to the primary research question.

1.3 Goals and non goals (TODO)

This section describes the goals and non-goals of the thesis.

Goals:

- Create two implementations adhering to an identical interface
- Provide quantitative measurements of performance, architectural flexibility and traceability
- Implement both applications "the best they can be": according to principles and best practices for both architectures
- Use out-of-the box configurations to provide the performance comparisons

Non-goals:

- Perfectly implement DDD semantics
- Tune applications to get the best performance
- Implement mechanics to support schema evolution

1.4 Structure of the paper (TODO)

Chapter 2

Basics

To provide a comprehensive framework for the technical implementation discussed in this thesis, this chapter outlines the foundations of modern web architecture, domain-driven design, and the mechanisms facilitating consistency and auditing in event-driven systems.

2.1 WWW, Web APIs, REST

The World Wide Web (WWW) is a connected information network used to exchange data. Resources are can be accessed via Uniform Resource Identifiers (URIs) which are transferred using formats like JSON or HTML via protocols like HTTP. HTTP is a stateless protocol based on a request-response structure. It supports standardized request types, such as `GET` and `POST`, which convey a semantic meaning [6].

Web APIs are interfaces that enable applications to communicate. They use HTTP as a network-based API [7, p. 138]. Modern APIs typically follow REST principles. REST stands for "Representational State Transfer" and describes an architectural style for distributed hypermedia systems [7, p. 76].

REST APIs adhere to principles derived from a set of constraints imposed by the HTTP protocol, for example. One such constraint is "stateless communication": Communication between clients and the server must be *stateless*, meaning the client must provide all the necessary information for the server to fully understand the request.

Furthermore, every resource in REST applications must be addressable via a unique ID, which can then be used to derive a URI to access the resource. Below are some examples for resources and URIs which could be derived from them:

- Book; ID=1; URI=`http://example.com/books/1`
- Book; ID=2; URI=`http://example.com/books/2`

- Author; ID=100; URI=`http://example.com/authors/100`

The "Hypermedia as the engine of application state (HATEOAS)" principle states that resources should be linked to each other. Clients should be able to control the application by following a series of links provided by the server [8].

Every resource must support the same interface, usually HTTP methods (GET, POST, PUT, etc.) where operations on the resource correspond to one method of the interface. For example, a POST operation on a customer might map to the `createCustomer()` operation on a service.

Resources are decoupled from their representations. Clients can request different representations of a resource, depending on their needs [8]: a web browser might request HTML, while another server or application might request XML or JSON.

2.2 Layered Architecture Foundations

Layered Architecture is the most common architecture pattern in enterprise applications. Applications following a layered architecture are divided into *horizontal layers*, with each layer performing a specific role. A standard implementation consists of the following layers:

- Presentation: Handles requests and displays data in a user interface or by turning it into representations (e.g. JSON).
- Business: Encapsulates business logic.
- Persistence: Persists data by interacting with the underlying persistence technologies (e.g. SQL databases).
- Database: Manages the physical storage, retrieval, and integrity of the application's data records.

A key concept in this design is layers of isolation, where layers are "closed", meaning a request must pass through the layer directly below it to reach the next, ensuring that changes in one layer do not affect others.

In a layered application, data flows downwards during request handling and upwards during the response: a request arrives in the presentation layer, which delegates to the business layer. The business layer fetches data from the persistence layer which holds logic to retrieve data, e.g. by encapsulating SQL statements.

The database responds with raw data, which is turned into a Data Access Object (DAO) by the persistence layer. The business layer uses this data to execute rules and make decisions. The result will be returned to the presentation layer which can then wrap the response and return it to the caller. [9]

The data in layered applications is often times modeled in an *anemic* way. In an Anemic Domain Model, business entities are treated as only data. They are objects which contain no business logic, only getters and setters. Business logic is entirely contained in the business (or "service") layer. Fowler [10] describes this as an object-oriented *antipattern*, as this approach effectively separates data from behavior, resulting in a procedural design that undermines the core principle of object-oriented programming: the encapsulation of state and process within a single unit.

TODO: A figure may placed here

2.3 Domain Driven Design

Domain Driven Design (DDD) is a different architectural approach for applications. It differs from layered architecture primarily in the way the domain is modelled and the responsibilities of application services.

The core idea of DDD is that the primary focus of a software project should not be the underlying technologies, but the domain. The domain is the topic with which a software concerns itself. The software design should be based on a model that closely matches the domain and reflects a deep understanding of business requirements. [11, pp. 8, 12]

This domain model is built from a *ubiquitous language* which is a language shared between domain experts and software experts. This ubiquitous language is built directly from the real domain and must be used in all communications regarding the software. [11, pp. 24–26]

The software must always reflect the way that the domain is talked about. Changes to the domain and the ubiquitous language must result in an immediate change to the domain model.

When modeling the domain model, the aim should not be to create a perfect replica of the real world. While it should carefully be chosen, the domain model is artificial and forms a selective abstraction which should be chosen for its utility. [11, pp. 12, 13]

While Layered Architecture organizes code into technical tiers and is typically built on Anemic Domain Models, often resulting in the *big ball of mud* antipattern [9, p. V], DDD demands a Rich Domain Model where objects incorporate both data and the behavior or rules that govern that data. The code is structured semantically into bounded context and modules which are chosen to tell the "story" of a system rather than its technicalities. [11, p. 80]

Entities (also known as reference objects) are domain elements fundamentally defined by a thread of continuity and identity rather than their specific attributes. Entities must be distinguishable from other entities, even if they share the same characteristics. To ensure consistency and identity, a unique identifier is assigned to entities. This identifier is immutable throughout the object's life. [11, pp. 65–69]

Value Objects are elements that describe the nature or state of something and have no conceptual identity of their own. They are interesting only for their characteristics. While two entities with the same characteristics are considered as different from each other, the system does not care about "identity" of a value object, since only its characteristics are relevant. Value objects should be used to encapsulate concepts, such as using an "Address" object instead of distinct "Street" and "City" attributes. Value objects should be immutable. They are never modified, instead they are replaced entirely when a new value is required. [11, pp. 70–72]

Using a Rich Domain Model does not mean that there should be no layers, the opposite is the case. Evans [11] advocates for using layers in domain driven designs. He proposes the following layers: [11, p. 53]

- Presentation: Presents information and handles commands
- Application Layer: Coordinates app activity. Does not hold business logic, but delegate tasks and hold information about their progress
- Domain Layer: Holds information about the domain. Stateful objects (rich domain model) that hold business logic and rules
- Infrastructure layer: Supports other layers. Handles concerns like communication and persistence

Evans [11, p. 75] points out that in some cases, operations in the domain can not be mapped to one object. For example, transferring money does conceptually not belong to one bank account. In those cases, where operations are important domain concepts, domain services can be introduced as part of model-driven design. To keep the domain model rich and not fall back into procedural style programming like with an Anemic Domain Model, it is important to use services only when necessary. Services are not allowed to strip the entities and value objects in the domain of behavior. According to Evans, a good domain service has the following characteristics:

- The operation relates to a domain concept which would be misplaced on an entity or a value object
- The operation performed refers to other objects in the domain
- The operation is stateless

2.4 CRUD architecture

Layered architectures are the standard for data-oriented enterprise applications. These applications mostly follow a CRUD architecture. CRUD is an acronym coined by Martin [12] that stands for "Create, Read, Update, Delete". These four actions can be applied to any record of data.

The state of domain objects in a CRUD architecture is often mapped to normalized tables on a relational database, though other storage mechanisms maybe used. The application acts on the current state of the data, with all actions (reads and writes) acting on the same data.

ACID (Atomicity, Consistency, Isolation, Durability) are an important feature of CRUD applications. They can be guaranteed using transactions, ensuring that data stays consistent and operations are atomic. [1, pp. 10, 11]

Databases in CRUD systems are typically normalized. Normalization is a process of organizing data into separate tables, removing redundancies and creating relationships through "foreign keys". It is the best practice for relational databases. There are several normal forms that can be achieved, each form building on the previous one: to achieve the second normal form, the first normal form has to be achieved first. [12, p. 203]

- 1NF (First Normal Form): Each table cell contains a single (atomic) value, every record is unique
- 2NF (Second Normal Form): Remove partial dependencies by requiring that all *non-key* columns are fully dependent on the primary key
- 3NF (Third Normal Form): Removes transitive dependencies by requiring that non-key columns depend *only* on the primary key
- Further Normal Forms (4NF, 5NF): Require a table can not be broken down into smaller tables without losing data

2.5 CQRS Architecture

Command Query Responsibility Segregation (CQRS) is an architectural pattern based on the fundamental idea that the models used to update information should be separate from the models used to read information. This approach originated as an extension of Bertrand Meyer's Command And Query Separation (CQS) principle, which states that a method should either perform an action (a command) or return data (a query), but never both. [13, p. 148]

CQRS is different from CQS in the fact that in CQRS, objects are split into two objects, one containing commands, one containing queries. [14, p. 17]

CQRS applications are typically structured by splitting the application into two paths:

- Command Side: Deals with data changes and captures user intent. Commands tell the system what needs to be done rather than overwriting previous state. Commands are validated by the system before execution and can be rejected. [14, pp. 11, 12]

- Read Side: Strictly for reading data. The read side is not allowed to modify anything in the primary data store. The read side typically stores Data Transfer Objects (DTOs) in its own data store that can directly be returned to the presentation layer. [14, p. 20]

In a CQRS architecture, the read side typically updates its data asynchronously by consuming notifications or events generated by the write side. Because the models for updating and reading information are strictly separated, a synchronization mechanism is required to ensure the read store eventually reflects the changes made by commands. This usually leads to stale data on the read side.

Each read service independently updates its model by consuming notifications or events published by the write side, allowing the read model to store optimized, denormalized views on the data. [14, p. 23]

TODO: place a figure / digram here

2.6 (Eventual) Consistency

Gray et al. [15] explain that large-scale systems become unstable if they are held consistent at all times according to ACID principles. This is mostly due to the large amount of communication necessary to handle atomic transactions in distributed systems. To address these issues, modern distributed systems often adopt the BASE (Basically Available, Soft State, Eventual Consistency) model which explicitly trades off isolation and strong consistency for availability. Eventually consistent systems are allowed to exist in a so-called "soft state" which eventually converges through the use of synchronization mechanisms over time rather than being strongly consistent at all times. [16, 17] This creates an inconsistency window in which data is not consistent across the system. During this window, stale data may be read. [17]

2.7 Event Sourcing and event-driven architectures

Event driven architecture is a design paradigm where systems communicate via the production and consumption of events. Events are records of changes in the system's domain. [18] This approach allows for a high degree of loose coupling, as the system publishing an event does not need to know about the recipient(s) or how they will react. These architectures offer excellent horizontal scalability and resilience, as individual system components can fail or be updated without bringing down the entire network. [19]

Event Sourcing is an architectural pattern within the landscape of event driven architectures. Event-sourced systems ensure that all changes to a system's state are captured and stored as an ordered sequence of domain events. Unlike traditional persistence models that overwrite data and store only the most recent state, event sourcing maintains an immutable record of every action taken over time. These events are persisted in an append-only event store, which serves as the principal source of truth from which the current system state can be derived. [20, pp. 457, 458]

The current state of any entity in such a system can be rebuilt by replaying the history of events from the log, starting from an initial blank state. [19] To address the performance costs of replaying thousands of events for every request, developers implement projections or materialized views, which are read-only, often denormalized versions of the data optimized for specific queries. [21, 20, pp. 461, 462] This separation of concerns is frequently managed by pairing event sourcing with the Command Query Responsibility Segregation (CQRS) pattern, which physically divides the data structures used for reading from those used for writing state changes. [14, p. 50]

To optimize the reconstruction of state, developers often employ *rolling snapshots*. These snapshots represent a serialized, denormalized point-in-time state of an aggregate, allowing the system to restore the state immediately and only replay the delta of events that occurred after the snapshot was taken. [14, p. 20] This heuristic effectively caps the maximum number of events to be processed, providing a predictable and significant performance gain during the loading process. It is worth noting that the event stream stays intact and is not impacted by a snapshot.

2.8 Traceability and auditing in IT systems

Traceability and auditing are legal requirements across various sectors, as they are derived from federal laws and regulations intended to protect the integrity and confidentiality of sensitive data. Organizations implement these mechanisms to stay compliant with mandates that require a verifiable, time-sequenced history of system activities to support oversight and forensic reviews. In the U.S. financial sector, for example, 17 CFR § 242.613 requires the establishment of a consolidated audit trail to track the complete lifecycle of securities orders, documenting every stage from origination and routing to final execution. [22]

2.8.1 Audit Logs

An audit log (often called audit trail) is a chronological record which provides evidence of a sequence of activities on an entity. [23] In information security, the audit log stores a record of system activities, enabling the reconstruction of events.

[24] A trustworthy audit log in a system can guarantee the principle of traceability which states that actions can be tracked and traced back to the entity who is responsible for them. [25, p. 266]

Fowler [26] describes an audit log as simple and effective way of storing temporal information. Changes are tracked by writing a record indicating *what* changed *when*. A basic implementation of an audit log can have many forms, for example a text file, database tables or XML documents. Fowler also mentions that while the audit log is easy to write, it is harder to read and process. While occasional reads can be done by eye, complex processing and reconstruction of historical state can be resource-intensive.

In distributed environments or complex application architectures, the implementation of an audit log often introduces the "dual-write" problem. This occurs when an application is responsible for updating the primary database (the current state) and simultaneously emitting a record to a separate audit log or messaging system. As Kleppmann [20, pp. 452, 453] notes, ensuring atomicity across these two distinct writes is technically challenging. If the primary database update succeeds, but the audit log write fails, or vice versa, the two systems will diverge, leading to a loss of data integrity where the audit trail no longer reflects the "real" state of the system.

This separation highlights, that in traditional CRUD systems, the audit log is simply a secondary source of truth. As it relies on application- or database-level logic, it is nothing more but a passive observer, relying on notifications from the primary process.

2.8.2 Event Streams as a Basis for Traceability

While traditional audit logs are often implemented as secondary systems that capture state changes, event-driven architectures, such as those utilizing Event Sourcing, turn an event stream into the primary source of truth. In this context, an event stream is not just a diagnostic tool but an exact, chronological sequence of intent-driven records.

As established in section 2.7, every state change is captured as a discrete event. Because these events are immutable and append-only, they provide a natural foundation for the principle of traceability. Unlike traditional "state-based" auditing, where the system might only record that a value changed from A to B, an event stream captures the specific domain context. Tthe *intent* behind a change is semantically conveyed through the event type. For example, while a traditional audit log might simply record a status update to `CLOSED`, an event-sourced system distinguishes between an `AccountDeletedByUser` event and an `AccountTerminatedForInactivity` event. This inherent metadata provides an exhaustive audit trail without the need for additional logging logic.

Fowler [19] notes that because the event log is complete, the system can perform *Temporal Queries*, effectively "time-traveling" to reconstruct the exact state of the system at any historical checkpoint. This makes event streams particularly robust for forensic reviews and regulatory compliance, as they eliminate the "information loss" associated with traditional database overwrites. In the context of the legal requirements discussed in the previous section, the event stream serves as a sequence of actions that satisfies the need for a verifiable, time-sequenced history.

2.8.3 Rebuilding state from an audit log and an event stream

There is a fundamental difference in how systems built with a secondary audit log versus an event-sourced architecture reconstruct historic state. It lies in the relationship between the operational data and the chronological record. In systems with a secondary audit log, the audit log and the application state are often updated as two separate operations. This decoupling introduces the risk of silent divergence. If a failure occurs during the logging process, but the primary state change succeeds, the audit trail becomes an incomplete reflection of reality. Because the system continues to function using the primary database, these discrepancies may remain undetected until a forensic reconstruction is attempted. In this scenario, the audit log serves as a secondary piece of evidence rather than a definitive blueprint, making it difficult to guarantee that a reconstructed state is perfectly synchronized with the original historical state.

In contrast, rebuilding state from an event stream is a deterministic process. Since the event stream is the primary source of truth, there is no secondary state to diverge from. State is reconstructed by mapping events to objects (aggregates or projections). If an event is not recorded, the state change never occurred. This guarantees that the log and the system state are consistent by design, ensuring traceability.

2.9 Scalability of systems (TODO)

This section describes which factors play a role in the scalability of a system. Architectural concerns (e.g. refactoring systems to be split up into microservices), resource consumption, etc.

We define different angles on scalability:

- Throughput Scalability: How the system handles an increasing number of commands (writes) vs. queries (reads).
- Data Volume: How the system behaves as the history of events or audit logs grows into the terabytes.

- Organizational / architectural Scalability: How easily multiple teams can work on the system without creating bottlenecks (the "microservices" angle).

Then address write and read scalability; and differences in ES and CRUD architectures. CRUD: e.g. database contention (locks); CQRS would require sharding based on aggregate ID. Two-phase commits in a distributed system?

ES-CQRS with snapshots as strategy for scaling huge event streams (quicker reconstruction of state); but increased storage size. Data is often duplicated across projections.

Chapter 3

Related Work (TODO)

After the theoretical foundations of the thesis are covered, this chapter presents work related to the Research Questions. As there are 3 sub-questions to address, this chapter is divided into three sections.

3.1 RS1 - Performance and Scalability

- Kleppman, 2017, data intensive systems - Chapter 1: scalability, performance, approaches for coping with load - Chapter 11 (stream processing). Performance implications of log structured storage vs. B-Trees (db index)
- Jogalekar et al, 2000. "Evaluating the scalability of distributed systems" provides a formal mathematical framework for defining "Scalability" ($P = \lambda/T$).
- Singh, 2025 presents a performance comparison between DDD and CQRS. While DDD is not a traditional CRUD architecture, clearly the migration to separate read and write paths gave a performance increase
- Jayaraman et al, 2024 "Implementing Command Query Responsibility Segregation (CQRS) in Large-Scale Systems" includes a performance benchmark (p. 58ff)

3.2 RS2 - architectural complexity, maintainability, flexibility

- Singh, 2025 "Using CQRS and Event Sourcing in the Architecture of Complex Software Solutions" - specific results on code metrics: the transition to

CQRS/ES increased the total number of classes (from 47 to 213) but decreased the overall cyclomatic complexity (from 534 to 522) - highlights that individual modules become simpler despite higher number of classes

- "Object Oriented Coupling based Test Case Prioritization": - statistical approach (linear regression, hypothesis testing) to correlate OO metrics with software quality - Defines and analyzes the CK Metrics Suite (WMC, DIT, NOC, CBO, RFC, LCOM). It establishes empirical correlations, such as Coupling Between Objects (CBO) having the strongest negative impact on quality
- Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development: can act as "dictionary" for all the code metrics

3.3 RS3 - traceability

- Maybe Gantz, 2014: Basics of IT Audit
- Vamshikrishna Monagari: "Demystifying Event-Driven Microservices in Cloud-Native FinTech Applications" - contains many quantitative results, e.g.40% lower latency for ES applications; 95% reduction in compliance audit preparation time; reduced time for root cause analysis. - Paper seems to be problematic as I could not find some of the mentioned results in the references

Maybe include sources talking about GDPR and the *problems* with a non-erasable event log, and some solutions mitigating the problem? → "right to be forgotten". This is not a goal of this thesis but related.

Chapter 4

Proposed Method

This thesis aims to provide a fair, quantitative comparison of CRUD and CQRS / ES architectures regarding all three research questions. To achieve this, the architectures should be applied not only to the same domain, but to the exact same requirements. The implementations can then be tested against the same contract tests to ensure their interfaces and behavior is identical.

This chapter will first present the requirements and the domain for the actual application, then outline the proposed method used to answer each research question. RS 1, concerning itself with performance and scalability evaluation, necessitates describing load testing foundations and implications of performance on scalability of a system.

Scalability is also a function of architectural flexibility, which is addressed by RS 2. This flexibility will be quantified using code quality metrics based on static analysis, which are also presented.

Finally, traceability, the core of RS 3, is defined and the method to evaluate it will be described. (TODO...)

4.1 Project requirements

The applications will implement a course enrollment and grading system which might for example be used in universities. Core features include:

- Professors can create courses and lectures
- Students can enroll and disenroll from lectures
- Professors can enter grades
- Students can view their current and past lectures
- Students can view their credits

TODO here: create a more precise feature / business rule matrix; consolidate with subsection 4.1.2.

4.1.1 Entities

Two types of users exist in the domain: professors and students. Their personal information is not relevant for this thesis, which is why only their first and last name are stored for presentation reasons. The student additionally has a semester.

Professors can create courses. Courses have a name, a description, an amount of credits they yield, a minimum amount of credits required to enroll and can have a set of courses as prerequisites.

Courses are the "blueprints" for lectures. Lectures are the "implementation" of a course for a semester. Each lecture created from a course yields the course's amount of credits and has the requirements specified by the course. Lectures have a lifecycle: they can be in draft state, open for enrollment, in progress, finished or archived. A lecture has a list of time slots and a maximum amount of students that can enroll.

A lecture can have several assessments. Each assessment has a type. The professor can enter grades for a student and an assessment. Grades are integers in the range of 0 to 100. Credits are awarded to a student as soon as they completed all assessments for a lecture with a passing grade (grade higher than 50) and once a lecture's status is set to finished.

4.1.2 Business rules

Relationships and business rules in this system are deliberately chosen complex, involving many relationships between entities and intricate validation rules. This approach was adopted in order to be able to make realistic assumptions about the research question by evaluating a project that closely resembles complex, real-world scenarios.

The following list presents a selection of business rules which were implemented.

- Existence checks: any requests including references to entities will fail if the references entities do not exist.
- Requests leading to conflicts, for example creating a lecture with overlapping time slots, will fail.
- When a student tries enrolling to a lecture which is already full, they will be put on a waitlist.
- When a student disenrolls from a lecture, the next eligible student (higher semesters are preferred) will be enrolled.

- Actions on a lecture can only be performed during the appropriate lifecycle state (enrolling only when the lifecycle is "open for enrollment", grades can only be assigned when the lecture is "finished").

4.1.3 Contract Tests

To ensure both implementations adhere to the business rules, an extensive test suite was set up. While the internals of the implementations are vastly different architecturally and conceptually, they both have the same public API. This makes it possible to run the same test suite on both apps by sending HTTP requests and verifying their responses. The test suite includes integration tests for all API endpoint covering both regular and edge-case (error) scenarios to ensure that the CRUD and ES-CQRS application exhibit identical state transitions and error behaviors. Section 5.1 outlines the implementation of those tests in detail.

4.2 Performance

To answer RS 1, the performance characteristics of both architectural patterns will be evaluated and compared. To achieve this, load tests are executed on selected endpoints. The theoretical foundations of load testing, the environmental constraints, and the data collection methodology are described in this section.

4.2.1 Theoretical Load Testing Foundations

To provide accurate performance metrics, load testing is performed on both implementations based on the methodology defined by Kleppmann [20, pp. 10–17]. Load is characterized using "load parameters," which vary depending on the system's nature. For this study, the primary load parameter is the number of concurrent requests to the web server. [20, p. 11]

The tests measure **Requests Per Second (RPS)** and client-side response times. Kleppmann [20, pp. 15, 16] emphasizes the importance of client-side measurement to account for "queueing delays." As server-side processing is limited by hardware resources (e.g., CPU cores), requests may be stalled before processing begins. Server-side metrics often exclude this wait time, leading to an overly optimistic view of performance. Consequently, the load-generating client must utilize an "open model," sending requests without waiting for previous ones to complete, to simulate realistic concurrent user behavior.

This thesis adopts the approach of keeping system resources constant while measuring performance fluctuations under varying load intensities. [20, p. 13]

To evaluate results, the arithmetic mean is avoided as it obscures the experience of typical users and the impact of outliers. Instead, this thesis uses percentiles. The median (P50) serves as the metric for "typical" response time. However, to capture the experience of users facing significant delays—often caused by data-intensive operations—it is critical to measure "tail latencys" via the P95 percentile. High percentiles identify the performance of outliers which, despite being a numerical minority, often represent the most valuable or complex operations. [20, pp. 14–16]

4.2.2 Service Level Objectives

While Service Level Agreements (SLAs) are agreements with users regarding uptime and performance, Service Level Objectives (SLOs) are the technical targets used by engineers to meet those requirements. [27] This thesis attempts to define realistic SLOs to establish a "breaking point" for each architecture.

Following Nielsen [28, p. 135], a response time of 100ms is the threshold for human perception of "instant" feedback. This serves as the baseline for the following targets:

- **Latency SLO**: All endpoints must maintain a client-side P95 latency of $\leq$100ms to ensure the system feels "instant" for 95% of requests.
- **Freshness SLO**: In the Event Sourcing implementation, the asynchronous nature of projections introduces a lag. All writes must be reflected in the PostgreSQL read-model within $\leq$100ms to ensure eventual consistency remains imperceptible. This SLO applies only to the ES-CQRS implementation.
- **Reliability SLO**: Both implementations must maintain a failure rate of $<$0.1% under stress.

4.2.3 Comparability and Environment

The test environment and scenarios are defined as code to ensure reproducibility. Tests are executed in an isolated environment with fixed hardware allocations as specified in Table 4.1.

Component	**Specification**
CPU	13th Gen Intel(R) Core(TM) i7-13700H. 14 Cores, 20 total threads. Max. 5GHz
RAM	32GB DDR4 (2x16GB), 3200 MT/s
Hard Drive	SanDisk Plus SSD 1TB 2.5" SATA 6GB/s

Table 4.1: Hardware specifications for the performance evaluation machine

The physical host provisions two Virtual Machines (VMs): the "client VM" for load generation and the "server VM" for the application and its dependencies

(PostgreSQL and Axon Server). While hosting both on one physical machine makes network latency negligible, the "queueing delay" remains measurable at the client level, allowing for the identification of request queues building up on the server, indicating bottlenecks.

4.2.4 Test Execution

The tool k6 is used to generate load. The client follows an "open model" via k6's arrival-rate executors to decouple request rate from response times, as discussed in subsection 4.2.1.

Each test follows a specific pattern:

- **Ramp-up**: Linear increase from 0 to the target RPS over 20 seconds.
- **Steady State**: Constant target RPS for 80 seconds.
- **Ramp-down**: Linear decrease to 0 RPS over 20 seconds.

A *test configuration* (the combination of a script and target RPS) is executed at least 25 times. Between runs, the application and its dependencies (PostgreSQL/Axon Server) are restarted to ensure independent results. Client-side metrics are captured in JSON format, while server-side resource consumption (CPU & RAM usage) is collected via Spring Boot Actuator.

4.2.5 Collected Metrics

All metrics collected during load testing are listed in Table 4.2. The column "Metric" shows the name that will be used to refer to this metric from now on. "Location" describes where the metric is being recorded.

As described in subsection 4.2.1, not just the average latency is measured, but percentiles are used to accurately report the number of users experiencing the respective latency.

While internal metrics, like CPU usage, RAM usage, the number of database connections (*hikari_connections*) and the number of Tomcat worker threads are not measures for user-perceived performance, they can give an indicator about bottlenecks inside the applications.

Even though latencies are measured on both the client and the server, the client latency will be used primarily when visualizing results, as it is the latency users perceive when interacting with the applications. The server latency is recorded because it can help identify queueing delays by exposing differences in server and client latencies which exceed a factor of 1.1x.

axon_storage_size is only recorded for the ES-CQRS application. *postgres_size* can be recorded for both applications. The CRUD implementation stores its entities and audit log in PostgreSQL, while the ES-CQRS implementation uses PostgreSQL as a secondary data store where denormalized projections and lookup tables live. These metrics are relevant when assessing a system's long-term performance, scalability and maintainability.

Metric	Description	Location
latency_avg	Average (arithmetic mean) latency	Server, Client
*latency_p*50	50th percentile Latency (median)	Server, Client
*latency_p*95	95th percentile Latency	Server, Client
*latency_p*99	99th percentile Latency	Server, Client
cpu_usage	CPU usage of the Server process	Server
ram_usage_heap	Usage of Heap memory	Server
ram_usage_total	Usage of total memory	Server
hikari_connections	Number of Data Source connections	Server
tomcat_threads	Number of Tomcat worker threads	Server
postgres_size	Size of PostgreSQL database	Server
axon_storage_size	Size of Axon storage, incl. Event and Snapshot store	Axon Server

Table 4.2: All metrics collected during load testing

4.2.6 Visualizing Results

Data is visualized using box plots and line graphs. Box plots are used to show the distribution of latencies, while line graphs illustrate performance changes as RPS increases. Per scientific standards, error bars are used to represent the variability of the measurements across the test runs.

4.2.7 Performance Implications (TODO)

The results collected during load testing show how the applications perform under varying load and different situations. Writes and reads are both tested using endpoints which require a varying degree of invariant checking or entity JOINS.

This section will describe the implications of the collected results on the predicted scalability of systems. To do so, literature is analyzed which describes the correlation between performance, resource usage and scalability. We analyze how quickly a system reaches its bottlenecks and in which cases which system is better scalable.

Scalability also depends on the architecture of a software. For example, an application designed to separate reads and writes allows to separate the application into microservices more easily than a monolith. This view on scalability will be described further when presenting the results of architectural metrics.

4.2.8 Load Testing Scenarios (TODO)

This subsection lists every load testing scenario and explains why its results are relevant to the research question.

- Create Courses Simple
- Create Courses with Prerequisites
- Get lectures a student is enrolled in
- Enrollment to a lecture
- Create lecture and then read (time to consistency)
- read grade history (simple time-travel query)
- more complex time travel query (TODO)

4.3 Flexibility - Architectural Metrics (TODO)

This section describes various architectural metrics which are established in literature and used to assess the flexibility and quality of a software architecture. It serves as a basis to answer RS 2.

4.4 Traceability (TODO)

This section describes how traceability will be compared; it serves as a basis to answer RS 3. The method uses established static analysis methods to compare the two architectures. Describe how the results will be visualized and what they may indicate. Maybe talk about schema evolution.

Metrics:

- Martin Metrics, focusing on coupling and instability
- MOOD Metrics evaluate the quality of object-oriented designs: method and attribute hiding factor, etc
- Complexity metrics, e.g. cyclomatic complexity
- Dependency metrics: afferent, efferent; abstractness

Literature:

- Richards and Ford, "Fundamentals of software architecture: an engineering approach"
- Chawla and Kauer, "Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development"
- Deshpande et al., "Object Oriented Design Metrics for Software Defect Prediction: An Empirical Study"
- Singh, "Object Oriented Coupling based Test Case Prioritization"
- Overeem et al., "An empirical characterization of event sourced systems and their schema evolution — Lessons from industry"

4.4.1 Accuracy of reconstruction

We define qualitative criteria for comparison:

- Source of Truth Integrity: How the system guarantees that the history matches the current state. Especially in distributed systems (dual-write problem)
- ntent Preservation: The ability to distinguish between different business reasons for the same data change (e.g., "Refund" vs. "Fee")
- Schema Resiliency: How the history survives changes to the database structure or business rules.

Attempt to define "business metrics":

- How ES helps with root cause analysis / time to reproduce and debug bugs in production
- Required time and work to prepare for security audits

4.4.2 Efficiency

- Project already employs load testing
- load test time-travel queries on both applications, assuming equal scenarios, equal amount of data etc. then compare performance
- Define use cases for time-travel queries:
  - grade history (this endpoint already exists). is rather simple as only the state of one entity is needed
  - Another, more complex question. Should be a use-case where a broader part of the application's historic state needs to be reconstructed. e.g. which lectures was a student enrolled in at point T

- measure CPU and RAM usage during reconstruction
- assumption: fetching data from audit log tables may be more efficient, because date filters can be used on indexed tables
- ES system has to play ALL events. (Snapshots can not be used when replaying events, they are only used when rehydrating aggregates. -> this part probably belongs in result)

4.5 Technologies

This section describes all technologies used for the implementation and evaluation of the two applications.

4.5.1 SpringBoot

SpringBoot [1] is an open-source, opinionated framework for developing enterprise Java applications. It is based on Spring Framework [2], which is a platform aiming to make Java development "quicker, easier, and safer for everybody" [29]. At Spring Framework's core is the Inversion of Control (IoC) container. The objects managed by this container are referred to as *Beans*. While the term originates from the JavaBeans specification, a standard for creating reusable software components, Spring extends this concept by taking full responsibility for the lifecycle and configuration of these objects [30, Chapter 1.1]. Instead of a developer manually instantiating classes using the `new` operator, the container "injects" required dependencies at runtime. This process is known as Dependency Injection. [31, Chapter 1]. Spring offers support for several programming paradigms: reactive, event-driven, microservices and serverless. [29]

SpringBoot builds on top of the Spring platform by applying a "convention-over-configuration" approach, intended to minimize the need for configuration. In a 2023 survey by JetBrains, SpringBoot was the most popular choice of web framework. [32]

Spring Boot starters are specialized dependency descriptors designed to simplify dependency management by aggregating commonly used libraries into feature-defined packages. Rather than requiring developers to manually identify and maintain a list of individual group IDs, artifact IDs, and compatible version numbers for every necessary library, starters use transitive dependency resolution to pull in all required components under a single entry. To quickly bootstrap a web application, a developer can simply add the `spring-boot-starter-web` dependency to their Maven or Gradle build file. By requesting this specific functionality, Spring

[1] SpringBoot
[2] Spring Framework

Boot automatically includes essential dependencies such as Spring MVC, Jackson for JSON processing, and an embedded Tomcat server, ensuring that all included libraries have been tested together for compatibility. This approach shifts the developer's focus from managing individual JAR files to simply defining the high-level capabilities the application requires, minimizing configuration overhead and reducing risk of version mismatches. [30, Chapter 1.1.2]

4.5.2 JPA

Jakarta Persistence API (JPA) [3], formerly Java Persistence API is a Java specification which provides a mechanism for managing persistence and object-relational mapping (ORM). Object-relational Mappers (ORMs) act as a bridge between the relational world of SQL databases and the object-oriented world of Java.

Instead of writing SQL to create the database schema, entities can be described using special Java classes (defined by annotations or XML configurations) which can be mapped to an SQL schema. JPA allows querying the database for these entities in a type-safe way by providing a range of helpful query methods on JPA repositories, for example `findAll()` or `findById(UUID id)`. This removes the need to write "low-level", database-specific SQL for basic CRUD operations. Complex data retrieval is also possible with JPA using the Java Persistence Query Language (JPQL), which is an object-oriented, database-agnostic query language.

When using JPA with SpringBoot by including the `spring-boot-starter-data-jpa` dependency, *Hibernate* [4] is used as implementation of the JPA standard. [33, Chapter 1]

4.5.3 PostgreSQL

PostgreSQL [5] is an open-source relational database system which has been in active development for over 35 years. Thanks to its reliability, robustness and performance, it has a strong earned reputation. [34] PostgreSQL is designed for a wide range of workloads and can handle many tasks thanks to its extensibility and large suite of extensions, such as the popular PostGIS extension for storing and querying geospatial data. [35]

4.5.4 Jackson

Jackson [6] is a high-performance, feature-rich JSON processing library for Java. It is the default JSON library used within the Spring Boot ecosystem. Its primary purpose is to provide a seamless bridge between Java objects and JSON data through

[3] JPA
[4] Hibernate
[5] PostgreSQL
[6] Jackson

three main processing models: the Streaming API for incremental parsing, the Tree Model for a flexible node-based representation, and the most commonly used Data Binding module. This data binding capability allows developers to automatically convert (*marshal*) Java Plain Old Java Objects (POJOs) into JSON and vice versa (*unmarshal*) with minimal configuration. Beyond its speed and efficiency, Jackson is highly extensible, offering modules to handle complex Java types like Java 8 Date/Time and Optional classes. Jackson also supports various other data formats such as XML, YAML and CSV. [36, 37]

4.5.5 Axon

Axon Framework [7] is an open-source Java framework for building event-driven applications. Following the CQRS and event-sourcing pattern, Commands, Events and Queries are the three core message types any Axon application is centered around. Commands are used to describe an intent to change the application's state. Events communicate a change that happened in the application. Queries are used to request information from the application.

Axon also supports Domain Driven Design by providing tools to manage entities and domain logic. [38, 39]

Axon Server [8] is a platform designed specifically for event-driven systems. It functions as both a high-performance Event Store and a dedicated Message Router for commands, queries, and events. By bundling these responsibilities into a single service, Axon Server replaces the need for separate infrastructures such as a relational database for events and a message broker like Kafka or RabbitMQ for communication. Axon Server is designed to seamlessly integrate with Axon Framework. When using the Axon Server Connector, the application automatically finds and connects to the Axon Server. It is then possible to use the Axon server without further configuration. [40, 41]

Command dispatching

Command dispatching is the starting point for handling a command message in Axon. Axon handles commands by routing them to the appropriate command handler. The command dispatching infrastructure can be interacted with using the low-level `CommandBus` and a more convenient `CommandGateway` which is a wrapper around the `CommandBus`.

`CommandBus` is the infrastructure mechanism responsible for finding and invoking the correct command handler. At most one handler is invoked for each command; if no handler is found, an exception is thrown.

[7] Axon Framework
[8] Axon Server

Using `CommandGateway` simplifies command dispatching by hiding the manual creation of `CommandMessages`. The gateway offers two main methods for synchronous and asynchronous patterns. The `send` method returns a `CompletableFuture`, which is an asynchronous mechanism in Java. If the thread needs to wait for the command result, the `sendAndWait` method can be used.

In general, a handled command returns `null`, if handling was successful. Otherwise, a `CommandExecutionException` is propagated to the caller. While returning values from a command handler is not forbidden, it is used sparsely as it contradicts with CQRS semantics. One exception: command handlers which *create* an aggregate typically return the aggregate identifier. [42, 43]

Query Handling

Before a query is handled, Axon dispatches it through its messaging infrastructure. Just like the command infrastructure, Axon offers a low-level `QueryBus` which requires manual query message creation and a more high-level `QueryGateway`.

In contrast to command handling, multiple query handlers can be invoked for a given query. When dispatching a query, callers can decide whether they want a single result or results from all handlers. When no query handler is found, an exception is thrown.

The `QueryGateway` includes different dispatching methods. For regular "point-to-point" queries, the `query` method can be used. Subscription queries are queries where callers expect an initial result and continuous updates as data changes. These queries work well with reactive programming. For large result sets, streaming queries should be used. The response returned by the query handler is split into chunks and streamed back to the caller. All query methods are asynchronous by nature and return Java's `CompletableFuture`. [44]

Aggregates

An aggregate is a core concept of Domain Driven Design (DDD). In Axon, an aggregate defines a consistency boundary around domain state and encapsulates business logic. Aggregates are the primary place where domain invariants are enforced and where commands that intend to change domain state are handled.

Aggregates define command handlers using methods or constructors annotated with `@CommandHandler`. These handlers receive commands and decide whether they are valid according to domain rules. If a command is accepted, the aggregate emits one or more domain events describing *what* happened. Command handlers are responsible only for decision-making; they must not directly mutate the aggregate's state. Instead, all state changes must occur as a result of applying events.

Every aggregate is typically annotated with `@Aggregate` and must declare exactly one field annotated with `@AggregateIdentifier`. This identifier uniquely identifies the aggregate instance. Axon uses it to route incoming commands to the correct aggregate and to load the corresponding event stream when rebuilding aggregate state.

By default, Axon uses event-sourced aggregates. This means that aggregates are not persisted as a snapshot of their fields. Instead, their current state is reconstructed by replaying all previously stored events. Methods annotated with `@EventSourcingHandler` are called by Axon during this replay process to update the aggregate's internal state based on event data. Since events represent facts that already occurred, event sourcing handlers must not contain business logic or make decisions.

Axon also supports multi-entity aggregates. In this model, an aggregate may contain child entities that participate in command handling. Such entities are registered using `@AggregateMember`, and each entity must define a unique identifier annotated with `@EntityId`. Based on this identifier, Axon is able to route commands to the correct entity instance within the aggregate. [45]

External Command Handlers

Often, command handling functions are placed directly inside the aggregate. However, this is not required and in some cases it may not be desirable or possible to directly route a command to an aggregate. Thus, any object can be used as a command handler by including methods annotated with `@CommandHandler`. One instance of this command handling object will be responsible for handling *all* commands of the command types it declares in its methods.

In these external command handlers, aggregates can be loaded manually from Axon's repositories using the aggregate's ID. Afterward, the `execute` function can be used to execute commands on the loaded aggregate. [46]

Set-based validation

When receiving a command, aggregates handle it by validating their internal state inside command handlers and either rejecting the command or publishing an event. However, validation across a set of aggregates, called "set-based validation", is not possible inside a single aggregate. A business requirement like "Usernames must be unique" can only be implemented using set-based validation, as the entire set of aggregates must be inspected before making a decision.

Set-based implementation in Axon can be implemented by using lookup tables. This approach utilizes a dedicated command-side projection, often referred to as a lookup projector, to maintain a specialized view of the system state. While projectors are typically associated with the read-side of a CQRS architecture, a lookup projector is specifically designed to support the command side. It maintains a highly optimized and consistent dataset, such as a registry of unique identifiers, which can be queried during the validation phase of a command.

To ensure that this lookup table remains synchronized and provides the necessary consistency for validation, Axon employs subscribing event processors, which are described in section 4.5.5. Unlike tracking event processors which operate asynchronously and introduce eventual consistency, subscribing event processors execute within the same thread and transaction as the event publication. This mechanism ensures that the lookup table is updated immediately after an event is applied to the aggregate. Consequently, if the update to the lookup table fails due to a constraint violation or database error, the entire transaction is rolled back, preventing the system from reaching an inconsistent state.

In practice, this validation logic is often encapsulated within a domain service or a validator interface that is injected directly into the aggregate's command handler. This service interacts with the lookup table repository to verify global invariants before the aggregate state is modified. By separating the lookup logic from the read-model, the system avoids the latency of eventual consistency while maintaining the architectural integrity of the aggregate as a boundary for consistency. This pattern effectively bridges the gap between the isolated nature of individual aggregates and the necessity for global state verification in complex domain models. [47]

Events

Event handlers are methods annotated with `@EventHandler` which react to occurrences within the app by handling Axon's event messages. Each event handler specifies the types of events it is interested in. When no handler for a given event type exists in the application, the event is ignored. [48]

Axon's `@EventBus` is the infrastructure mechanism dispatching events to the subscribed event handlers. Event stores offer these functionalities and additionally persist and retrieve published events. [49]

Event processors take care of the technical part aspects of event processing. Axon's `EventBus` implementations support both subscribing and tracking event processors. [49] Subscribing event processors subscribe to a message source, which delivers (pushes) events to the processor. The event is then processed in the same thread that published the event. This makes subscribing event processors suitable for real-time updates of models. However, they can only be used to receive current events and do not support event replay. Additionally, as they run on the same thread, they can not be parallelized. [50]

Tracking event processors, which a type of streaming event processors, read (pull) events to be processed from an event source. They run decoupled from the publishing thread, making them parallelizable. These event processors use tracking tokens track their position in the event stream. Tracking tokens can be reset and events can be replayed and reprocessed. Tracking event processors are the default in Axon and recommended for most ES-CQRS use cases. [51]

Subscribing event processors can be configured using SpringBoot's `application.properties` file or through Java configuration classes.

Sagas

In Axon, Sagas are long-running, stateful event handlers which not just react to events, but instead manage and coordinate business transactions. For each transaction being managed, one instance of a Saga exists. A Saga, which is a class annotated with `@Saga` has a lifecycle that is started by a specific event when a method annotated with `@StartSaga` is executed. The lifecycle may be ended when a method annotated with `@EndSaga` is executed; or conditionally using `SagaLifecycle.end()`. A Saga usually has a clear starting point, but may have many different ways for it to end. Each event handling method in a Saga must additionally have the `@SagaEventHandler` annotation. [52]

The way Sagas manage business transactions is by sending commands upon receiving events. They can be used when workflows across several aggregates should be implemented; or to handle long-running processes that may span over any amount of time. [52] For example, the lifecycle of an order, from being processed, to being shipped and paid, is a process that usually takes multiple days. A use case like this is typically implemented using Sagas.

A Saga is associated with one or more association values, which are key-value pairs used to route events to the correct Saga instance. A `@StartSaga` method together with the `@SagaEventHandler(associationProperty="aggregateId")` automatically associates the Saga with that identifier. Additional associations can be made programmatically, by calling `SagaLifecycle.associateWith()`. Any matching events are then routed to the Saga. [53]

For example, a Saga managing an order's lifecycle may be started by an `@OrderPlaced` event and associated with the `orderId`. It can then issue a `CreateInvoiceCommand` using an `invoiceId` generated inside the event handler. The Saga then associates itself with this ID to be notified of further events regarding this invoice, such as an `InvoicePaidEvent`.

4.5.6 Testing

To ensure functionality of the applications, unit and integration tests were implemented using various testing libraries like JUnit as the testing platform, REST Assured for making and asserting HTTP calls, Mockito for unit testing and ArchUnit for architecture tests. This section describes all mentioned technologies.

JUnit [9] is an open-source testing framework for Java. It offers a structured way of writing tests, driven by lifecycle methods like `beforeEach` or `afterAll`. Tests are annotated with `@Test`. They can also be parametrized and run repeatedly. Results can be asserted using assertion methods like `assertTrue()`. [54]

REST Assured [10] is a Java library that provides a highly fluent DSL for testing and validating REST APIs in a readable, chainable style. It allows complex assertions to be written inline using Groovy expressions, making it easy to deeply verify JSON responses beyond simple field checks. [55]

The below code example shows how one might use a Groovy expression to find and validate a path in the returned JSON object:

```
RestAssured.when()
    // omitted request
    .then()
    .body(
        "data.grades.find { it.combinedGrade == 0 }.credits",
        equalTo(0)
    );
```

Listing 4.1: Validating JSON path using Rest Assured

Here, the path `data.grades` of the returned JSON object is expected to be an array. The array is filtered using a GPath expression with a closure to find the first entry where `combinedGrade` equals 0. Then, this entry's `credits` field is extracted and validated using the `equalTo(0)` matcher.

4.5.7 SpringBoot Actuator

Spring Boot Actuator [11] is a tool designed to help monitor and manage Spring Boot applications running in a production environment. It provides several built-in features that allow developers to check the status of the application, gather performance data, and track HTTP requests. These features can be accessed using either HTTP or JMX (Java Management Extensions), which is a standard Java management technology. By using Actuator, developers can quickly see if an application is running correctly without the need to write custom monitoring code.

The most common way to use Actuator is through its "endpoints", which are specific web addresses that provide different types of information. For example, the health endpoint shows whether the application and its connected services, like databases, are functioning correctly, while the metrics endpoint displays detailed data on memory and CPU usage. Beyond the standard options, developers can

[9] JUnit 5
[10] REST Assured
[11] SpringBoot Actuator

also create their own custom endpoints or connect the data to external monitoring software to visualize how an application is performing over time.

Actuator can be enabled in a Spring Boot project by including the `spring-boot-starter-actuator` dependency. [56]

4.5.8 Prometheus

Prometheus [12] is an open-source systems monitoring toolkit that was originally developed at SoundCloud and is now a project of the Cloud Native Computing Foundation. It is primarily used for collecting and storing multidimensional metrics as time-series data, meaning information is recorded with a timestamp and optional key-value pairs called labels. The system is designed for reliability and is capable of scraping data from instrumented jobs and web servers, storing it in a local time-series database, and triggering alerts based on predefined rules when specific thresholds are met. Through its powerful functional query language, PromQL, developers can aggregate and visualize performance data. [57, 58]

To collect and export Actuator metrics specifically for Prometheus, the `micrometer-registry-prometheus` dependency must be included in the classpath. [59] Access to the metrics is granted by including "prometheus" in the list of exposed web endpoints within the application's configuration properties. Once these components are in place, the metrics are automatically formatted for consumption and can be scraped by a Prometheus server. [60]

4.5.9 Docker

Docker [13] is a platform used for developing and deploying applications. It is designed to separate software from the underlying infrastructure, allowing for faster delivery and consistent environments.

Docker's capabilities are centered around the use of containers, which are lightweight and isolated environments. Each container is packaged with all necessary dependencies required for an application to run, ensuring it operates independently of the host system. These workloads can be executed across different environments, such as local computers, data centers, or cloud providers, ensuring high portability. [61]

A Dockerfile is a text-based document containing a series of instructions for assembling a Docker image. Each command in this file results in the creation of a layer in the image, making the final template efficient and fast to rebuild. These images serve as read-only blueprints from which runnable instances, or containers, are created. [62]

[12] Prometheus
[13] Docker

Docker Compose is a tool used to define and manage applications consisting of multiple containers. A single configuration file is used to specify the services, networks, and volumes required for the entire application stack. The lifecycle of complex applications can be managed with this tool, enabling all associated services to be started, stopped, and coordinated with a single command. [63]

4.5.10 k6

Grafana k6 [14] is an open-source performance testing tool designed to evaluate the reliability and performance of a system. It simulates various traffic patterns, such as constant load, sudden stress spikes, and long-term soak tests, to identify slow response times and system failures during development and continuous integration. Metrics are collected during execution and can be visualized through platforms like Grafana or exported to various data backends for detailed reporting. [64]

k6 allows tests to be written in JavaScript, making it accessible and easy to integrate into existing codebases. Every k6 test follows a common structure. The main component is a function that contains the core logic of the test. This function should be the default export of the JavaScript file. It is executed concurrently for each Virtual User (VU), which act as independent execution threads to repeatedly apply the test logic. The tests can be enhanced using k6's lifecycle functions, such as a setup function, which is executed only once and may be utilized to insert seed data into the system. The test execution can be configured using an "options" object, where VUs, test duration and performance thresholds can be set. [65]

[14] Grafana k6

Chapter 5

Implementation

5.1 Contract Test Implementation

The contract tests are implemented in a separate maven module called `test-suite`[1]. The test classes use the `JUnit 5` testing framework and `REST Assured` to send and assert HTTP requests. A basic test might look like this:

```
@DisplayName("GET /lectures should return 200 and include 2 dates")
@Test
void getLectureDetails_shouldReturn200_returnTwoDates() {
    // First, create seed data
    var lectureSeedData = createLectureSeedData();

    RestAssured.given()
            .when()
            .get("/lectures/{lectureId}",
                lectureSeedData.lectureId())
            .then()
            .statusCode(200)
            .body("data.dates", hasSize(2));
}
```

Listing 5.1: Contract test example; adapted from `test-suite/src/test/karsch.lukas.lectures.AbstractLecturesE2ETest`

All contract tests follow a consistent pattern as shown in Listing 5.1. First, a test method is annotated with `@DisplayName` to provide a descriptive, human-readable name. The test method itself is precisely named after the behavior it asserts. In the example above, the test verifies that the response status code is `200` and that the response body contains a field called `dates` consisting of an array of

[1] `test-suite/src/test/java/karsch.lukas`

size two.

Before making these assertions, each test creates "seed data". Seed data is prerequisite data that must exist on the system under test for the execution to be valid. For instance, a professor, a course, and a lecture must be created before the endpoint to `GET` that specific lecture can be tested. Tests that assert invariants, such as the business rule preventing lecture from having overlapping timeslots, typically set the system time via a Spring Boot Actuator endpoint first.

Once the prerequisites are met, the request is executed and assertions are made using REST Assured. The `given()` block sets up the request requirements like headers, parameters, or body content; the `when()` block defines the action, such as the HTTP method (GET, POST) and the endpoint URL. Finally, the `then()` block is used to verify the response, allowing the developer to assert status codes and validate the data returned in the response body.

The test classes in `test-suite` are all `abstract`, meaning they can not be run directly. Instead, they are intended to be subclassed by the modules implementing the concrete applications (`impl-crud` & `impl-es-cqrs`). The subclasses must implement a set of abstract methods which are implementation specific, for example a method to reset the database in between each test, a method to set the application's time and methods to create seed data for tests.

Necessary infrastructure is spun up by the subclasses using Testcontainers. Testcontainers is a way to declare infrastructure dependencies as code and is an open-source library available for many programming languages. [5]

```
@TestConfiguration
public class PostgresTestcontainerConfiguration {
    @Bean
    @ServiceConnection
    @RestartScope
    PostgreSQLContainer<?> postgreSQLContainer() {
        return new PostgreSQLContainer<>(
                DockerImageName.parse("postgres:latest"));
    }
}
```

Listing 5.2: `impl-crud/src/test/karsch.lukas.PostgresTestcontainerConfiguration`

Listing 5.1 starts a PostgreSQL container using the latest available image. `@ServiceConnection` makes sure the Spring application can connect to the container. This configuration can then be imported as shown in Listing 5.1.

```
@SpringBootTest
@Import(PostgresTestcontainerConfiguration.class)
public class CrudLecturesE2ETest extends AbstractLecturesE2ETest { }
```

Listing 5.3: `impl-crud/src/test/karsch.lukas.e2e.lectures.CrudLecturesE2ETest`

5.2 CRUD implementation

This section presents the relevant aspects of the CRUD implementation[2], mainly focusing on relational modeling using JPA and the audit log implementation.

5.2.1 Relational Modeling

The CRUD implementation uses a normalized database in the Third Normal Form.

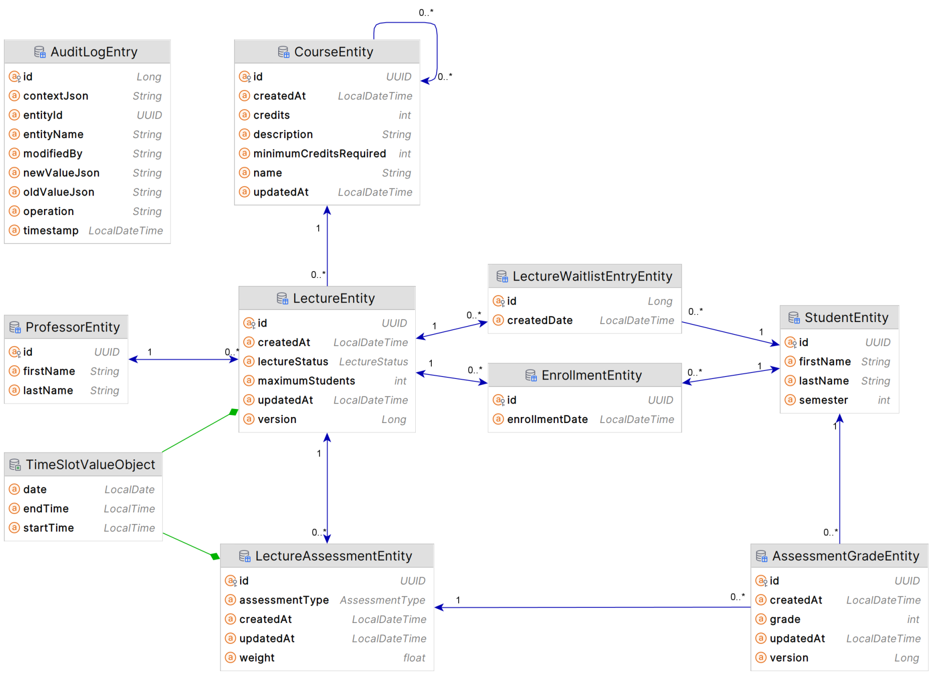

Figure 5.1: Entity Relationship Diagram for the CRUD App

Figure 5.1 shows the Entity Relationship Diagram for the CRUD app. It includes nine entities and a value object for the app's relational database schema. Each box corresponds to an entity or value object, with the bold text being the name. Below the table's name, all attributes of the entity are listed with their type and name.

[2]`impl-crud/src/main/java/karsch.lukas`

Arrows represent an association. The numbers at the end of the arrows convey the multiplicity. An arrow pointing in only one direction stands for a unidirectional association, while an arrow pointing in both directions conveys a bidirectional association. For example, an arrow pointing between entity `A` and entity `B` like so: `1` $\longleftrightarrow$ `0..1` shows that one `A` can be associated with any number of `B`'s, and a `B` is always associated with exactly one `A`.

Arrows with a filled diamond represent a composition. Compositions are used when an entity has a reference to a value object. This value object has no identity and is directly embedded into the entity. The only value object in figure 5.1 is the `TimeSlotValueObject`.

In the app's ER diagram, the `LectureEntity` serves as core of the schema, having several key associations. The 0..* $\longrightarrow$ 1 association to `CourseEntity` shows that many lectures can be created from a course and a lecture is always associated with a course. The 0..* $\longrightarrow$ 1 association to `ProfessorEntity` shows that a professor can hold many lectures (or none), and that a lecture is always associated with a professor. From the lecture's side, these relationships are called "Many to One" relationships.

`LectureEntity` also has "One to Many" relationships to `LectureWaitlistEntryEntity`, `EnrollmentEntity` and `LectureAssessmentEntity`. `LectureWaitlistEntryEntity` is a table which stores students who are waitlisted for a lecture. It is effectively a join table (with one extra column to track when the student was waitlisted) and represents a Many to Many relationship between lectures and students. The same applies to `EnrollmentEntity` which is a table storing which students are enrolled to which lecture. `LectureAssessmentEntity` represents the fact that a lecture can have many assessments (which may be an exam, a paper or a project). Each assessment in turn has many `AssessmentGradeEntitys` associated with it. This table stores which student scored which grade on an assessment.

The `AuditLogEntry` is also visible on the ER diagram, however it has no relationships. This table and the entire audit log implementation will be laid out in the following section.

These entities are implemented using SpringBoot's JPA integration. For example, an entity with a "One to Many" relationship can be implemented like this:

```
@Entity
class LectureEntity {
    @Id
    private UUID id;

    @OneToMany(fetch=FetchType.LAZY)
    private List<EnrollmentEntity> enrollments;
```

```
}
```

Listing 5.4: Simple JPA entity with a "One to Many" relationship

The `@Entity` annotation informs JPA that the class should be mapped to a database table. If the schema generation feature is enabled, JPA automatically creates a table structure that mirrors the class definition. In production environments where this feature is typically disabled, developers must provide SQL scripts to manually define the expected structure. This is commonly achieved either by including a basic initialization script or by utilizing dedicated database migration tools such as Flyway or Liquibase to manage versioned schema changes.

Each entity must include a field annotated with `@Id`, which serves as the unique primary key for the corresponding database record.

The `@OneToMany` annotation defines a relational link between two entities. While the collection is accessed in Java as a standard list via `lecture.getEnrollments()`, JPA manages this behind the scenes using a foreign key relationship. The `fetch` parameter determines when this data is retrieved: `LAZY` loading defers the database query until the collection is explicitly accessed in the code, whereas `EAGER` loading fetches the related entities immediately alongside the parent object.

5.2.2 Audit Log implementation

There are several strategies to implement an audit log, each with its own trade-offs:

1. **Manual Logging**: Developers explicitly call a logging service in every service method that modifies data. While simple, this can lead to code duplication and is prone to human error, such as developers forgetting to add a log statement. A code example might look like this:

```
public void updatePhoneNumber(User user, int newNumber) {
    logChange(Date.now(), user, user.getPhoneNumber(),
        newNumber, "UserRequestedNumberChange");
    user.setPhoneNumber(newNumber);
}

void logChange(
    Date date, User user, Object oldValue, Object newValue,
        String context
) {
    LogEntry logEntry = new LogEntry(date, user, oldValue,
        newValue);
    logRepository.persist(logEntry);
}
```

Listing 5.5: Code example for manual audit logging

2. **Database Triggers or Stored Procedures** can capture changes automatically and directly on the database. This guarantees that no change is missed, even if made outside the application. Ingram [66, p. 515] mentions that database triggers run on a "per-record" basis, meaning the logic is run for each changed record individually. This may lead to degraded performance during batch operations, which is why stored procedures should be preferred over triggers for auditing concerns. It is also worth noting that this approach ties the auditing logic to a specific database, making it less portable.
3. **Hibernate Envers** is an auditing solution for JPA-based applications which automatically versions entities by using the concept of revisions. Envers creates an auditing table for each entity which stores historical data, whenever a transaction is committed. [67]
4. **JPA Entity Listeners**: JPA's lifecycle events (`@PrePersist`, `@PreUpdate`, etc.) can be used to intercept changes. This approach is database-independent and keeps the logic within the Java application, allowing access to application internals like beans and Spring's security context. In full-grade applications built using Spring Security, the security context lets developers access the current user, making it possible to attach them to the new audit log entry. (needs reference)
5. TODO: talk about CDC (change data capture) here or in subsection 2.8.1

5.2.3 Chosen Implementation: Hibernate Envers (TODO)

This section describes Hibernate Envers in detail; how it was implemented into the CRUD app, how it captures context, etc.

Reconstructing historic state (TODO)

Show how Envers can be queried for historic state (see `impl-crud/src/main/java/karsch.lukas.stats.StatsService`).

5.3 ES/CQRS implementation

5.3.1 Architecture Overview

The architecture of the `impl-es-cqrs` application [3] differs from the traditional layered architecture seen in the `impl-crud` application. While the CRUD implementation also has some vertical slicing, the ES-CQRS implementation is much more explicit about it. The code is organized into "features", each representing a vertical

[3] `impl-es-cqrs/src/main/java/karsch.lukas`

slice of the application's functionality (e.g., `course`, `enrollment`, `lectures`). Each feature is self-contained and includes its own command handlers, event sourcing handlers, query handlers, and its own web controller, if needed.

A "feature slice" architecture is descriptive and able to communicate the features of a project at a glance. As clean architecture is not in the scope of this thesis, the separation into features with clear naming conventions for command and query components is sufficient, however introducing completely separate modules for the command and read sides would have increased the project structure's readability even more by clearly showing how command and read side have no access to each other.

5.3.2 The API Layer

The `api` package in each feature slice is shared between web controllers, command side and read side, containing the public interface of the application. It defines the Commands, Events, and Queries that are dispatched and handled by the `impl-es-cqrs` application. Keeping the public API in a separate package ensures that the internal implementation details of the `impl-es-cqrs` application are not exposed to its clients.

5.3.3 Command Side

The command side is responsible for handling state changes in the application. It is implemented using Axon's Aggregates, Command Handlers, and Sagas. This section goes in detail about the implementation aspects, using the courses feature as an example.

Aggregates and Set-Based Validation

Aggregates are the core components of the command side. They represent a consistency boundary for state changes. In this implementation, an example of an aggregate is the `CourseAggregate` [4]. It handles the `CreateCourseCommand`, validates it, and if successful, emits a `CourseCreatedEvent`.

Before creating a course, the system must verify that all the specified prerequisite courses actually exist. This is handled by the `ICourseValidator`,[5] which is injected into the aggregate's command handler. The validator employs set-based validation as described in section 4.5.5. Once the prerequisite courses are validated, a `CourseCreatedEvent` is emitted. Otherwise, a specific `MissingCoursesException` is thrown, indicating that command handling was rejected.

[4] `impl-es-cqrs/src/main/java/karsch.lukas.features.course.commands.CourseAggregate`

[5] `impl-es-cqrs/src/main/java/karsch.lukas.features.course.commands.ICourseValidator`

External Command Handlers

Not all commands can be handled by a single aggregate. For instance, assigning a grade to a student for a specific lecture involves the `EnrollmentAggregate` and the `LectureAggregate`. In such cases, a dedicated command handler, `EnrollmentCommandHandler`, is used. This handler coordinates the interaction between the aggregates. It loads the `EnrollmentAggregate` from the event sourcing repository, validates the command (e.g., checking if the professor is allowed to assign a grade for the lecture), and then executes the command on the aggregate.

Sagas for Process Management

Sagas are used to manage long-running business processes that span multiple aggregates. The `AwardCreditsSaga` is a prime example. It is initiated when an `EnrollmentCreatedEvent` occurs. The saga then waits for a `LectureLifecycleAdvancedEvent` with the status `FINISHED`. Once this event is received, the saga sends an `AwardCreditsCommand` to the `EnrollmentAggregate`. The saga ends when it receives a `CreditsAwardedEvent`. This ensures that credits are only awarded after a lecture is finished and all assessments have been graded. It is interesting to note that while the CRUD application calculates awarded credits based on the current state of a lecture, in the ES-CQRS implementation, the fact that credits are awarded after finishing a lecture is explicit. Even when changing the Saga later on, credits which have already been awarded will not be revoked, unless additional, explicit logic is implemented (e.g. by applying a `CreditsRevokedEvent`).

5.3.4 Read Side

The read side listens to events asynchronously and builds read models, called "projections", which are views of the system. A component that listens for events and maintains projections is called a "projector". Projections are designed to answer specific questions about the system: each projector saves exactly the necessary information. This is achieved by using denormalized data models, a contrast to typical CRUD systems that follow normalization rules.

When the system is queried, the queries are routed to the read side. The read side can efficiently fetch data from the projections, usually without `JOIN`s. This makes reads fast. It is important to keep in mind that projections are built asynchronously, meaning they are eventually consistent and may not always reflect the latest changes applied by the command side.

In the context of the ES-CQRS implementation, a good example of a projector that stores denormalized data for efficient querying is the `LectureProjector`. It demonstrates the fact that each projector maintains its own view of the system. Projectors must not query the system using Axon's `QueryGateway` to get access to any data needed for the projection. One reason for that is the fact that when *rebuilding* projections, a common use case in event sourcing, the projectors should be able to run in parallel. If projectors depend on each other, this can result in one projection attempting to query data from another projection that is not yet up to date. This is why the `LectureProjector` not only maintains a view of lectures, but also of courses, professors and students, which are then used when building the lecture's projection.

The projector also illustrates how the projection's database entities are designed: they are built in the same way as the DTO which is returned from the query handler. Arrays and associated objects are not stored via foreign keys but are instead serialized to JSON. This allows the retrieval of all the necessary data to respond to a query with a simple `SELECT` statement. The same concepts apply to all other projectors in the ES-CQRS implementation.

5.3.5 Synchronous Responses with Subscription Queries

A common challenge in CQRS and event-driven architectures is providing synchronous feedback to users. For example, when a student enrolls in a lecture, they expect an immediate response indicating whether they were successfully enrolled or placed on a waitlist. However, commands are usually handled asynchronously. In CQRS, commands are also not intended to return data.

To solve this, the `LecturesController` uses Axon's subscription queries. When an enrollment request is received, it sends the `EnrollStudentCommand` and simultaneously opens a subscription query (`EnrollmentStatusQuery`). This query waits for an `EnrollmentStatusUpdate` event. The read-side projector responsible for processing enrollments publishes this update after processing the respective `StudentEnrolledEvent` or `StudentWaitlistedEvent`. The controller blocks for a short period, waiting for this update to be published, and then returns the result to the user. This approach makes the user interface synchronous, while not contradicting with the asynchronous nature of CQRS systems, as the command handling process is unchanged. While this approach provides the desired synchronous user experience, it has the downside of coupling the client to the event processing flow. In a typical scenario, developers might employ WebSockets or other client-side notification mechanisms to inform the user about the result of their action. However, for the context of this thesis, where the primary goal is to implement two applications with an identical interface, this solution is a pragmatic compromise.

5.3.6 Encapsulation and API Boundaries

To enforce the separation of concerns and maintain a clean architecture, the internal components of the command and read sides are package-private. For example, the `CourseAggregate` and `CourseProjector` are not accessible from outside their respective feature packages. The public API of the application is exposed through the controllers, which only interact with the `CommandGateway` and `QueryGateway`. This ensures that all interactions with the system go through the proper channels and that internal implementations can be changed without affecting the clients.

5.3.7 Tracing Request Flow

This section illustrates the flow of commands and queries through the system. Axon's `CommandGateway` and `QueryGateway` are used in controllers to decouple them from the internals of the application. The gateways create location transparency: a controller does not need to know where its commands and queries are being routed to.

Command Request: CreateCourseCommand

Figure 5.2 illustrates the flow of a command through the system using the example of the `POST /courses` endpoint. Upon receiving a request, the controller constructs a `CreateCourseCommand` containing the request data and dispatches it through the `CommandGateway`. This gateway is responsible for routing the command to the appropriate destination, which in this case is the constructor of the `CourseAggregate`. This constructor is annotated with `@CommandHandler`. The command handler verifies that the command is allowed to be executed by performing validation logic. When creating courses, it has to be made sure that all prerequisite courses actually exist. This check is done using set-based validation. If the validation is successful, the aggregate triggers a state change by applying a `CourseCreatedEvent` via the `AggregateLifecycle.apply()` method. This action notifies the system of the change and persists the event by recording it in the event store.

After being applied, Axon routes the event to all subscribed handlers. The `CourseAggregate`'s `@EventSourcingHandler` is executed, changing the aggregate's internal state. What is worth noting here is that in the case of `CourseAggregate`, only the `id` of the course is set as other properties of the event, like name or description of the newly created course, are not relevant to the command side. Any read-side projectors with `@EventHandlers` for the `CourseCreatedEvent` are also executed after the event is applied.

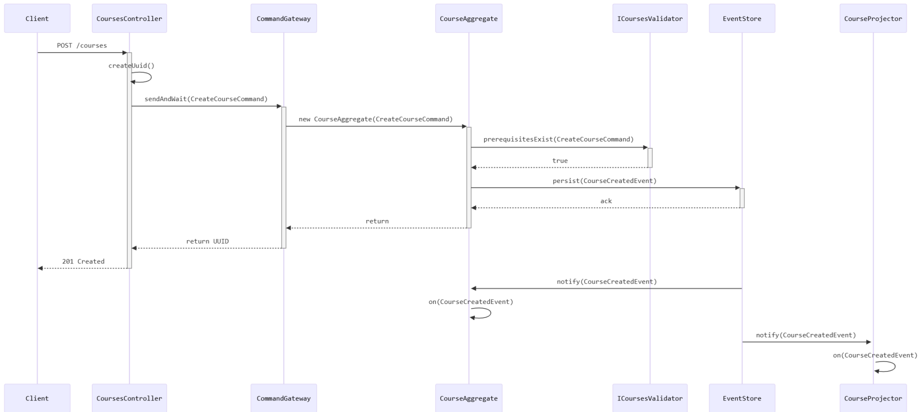

Figure 5.2: Sequence Diagram: Command Flow inside the ES-CQRS application

Query Request: FindAllCoursesQuery

Figure 5.3 illustrates the flow of a query through the application using the `GET /courses` request as an example. The request is received by `CoursesController`. It creates a `FindAllCoursesQuery` instance and sends it to Axon's `QueryGateway`, which routes the query to the appropriate `@QueryHandler` method responsible for `FindAllCoursesQuery`. The query handler method then accesses its JPA repository to get all courses, maps them to a list of `CourseDTOs` and returns this list. The `QueryGateway` hands this result over to the web controller which reads the data and sends it back to the client.

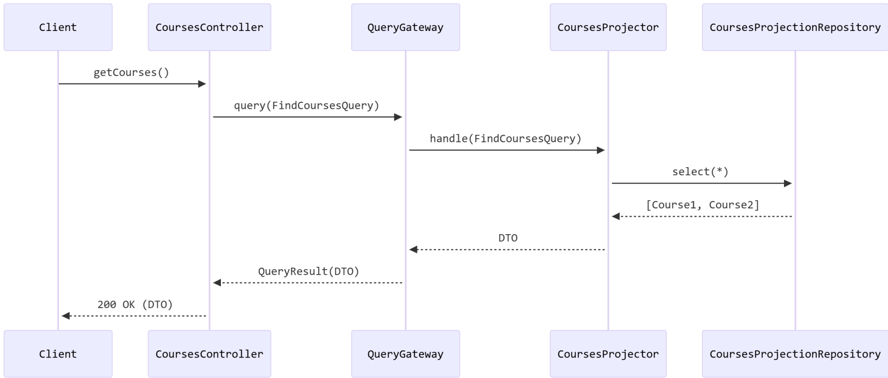

Figure 5.3: Sequence Diagram: Query Flow inside the ES-CQRS application

5.4 Infrastructure

The project's infrastructure is designed for consistency and reproducibility across development and testing environments. It is composed of a containerized environment for running the applications and their dependencies, an automated VM provisioning setup for performance testing, as well as an integration testing strategy using Testcontainers, described in section 5.1.

5.4.1 Containerized Services

The core of the infrastructure is defined in a Docker compose file at the root of the project, which orchestrates the deployment of the two primary applications and their external dependencies: a PostgreSQL database, used by both applications, and an Axon Server instance, used by the ES-CQRS application.

A `postgres:18-alpine` container provides the relational database used by both applications. The database schema, user, and credentials are configured through environment variables. A volume is used to persist data across container restarts.

An `axoniq/axonserver` container provides the necessary infrastructure for the Event Sourcing and CQRS implementation, handling event storage and message routing. It is configured to run in development mode.

The CRUD and ES-CQRS applications are containerized using `Dockerfile`s. Both use `amazoncorretto:25` as the base image, and the compiled Java application (`.jar` file) is copied into the container and executed.

Configuration details, such as database connection strings and server hostnames, are externalized from the `application.properties` files. They are injected into the application containers at runtime as environment variables via the `docker-compose.yml` file, allowing for flexible configuration without modifying the application code.

5.4.2 Local Development and Integration Testing

For local development and integration testing, the project uses the Testcontainers library. This approach allows developers to programmatically define and manage the lifecycle of throwaway Docker containers for dependencies like PostgreSQL and Axon Server directly from the test code. (TODO duplicate?)

By integrating with Spring Boot's Testcontainers support, running the application or its tests automatically starts the required containers. This eliminates the need to manually install and manage these services on their local machines, ensuring a consistent and isolated testing environment. The configuration for this is found in the test resources, where a special JDBC URL prefix signals Spring Boot to manage the database container.

5.4.3 VM Provisioning for Performance Testing

To ensure a stable and isolated environment for performance benchmarks, a dedicated VM setup is used. The process of creating and provisioning these VMs on a Proxmox host is fully automated.

A shell script, `create-vm.sh`,[6] orchestrates the creation of a VM template from an Ubuntu 24.04 cloud image. Cloud images are pre-configured, lightweight variants of operating systems. This script works in conjunction with a CloudInit [7] configuration file that handles the provisioning of the VM upon its first boot. [8]

During the provisioning process, a number of steps are executed. First, it is made sure that the system is up-to-date by installing any available software updates. Next, a 'thesis' user is created for which the environment is configured. Afterward, the script installs all necessary software, including Docker, git, Conda, Python, k6, Maven, and Java 25. Once all necessary software is installed, the project's git repository is cloned and a Maven build is triggered. Finally, the Docker images are built. After these steps are completed, the provisioned VM is ready to run the applications and load tests.

Instead of starting the VM directly, the script shuts the VM down and converts it into a Proxmox template, which can be re-created efficiently. This template is used to create the client and server VMs.

5.5 Load Tests

This section describes the implementation of load tests.

[6] `performance-tests/vm/scripts/create-vm.sh`

[7] `https://cloudinit.readthedocs.io/en/latest/`

[8] `performance-tests/vm/scripts/cloud-init.yml`

5.5.1 k6 Scripts

The core of the load testing suite are the load-generating scripts developed using k6. Listing 5.5.1 illustrates the implementation of a typical k6 script using the creation of courses with prerequisites as an example. [9]

After defining necessary imports, the test script extracts execution parameters from the `__ENV` object which is injected by the k6 test runner. Most k6 scripts written for this project rely on RPS, representing the target iteration rate, and `TARGET_HOST`, which is the URL the application under test is reachable at.

The value of RPS is used to define test options. Namely, a scenario, optional thresholds and the statistics to collect are defined. A test may have several scenarios, however in the k6 scripts used in this project, only one scenario per test is defined. Each scenario has a specific executor. In this case, the "ramping-arrival-rate" executor is used, as opposed to the "ramping-vus" executor. While the "ramping-vus" executor defines the number of virtual users interacting with the application (closed model), "ramping-arrival-rate" executors define the number of iterations per second (open model). This important distinction is described in more detail in subsection 4.2.1. Stages in a scenario define the "timeline" of RPS. In the given example, RPS are increased from 0 to the target RPS over a duration of 20 seconds. This RPS is then held for a duration of 80 seconds, before decreasing RPS back to 0 over a span of 20 seconds.

After defining test options, an optional setup function is implemented. It is executed once by k6, before running the load-generating "export default" function. In the setup function, seed data can be created. The given code example uses the setup function to create 10 prerequisite courses. Their IDs are returned from the setup function.

Data returned from the setup function can be passed to the "export default" function, which is the core of any load test. This is the function that is executed repeatedly to generate load. The implementation of this function in the given example is rather simple. One POST request is sent to the server. This request includes a payload which references a random number of prerequisite courses, as well as other required parameters for course creation.

```
// Imports omitted
const {TARGET_HOST, RPS} = __ENV;

export const options = {
    scenarios: {
        createCourses: {
            executor: "ramping-arrival-rate",
            timeUnit: "1s",
```

[9] `performance-tests/k6/writes/create-course-prerequisites/create-course-prerequisites.js`

```
            preAllocatedVUs: RPS,
            stages: [
                {target: RPS, duration: "20s"},
                {target: RPS, duration: "80s"},
                {target: 0, duration: "20s"}
            ]
        }
    },
    thresholds: {
        'http_req_failed': ['rate<0.01'], // Error rate must be <1%
    },
    summaryTrendStats: ["med", "p(99)", "p(95)", "avg"],
};

export function setup() {
    const prerequisiteIds = createPrerequisites(10);
    return { prerequisiteIds };
}

export default function (data) {
    const {prerequisiteIds} = data;

    const url = `${TARGET_HOST}/courses`;
    const prerequisiteCourseIds = selectRandomPrerequisiteIds();
    const payload = createPayload(prerequisiteCourseIds);
    const res = http.post(url, payload);
    checkResponseIs201(res);
}
```

Listing 5.6: Simplified code example of a k6 script to test course creation. Adapted from `performance-tests/k6/writes/create-course-prerequisites/create-course-prerequisites.js`

5.5.2 Load Test Lifecycle

The k6 scripts alone are not enough to execute a large, repeated load test. While they can generate load on a running application and are capable of collecting client-side metrics, external lifecycle management is needed to control the infrastructure and ensure a clean environment in between each test run.

The lifecycle of repeated load tests is managed using python scripts. The core scripts are `perf_runner.py` [10] and `many_runs.py` [11]. These scripts instrument the entire lifecycle of the application and k6 runs. They are responsible for starting the application using Docker, collecting server-side metrics using Prometheus and

[10] `performance-tests/perf_runner.py`
[11] `performance-tests/many_runs.py`

post-processing results.

The core logic within `perf_runner.py` follows a defined flow for every single test run. It begins by determining the execution context. If a remote configuration is provided, it establishes a Docker Remote Context via Secure Shell (SSH) to interact with the target VM. It then deploys the application using `docker compose up`. Before directing any traffic towards the application, the Actuator's health endpoint is polled to ensure the application is running properly.

Once the application is healthy, the script sets up Prometheus for server-side monitoring. After dynamically generating a `prometheus.yml` configuration file, a Prometheus container is started, targeted to scrape the application under test. To ensure short-term spikes in latency or resource consumption can be captured, the configuration defines a polling interval of 2 seconds.

With the environment and monitoring active, the script invokes k6. Configuration parameters for the test run are expected to be defined in `metric.json`, which is a file placed alongside a test script. It includes metadata and parameters such as the number of VUs and the target host URL. These parameters are passed directly to the k6 engine via environment variables. Inside the k6 scripts, the VUs environment variable is used to define the arrival rate within the ramping-arrival-rate executor rather than a fixed number of concurrent users. Because k6 is configured to trigger a specific number of iterations per second, this parameter effectively acts as a control for Requests Per Second (RPS), ensuring the load remains consistent regardless of how long the individual HTTP calls take to complete.

After k6 completed its load generation, the script enters a data-extraction phase. It queries the Prometheus API to retrieve system-level metrics. Next, it parses the k6-summary.json file, which is a file generated by k6 that includes all metrics recorded during the run. The collected data is processed and merged into standardized CSV files (client_metrics.csv and server_metrics.csv).

Once all data is extracted, the system is ready for the next run. To prepare the environment, all containers need to be stopped first. That is done by running `docker compose down -v` inside the Docker remote context, with the `-v` argument explicitly removing all docker volumes. This makes sure PostgreSQL's and Axon Server's data stores are emptied out before the next test iteration.

While `perf_runner.py` manages the lifecycle of a single test, `many_runs.py` acts as a high-level orchestrator, designed to automate large-scale comparative benchmarks by executing multiple iterations across both implementations by running a single command. The script can be configured to run an arbitrary number of tests, which will be executed for both applications. The script accepts the metric configuration files and passes them on to `perf_runner.py`.

5.5.3 Post Processing Test Results

After extracting data from the k6 output and Prometheus, it is consolidated into a unified CSV format. This is necessary because the two systems use differing naming conventions and units: while k6 might report the 95th percentile latency as $p(95)$ in milliseconds, Prometheus might expose it through a complex PromQL query resulting in a label like $latency_p95$, measured in seconds. Precisely, k6's *med*, *avg* and percentile latency metrics are mapped to the Prometheus equivalent. Performing this normalization step immediately after the test run means the collected data can easily be compared and visualized later.

5.5.4 Testing "Freshness": Time to Consistency

To assess the eventual consistency of the ES-CQRS architecture, a specialized test for the Freshness SLO was developed.[12] Unlike standard performance scripts, which measure the speed of isolated requests, this script is specifically designed to measure the synchronization delay between the command and query sides of the application. This delay, called eventual consistency, occurs because the write-side (Command) and read-side (Query) are strictly separated in CQRS.

The primary difference from a typical k6 test lies in the execution flow within the default function. Rather than executing a single call, this test executes two calls to the application. First, it performs a POST request to create a lecture and captures the resulting ID. After creating the lecture, the script performs sleeps for exactly 0.1 seconds, the threshold defined in the Freshness SLO. After this threshold, the application is expected to have synchronized the write- and read-side. Once the script wakes up from its sleep, it performs a GET request, attempting to fetch the newly created lecture.

To track the success rate of this request, the script introduces a custom Rate metric named *read_visible_rate*. By manually adding true or false to this metric based on whether the lecture was found, indicated by a response status of 200, the script generates a percentage of "fresh" requests inside the required threshold of 100ms. This provides a clear statistical view of how reliably the ES-CQRS system maintains its "fresh" data under varying levels of load.

[12] `performance-tests/k6/time-to-consistency/create-lecture/create-lecture-.js`

Chapter 6

Results (TODO)

		CRUD (ms)		ES-CQRS (ms)			
Metric	**Users**	**Mean**	**CI ±**	**Mean**	**CI ±**	**Speedup**	**Significance**
Avg	500	840.33	0.00	1.47	0.00	573x	***
Median	500	913.11	0.00	1.25	0.00	728x	***
P95	500	2146.04	0.01	2.85	0.00	754x	***
P99	500	2922.25	0.03	5.08	0.00	575x	***

Table 6.1: Statistical comparison of latency for GET /lectures (500 RPS) over 30 iterations. A Significance of *** indicates a p-value ≤ 0.001

Chapter 7

Discussion (TODO)

7.1 Analysis of results

7.2 Conclusion & Further work (TODO)

Bibliography

[1] P. A. Bernstein & E. Newcomer, *Principles of transaction processing* (The Morgan Kaufmann series in data management systems), en, 2nd edition. Burlington, MA: Morgan Kaufmann Publishers, 2009, ISBN: 978-1-55860-623-4.

[2] Apache Groovy project, *The Apache Groovy programming language - Processing XML*. Accessed: Jan. 8, 2026. [Online]. Available: `https://groovy-lang.org/processing-xml.html#_gpath`

[3] Apache Groovy project, *The Apache Groovy™ programming language*. Accessed: Jan. 8, 2026. [Online]. Available: `https://groovy-lang.org/`

[4] The Internet Society, *RFC 2616: HTTP/1.1*, 1999. Accessed: Dec. 27, 2025. [Online]. Available: `https://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf`

[5] Testcontainers, *Testcontainers*, en-us. Accessed: Jan. 8, 2026. [Online]. Available: `https://testcontainers.com/`

[6] I. Jacobs & N. Walsh, *Architecture of the World Wide Web, Volume One*, Dec. 2004. Accessed: Dec. 27, 2025. [Online]. Available: `https://www.w3.org/TR/webarch/`

[7] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," en, Ph.D. dissertation, University of California, 2000. Accessed: Jan. 8, 2026. [Online]. Available: `https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf`

[8] S. Tilkov, *A Brief Introduction to REST*, en, Dec. 2007. Accessed: Jan. 8, 2026. [Online]. Available: `https://www.infoq.com/articles/rest-introduction/`

[9] M. Richards, *Software Architecture Patterns*, en. O'Reilly, 2015, ISBN: 978-1-4919-2424-2. Accessed: Jan. 8, 2026. [Online]. Available: https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf

[10] M. Fowler, *Anemic Domain Model*, Nov. 2003. Accessed: Dec. 27, 2025. [Online]. Available: https://martinfowler.com/bliki/AnemicDomainModel.html

[11] E. Evans, *Domain-driven design: tackling complexity in the heart of software*, en. Boston: Addison-Wesley, 2004, ISBN: 978-0-321-12521-7.

[12] J. Martin, *Managing the data-base environment*, en. Englewood Cliffs, N.J.: Prentice-Hall, 1983, ISBN: 978-0-13-550582-3.

[13] B. Meyer, *STANDARD EIFFEL*, en, 2006.

[14] G. Young, *CQRS Documents by Greg Young*, en, 2010. Accessed: Dec. 26, 2025. [Online]. Available: https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf

[15] J. Gray, P. Helland, P. O' Neil, & D. Sasha, "The dangers of replication and a solution," en, in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1996. DOI: 10.1145/233269.233330 Accessed: Dec. 28, 2025. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/233269.233330

[16] S. Braun, S. Deßloch, E. Wolff, F. Elberzhager, & A. Jedlitschka, "Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study," en, in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, arXiv:2108.03758 [cs], Oct. 2021, pp. 1–11. DOI: 10.1145/3475716.3475771 Accessed: Dec. 27, 2025. [Online]. Available: http://arxiv.org/abs/2108.03758

[17] W. Vogels, "Eventually consistent," en, *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. DOI: 10.1145/1435417.1435432 Accessed: Dec. 28, 2025. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/1435417.1435432

[18] B. Michelson, "Event-Driven Architecture Overview," en, Patricia Seybold Group, Boston, MA, Tech. Rep. 681, Feb. 2006, p. 681. DOI: 10.1571/bda2-2-06cc Accessed: Jan. 2, 2026. [Online]. Available: http://www.customers.com/articles/event-driven-architecture-overview

[19] M. Fowler, *Event Sourcing*, Dec. 2005. Accessed: Nov. 13, 2025. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[20] M. Kleppmann, *Designing Data-Intensive Applications*, en. O'Reilly, 2017, ISBN: 978-1-4493-7332-0.

[21] R. Malyi & P. Serdyuk, "Developing a Performance Evaluation Benchmark for Event Sourcing Databases," en, *Vìsnik Nacìonal'nogo unìversitetu "L'vìvs'ka polìtehnìka". Serìâ Ìnformacìjnì sistemi ta merežì*, vol. 15, pp. 159–168, Aug. 2024, ISSN: 2524065X, 26630001. DOI: 10.23939/sisn2024.15.159 Accessed: Nov. 3, 2025. [Online]. Available: https://science.lpnu.ua/sisn/all-volumes-and-issues/volume-15-2024/developing-performance-evaluation-benchmark-event

[22] U.S. Securities and Exchange Commission, *17 CFR § 242.613 - Consolidated Audit Trail*, en, Aug. 2012. Accessed: Jan. 7, 2026. [Online]. Available: https://www.law.cornell.edu/cfr/text/17/242.613

[23] Committee on National Security Systems, *National Information Assurance Glossary*, en, Apr. 2010. Accessed: Jan. 7, 2026. [Online]. Available: https://web.archive.org/web/20120227163121/http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

[24] ATIS Committee, *ATIS Telecom Glossary - audit trail*, en, Mar. 2013. Accessed: Jan. 7, 2026. [Online]. Available: https://web.archive.org/web/20130313232104/http://www.atis.org/glossary/definition.aspx?id=5572

[25] Joint Task Force Interagency Working Group, "Security and Privacy Controls for Information Systems and Organizations," en, National Institute of Standards & Technology, Tech. Rep., Sep. 2020, Edition: Revision 5. DOI: 10.6028/NIST.SP.800-53r5 Accessed: Jan. 7,

2026. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf

[26] M. Fowler, *Audit Log*, Apr. 2004. Accessed: Nov. 13, 2025. [Online]. Available: https://martinfowler.com/eaaDev/AuditLog.html

[27] B. Beyer, C. Jones, J. Petoff, & N. R. Murphy, Eds., *Site reliability engineering: how Google runs production systems*, eng, First edition. O'Reilly, 2016, ISBN: 978-1-4919-2912-4.

[28] J. Nielsen, *Usability engineering*, en. Academic Press, Inc., 1993, ISBN: 978-0-12-518406-9.

[29] Broadcom, Inc., *Why Spring*, en, 2026. Accessed: Jan. 12, 2026. [Online]. Available: https://spring.io/why-spring

[30] C. Walls, *Spring Boot in Action*, en. New York: Manning Publications Co. LLC, 2016, ISBN: 978-1-61729-254-5.

[31] M. Deinum, D. Rubio, & J. Long, *Spring 6 Recipes: A Problem-Solution Approach to Spring Framework*, en. Berkeley, CA: Apress, 2023, ISBN: 978-1-4842-8648-7. DOI: 10.1007/978-1-4842-8649-4 Accessed: Jan. 12, 2026. [Online]. Available: https://link.springer.com/10.1007/978-1-4842-8649-4

[32] JetBrains, *Java Programming - The State of Developer Ecosystem in 2023 Infographic*, en, 2023. Accessed: Jan. 12, 2026. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023

[33] C. Bauer, *Java persistence with Hibernate*, en, Second edition. Shelter Island, NY: Manning Publications, 2016, ISBN: 978-1-61729-045-9.

[34] PostgreSQL Global Development Group, *PostgreSQL*, en, Jan. 2026. Accessed: Jan. 12, 2026. [Online]. Available: https://www.postgresql.org/

[35] PostGIS PSC, *PostGIS*, en, 2023. Accessed: Jan. 12, 2026. [Online]. Available: https://postgis.net/

[36] Oracle, *Jackson JSON processor*, en-US. Accessed: Jan. 19, 2026. [Online]. Available: https://docs.oracle.com/en/middleware/goldengate/core/23/ogglc/jackson-json-processor.html

[37] FasterXML, *Jackson Project Home @github*, original-date: 2011-10-19T05:28:40Z, Oct. 2025. Accessed: Jan. 19, 2026. [Online]. Available: https://github.com/FasterXML/jackson

[38] Axoniq, *Introduction (5.0)*, 2025. Accessed: Jan. 12, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/5.0/

[39] Axoniq, *Messaging Concepts (4.12)*, 2025. Accessed: Jan. 12, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/messaging-concepts/

[40] Axoniq, *Introduction (v2025.2)*, 2025. Accessed: Jan. 12, 2026. [Online]. Available: https://docs.axoniq.io/axon-server-reference/v2025.2/

[41] Axoniq, *Axon Server - Event Store & Message Delivery System*, en, 2025. Accessed: Jan. 12, 2026. [Online]. Available: https://www.axoniq.io/server

[42] Axoniq, *Command Dispatchers*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/command-dispatchers/

[43] Axoniq, *Infrastructure*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/infrastructure/

[44] Axoniq, *Query Dispatchers*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/queries/query-dispatchers/

[45] Axoniq, *Multi-Entity Aggregates*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/modeling/multi-entity-aggregates/

[46] Axoniq, *Command Handlers*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/command-handlers

[47] Y. Ceelie, *Set Based Consistency Validation*, en, Nov. 2020. Accessed: Jan. 23, 2026. [Online]. Available: https://www.axoniq.io/blog/2020set-based-consistency-validation

[48] Axoniq, *Event Handlers*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/events/event-handlers/

[49] Axoniq, *Event Bus & Event Store*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/events/infrastructure/

[50] Axoniq, *Subscribing Event Processor*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/events/event-processors/subscribing/

[51] Axoniq, *Streaming Event Processor*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/events/event-processors/streaming/

[52] Axoniq, *Saga Implementation*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/sagas/implementation/#_injecting_resources

[53] Axoniq, *Saga Associations*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/sagas/associations/

[54] *JUnit User Guide*. Accessed: Jan. 8, 2026. [Online]. Available: https://docs.junit.org/5.14.2/overview.html

[55] Johan Haleby, *REST Assured Documentation*, en. Accessed: Jan. 8, 2026. [Online]. Available: https://github.com/rest-assured/rest-assured/wiki/Usage

[56] Broadcom, Inc., *Production-ready Features :: Spring Boot*, 2026. Accessed: Jan. 19, 2026. [Online]. Available: https://docs.spring.io/spring-boot/reference/actuator/index.html

[57] Prometheus Authors, *Prometheus - Monitoring system & time series database*, en, 2026. Accessed: Jan. 19, 2026. [Online]. Available: https://prometheus.io/

[58] Prometheus Authors, *Overview | Prometheus*, en, 2026. Accessed: Jan. 19, 2026. [Online]. Available: https://prometheus.io/docs/introduction/overview/

[59] VMWare, Inc., *Micrometer Prometheus :: Micrometer.* Accessed: Jan. 19, 2026. [Online]. Available: `https://docs.micrometer.io/micrometer/reference/implementations/prometheus`

[60] Broadcom, Inc., *Metrics :: Spring Boot*, 2026. Accessed: Jan. 19, 2026. [Online]. Available: `https://docs.spring.io/spring-boot/reference/actuator/metrics.html#actuator.metrics.export.prometheus`

[61] Docker Inc., *What is Docker?* en. Accessed: Jan. 19, 2026. [Online]. Available: `https://docs.docker.com/get-started/docker-overview/`

[62] Docker Inc., *Writing a Dockerfile*, en. Accessed: Jan. 19, 2026. [Online]. Available: `https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/`

[63] Docker Inc., *What is Docker Compose?* en. Accessed: Jan. 19, 2026. [Online]. Available: `https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-docker-compose/`

[64] Grafana Labs, *Grafana k6*, en. Accessed: Jan. 19, 2026. [Online]. Available: `https://grafana.com/docs/k6/latest/`

[65] Grafana Labs, *Write your first test*, en. Accessed: Jan. 19, 2026. [Online]. Available: `https://grafana.com/docs/k6/latest/get-started/write-your-first-test/`

[66] D. Ingram, *Design – Build – Run: Applied Practices and Principles for Production-Ready Software Development*, en. 2009, ISBN: 978-0-470-25763-0. Accessed: Nov. 14, 2025. [Online]. Available: `https://www.oreilly.com/library/view/design-build/9780470257630/`

[67] Hibernate, *Envers - Hibernate ORM*, en. Accessed: Jan. 20, 2026. [Online]. Available: `https://hibernate.org/orm/envers/`

Appendix A

Source Code

The full source code for this thesis, including both apps, performance tests and markdown notes, is available at: `https://gitlab.mi.hdm-stuttgart.de/lk224/thesis`

Appendix B

Resources

Technology	Version
Java	25
JUnit	5.11.0
AssertJ	3.27.6
Axon Framework	4.11.3
Testcontainers Java Library, incl. child dependencies: PostgreSQL testcontainer & JUnit support	1.19.7

Table B.1: Version matrix for all technologies used throughout the work