Bachelorarbeit im Studiengang Medieninformatik

Performance, flexibility and traceability: Evaluating the impact of Event Sourcing and CQRS compared to traditional systems with an audit log.

Vorgelegt von Lukas Karsch

mit der Matrikelnummer 45259

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: "Performance, flexibility and traceability: Evaluating the impact of Event Sourcing and CQRS compared to traditional systems with an audit log" 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 im Anhang aufgelisteten KI-Werkzeuge wurden ausschließlich zum Lektorat verwendet. Anschließend wurde sichergestellt, dass der Inhalt mit den intendierten Aussagen noch übereinstimmt. 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. 5–7

API API stands for *Application Programming Interface*. It describes the public interface of a module or service, often exposed over a network. 34, 37, 44, 51, 57

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

Docker Open platform for developing, shipping and running containerized applications. 43, 60, 61, 64, 65

Dockerfile A text document containing a series of instructions used to assemble a Docker image. 43

HTTP HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 [2]. 3, 4, 35, 42, 44, 53, 65

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. 22, 23

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. 22, 23

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

Rich Domain Model Objects incorporate both data and the behavior or rules that govern that data. 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. 22

Test Configuration A test configuration is the combination of a test script and the target RPS.. 22

Testcontainer Testcontainers are a way to declare infrastructure dependencies as code using Docker [3]. 44, 60

Acronyms

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

AHF Attribute Hiding Factor. 26

AIF Attribute Inheritance Factor. 27

BASE Basically Available, Soft State, Eventual Consistency. 8

CBO Coupling Between Objects. 17, 27, 28

CF Coupling Factor. 27

CK Chidamber and Kemerer Metrics. 17, 27

CLF Clustering Factor. 27

CQRS Command Query Responsibility Segregation. 1, 9–11, 13, 16–18, 20, 31, 34, 35, 38, 40, 57, 65

CQS Command And Query Separation. 9

CRUD Create Read Update Delete. 1, 7, 13, 18–20, 30, 31, 34, 35, 37, 45, 47, 49, 53, 57, 58

DAO Data Access Object. 5

DDD Domain Driven Design. vi, 5, 6, 11, 17, 38, 39

DIT Depth of Inheritance Tree. 17, 27

DTO Data Transfer Object. 9, 44, 47, 48, 53, 60

ES Event Sourcing. v, 1, 11–13, 17–20, 31, 34, 35, 57, 58

HATEOAS Hypermedia as the engine of application state. 3

HTML HyperText Markup Language. 3, 4

I/O Input / Output. 26

JMX Java Management Extensions. 42

JPA Jakarta Persistence API. 37, 45, 47, 51, 53

JPQL Java Persistence Query Language. 37

JSON JavaScript Object Notation. 3, 4, 37, 56

LCOM Lack of Cohesion in Methods. 17, 28

MHF Method Hiding Factor. 26

MIF Method Inheritance Factor. 27

MOOD Metrics for Object-Oriented Design. 26, 27

NOC Number of Children. 17, 27

ORM Object-relational Mapper. 37

PF Polymorphism factor. 27

POJO Plain Old Java Object. 37

RF Reuse Factor. 27

RFC Response for a Class. 17, 28

RPS Requests Per Second. vi, 21–23, 62, 67

RQ Research Question. 16, 21, 26, 28, 29

SLA Service Level Agreement. 33

SLO Service Level Objective. 20, 24, 33, 65

SSH Secure Shell. 64

URI Uniform Resource Identifier. 3

VM Virtual Machine. 60, 61, 64, 79

VU Virtual User. 44, 64

WMC Weighted Methods per Class. 27, 28

WWW World Wide Web. 3

XML Extensible Markup Language. 4, 13

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-research questions (RQ) are addressed:

RQ 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?

RQ 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?

RQ 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 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 / configurations to get the best performance
- Implement mechanics to support schema evolution
- Implement full-grade application using security best practices, etc -> custom RequestContext with own "authorization" methods
- Creating a "real", usable application with GUI

1.4 Structure of the thesis (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 [4].

Web APIs are interfaces that enable applications to communicate. They use HTTP as a network-based API [5, p. 138]. Modern APIs typically follow REST principles. REST stands for "Representational State Transfer" and describes an architectural style for distributed hypermedia systems [5, 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. Table 2.1 presents examples for resources and URIs which could be derived from them:

The "Hypermedia as the engine of application state (HATEOAS)" principle states that resources should be linked to each other. Clients should be able to

Resource Type	**ID**	**URI**
Book	1	`http://example.com/books/1`
Book	2	`http://example.com/books/2`
Author	100	`http://example.com/authors/100`

Table 2.1: Resource Metadata Mapping

control the application by following a series of links provided by the server [6].

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 [6]: a web browser might request HTML, while another server or application might request XML or JSON.

TODO: replace Tilkov source, its a blog post

2.2 Layered Architecture Foundations

Richards [7] describes Layered Architecture as 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 layers described in Table 2.2.

Layer	**Responsibility**
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 and fetches 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.

Table 2.2: Layers in Layered Architecture [7, Chapter 1]
.

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 [7, Chapter 1].

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 [7, Chapter 1].

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 [7, Chapter 1].

Figure 2.1 illustrates the flow of a request through a typical Layered Architecture.

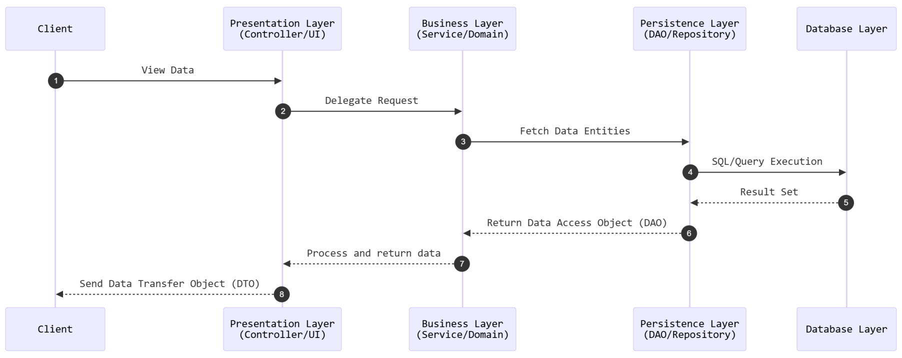
Client
Presentation Layer (Controller/UI)
Business Layer (Service/Domain)
Persistence Layer (DAO/Repository)
Database Layer
1 View Data
2 Delegate Request
3 Fetch Data Entities
4 SQL/Query Execution
5 Result Set
6 Return Data Access Object (DAO)
7 Process and return data
8 Send Data Transfer Object (DTO)

Figure 2.1: Layered Architecture sequence diagram, adapted from [7, p. 6].

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 [8] 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.

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 [9, 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 [9, 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. [9, pp. 12, 13]

As described in section 2.2, Layered Architecture organizes code into technical tiers and is typically built on Anemic Domain Models, often resulting in the *big ball of mud* antipattern: source code which is unorganized and is missing clear responsibilities and relationships. [7, p. V] In contrast, 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. [9, p. 80]

Using a Rich Domain Model does not mean that there should be no layers, the opposite is the case. Evans [9] advocates for using layers in domain driven designs. He proposes the layers presented in Table 2.3.

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

Table 2.3: Layers in DDD [9, p. 53]

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 [9, 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 [9, pp. 70–72].

Aggregates are clusters of associated objects (Entities and Value Objects) that are treated as a single unit during data changes. Each Aggregate has a designated *Aggregate Root*, which is the only member of the Aggregate that external objects are allowed to hold a reference to. This structure ensures that all business invariants and consistency rules within the Aggregate boundary are enforced, as all changes must go through the root [9, p. 89].

Evans [9, 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 and ACID

Layered architectures often follow a *Create Read Update Delete (CRUD)* architecture. CRUD is an acronym coined by Martin [10, Chapter 21]. These four actions can be applied to any record of data.

The state of domain objects in a CRUD architecture is often mapped to tables in 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 [10, Chapter 21].

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. [10, 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 Eventual Consistency

Gray et al. [11] 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 Basically Available, Soft State, Eventual Consistency (BASE) model [12] 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 over time through the use of synchronization mechanisms rather than being strongly consistent at all times [13], [14]. This creates an inconsistency window in which data is not consistent across the system. During this window, stale data may be read [14]. This concept is visualized in Figure 2.2.

Consistency guarantees described by Terry et al. [15] such as *Read Your Writes*, *Monotonic Reads*, *Writes Follow Reads* or *Monotonic Writes* can mitigate these problems by providing applications with a view of the database that is consistent with their own actions. By associating operations with a session, these guarantees ensure that a client's sequence of reads and writes remains logical and predictable, even when they interact with various, potentially inconsistent servers. For instance, these strategies can ensure that a client always perceives their own updates or sees an increasingly up-to-date version of the database. However, these guarantees are applied on a per-session basis and do not resolve the issue for other clients. Users in different sessions may still observe stale data or inconsistent orderings until the system achieves eventual consistency through successful synchronization.

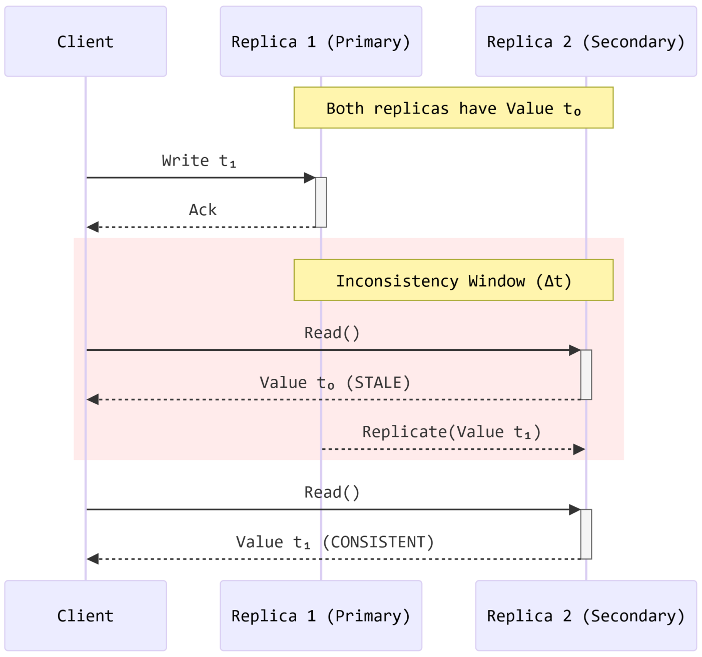

Figure 2.2: Sequence diagram illustrating the inconsistency window (Δt) in eventually consistent systems where read results depend on the selected replica. Based on [14]

2.6 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 [16, 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 [17, 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 [17, 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 [17, 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, creating an eventually consistent system as described in section 2.5.

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 [17, p. 23].

Figure 2.3 presents the CQRS architecture in a flowchart, visually highlighting the separation of read- and write-side.

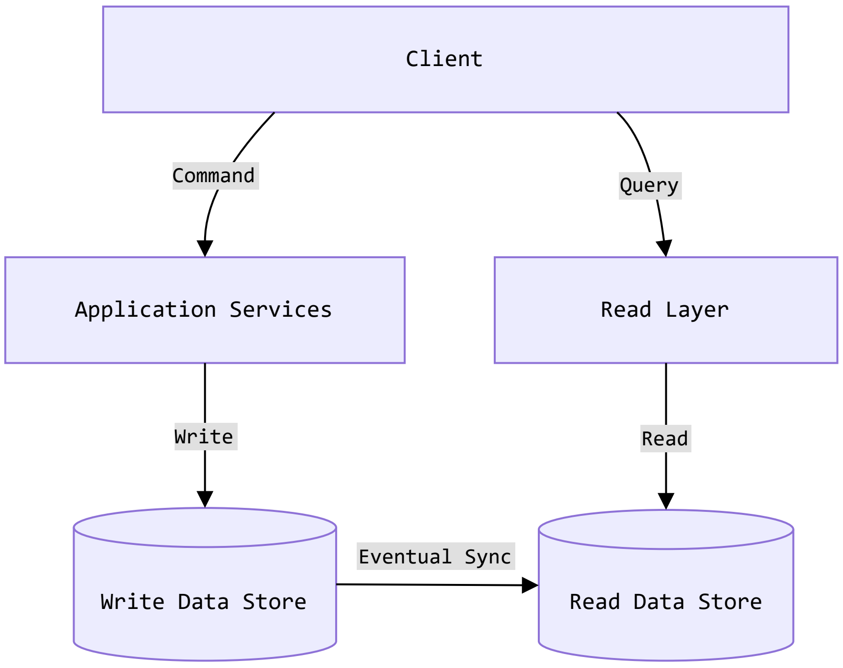

Figure 2.3: CQRS Architecture, adapted from [17, p. 24]

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 improved horizontal scalability and failure 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 Sourcing ensures 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 CQRS pattern described in section 2.6, which physically divides the data structures used for reading from those used for writing state changes [17, p. 50]. TODO: Zusammenhang CQRS und DDD (Aggregates) unklar

Figure 2.4 demonstrates the process of state reconstruction in an event-sourced CQRS system using Domain Driven Design (DDD). To process a new command, the Aggregate must first be rehydrated to its current state. It (e.g., $ID = 1$) is rehydrated by iterating through the event stream and applying those events that target its unique identifier. Events belonging to other aggregates are ignored during this process. Through applications of the events, the Aggregate transitions from its initial blank state to its current, functional state.

To optimize the reconstruction of state, developers often employ *rolling snapshots*. These snapshots represent a serialized state of an aggregate at a specific point in time, allowing the system to restore the state quicker and only replay the delta of events that occurred after the snapshot was taken [17, p. 20]. This approach caps the maximum number of events to be processed during Aggregate rehydration, providing a performance gain. It is worth noting that the event stream stays intact and is not impacted by a snapshot. Figure 2.5 illustrates the state reconstruction process using snapshots. Instead of replaying the entire event stream from the initial state, the system loads the most recent serialized state from the Snapshot Store. Therefore, only the delta needs to be applied to the Aggregate.

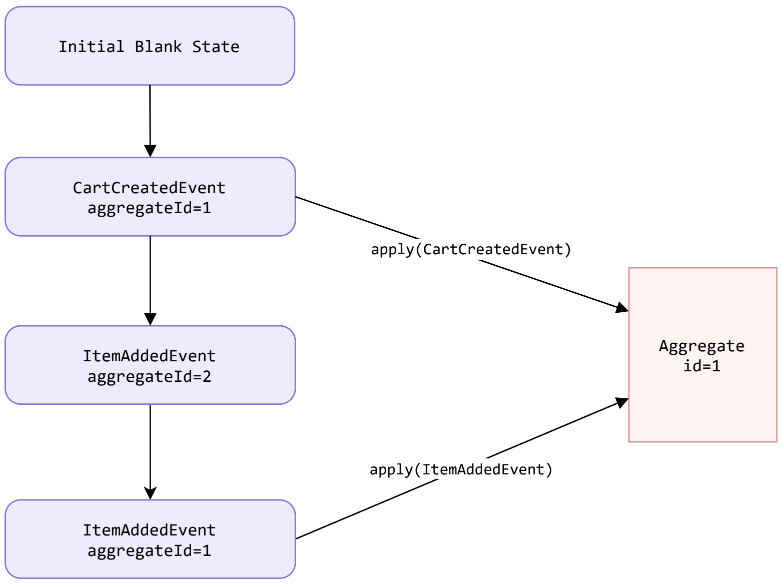
Initial Blank State
CartCreatedEvent aggregateId=1
ItemAddedEvent aggregateId=2
ItemAddedEvent aggregateId=1
apply(CartCreatedEvent)
apply(ItemAddedEvent)
Aggregate id=1

Figure 2.4: Applying events to rehydrate Aggregates in Event Sourcing.

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 [22, pp. 4, 17]. 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 [23].

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 [24]. In information security, the audit log stores a record of system activities, enabling the reconstruction of events [25]. 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 [26, p. 266].

Fowler [27] 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

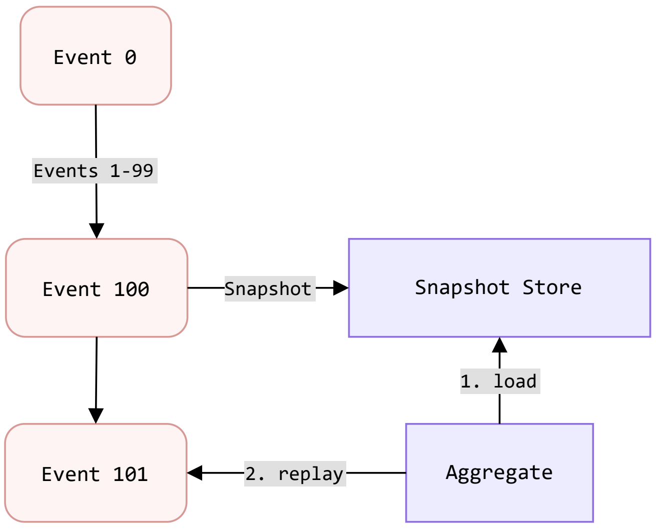

Figure 2.5: Snapshots in CQRS / Event Sourcing. Only the events between the most recent snapshot and the most recent event need to be applied to rehydrate the Aggregate [17, p. 20]

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 [28].

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 [20, pp. 457, 531]. The *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 intent preservation 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 legal compliance, as they eliminate the "information loss" associated with traditional database overwrites. In the context of the legal requirements discussed previously, the event stream serves as a sequence of actions that satisfies the need for a verifiable history [20, p. 531].

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 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 [20, Chapter 11]. 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) [17]. If an event is not recorded, the state change never occurred [20, p. 460]. 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 partitioning 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.

Literature

- Abbott & Fisher [29]
- Jogalekar & Woodside [30]
- Kleppmann [20]

Chapter 3

Related Work

This chapter places the thesis within existing knowledge regarding distributed systems and architectures. It evaluates established theoretical frameworks and empirical studies to provide a foundation for assessing the performance, structural complexity, and auditing capabilities of different architectural patterns. To align with the research sub-questions, the review is organized into three sections: performance and scalability (RQ 1), maintainability and flexibility (RQ 2), and historical traceability (RQ 3).

3.1 RQ 1 — Performance and Scalability

Scalability and performance are critical considerations in modern distributed system design. Kleppmann [20] provides a foundational understanding of these concepts, defining scalability not simply as the ability to handle increased load, but as the capacity to maintain performance as load parameters grow. [20, Chapter 1] To quantify scalability characteristics, Jogalekar & Woodside [30] established a formal mathematical framework for scalability, defining a metric based on cost-effectiveness where productivity is a function of throughput and quality of service (specifically response time). Their work emphasizes that good scalability results from the system architecture and the scaling strategy.

Building on these theoretical foundations, recent studies have empirically evaluated the impact of architectural patterns like CQRS on system performance. Jayaraman & Mishra [31] investigated the implementation of CQRS in large-scale systems, using performance benchmarks to demonstrate that separating read and write models allows for independent optimization, reducing response times and improving throughput compared to monolithic architectures. Their research, based on simulations in a controlled environment, indicated that CQRS systems, particularly when using read replicas and caching, maintained superior response times under high-concurrency workloads where traditional monolithic systems faced bottlenecks.

Further supporting these findings, Hruzin & Lytvynov [32] presented a comparative analysis of a task-tracking system migrated from a traditional DDD architecture to one using CQRS and Event Sourcing. The study found that while the transition increased code and infrastructure complexity, it yielded performance gains, increasing bulk read operation speeds by a factor of 6. Write operation performance varied after making the transition to CQRS, with some operations demonstrating acceleration and others experiencing a slowdown.

Generally, there exists a consensus in the reviewed literature that decoupling Command processing from Query processing enables more granular scaling strategies, allowing systems to handle load more efficiently than traditional CRUD-based approaches. [31], [32] TODO: more literature

3.2 RQ 2 — Architectural Complexity, Maintainability, Flexibility

Recent literature provides empirical evidence regarding the structural impact of adopting CQRS and Event Sourcing. The work conducted by Hruzin & Lytvynov [32], described in section 3.1, also showed specific results regarding code complexity: the migration increased the total number of classes from 47 to 213, while the overall cyclomatic complexity of the system decreased from 534 to 522. This suggests that while additional infrastructure classes are required, the individual modules become simpler. The authors argue that the separation of Commands and Queries simplifies debugging and extension of the application.

To objectively measure such qualities, researchers rely on established object-oriented metrics. Singh et al. [33] employed statistical approaches, including hypothesis testing and linear regression, to correlate the Chidamber and Kemerer Metrics (CK) suite with software quality. Their analysis established that metrics such as Depth of Inheritance Tree (DIT) and Number of Children (NOC) impact quality, but Coupling Between Objects (CBO) demonstrated the strongest negative correlation with software quality. Basili et al. [34] present a validation of the CK metrics, conducting experiments to test how these metrics correlate with defects in a program. Their results show that DIT, Response for a Class (RFC) and NOC were most significant for predicting defects in a program, while Lack of Cohesion in Methods (LCOM) was shown to be insignificant. TODO: quickly mention other metrics?

Kleppmann highlights the evolutionary advantages of Event Sourcing. He argues that systems gain the flexibility to derive new read-optimized views from the same history without the need for schema migrations. [20, pp. 461, 462]

While Kleppmann emphasizes the architectural freedom to evolve views on existing data, Overeem et al. [35] discuss 19 event-sourced systems and point out that many engineers struggle with *schema evolution* in Event Sourcing. Schema evolution is the process of changing the schema of events. The challenge is that the immutable history of the event log must remain compatible with evolving business logic, which turns the flexibility of the read-side into difficulties on the write-side. Schema evolution is not a goal of this thesis. However, these characteristics will be considered when comparing the architectural flexibility of event-sourced systems to traditional CRUD systems.

In terms of maintainability, while Jayaraman & Mishra [31] noted increased performance and scalability when using CQRS with Event Sourcing, they also found higher maintenance complexity and higher operational costs.

3.3 RQ 3 — Historical Traceability

Kleppmann [20, pp. 457, 531] mentions that using Event Sourcing (ES) makes it easier to reproduce bugs and diagnose unexpected behaviors by replaying the event log. Monagari [36] cites empirical data showing that financial institutions using Event Sourcing reduced incident resolution time from 4.2 hours to 23 minutes by replaying events to reproduce exact system states, though the primary data could not be verified during literature review.

Gantz [37] defines IT auditing as the process of validating controls to protect assets and information. Historically, audit logs were often viewed as "disposable" data, overwritten regularly to save space [22, p. 3]. However, Maier argues that regulatory pressure and the need for accountability have made log retention critical for event reconstruction and forensic analysis [22, pp. 4, 17].

Effective auditing requires "evidence": information that auditors can verify against established criteria. [37, p. 155] Gantz emphasizes that the reliability of a system depends on its ability to produce accurate evidence of past states and operations. [37, p. 4]

The standard audit logging approach employed in traditional CRUD systems has several downsides which can be found in the literature. Kleppmann [20, p. 531] notes that even if transaction logs are captured, they reveal what changed, but not necessarily *why*. The application logic that decided on the mutation is transient and context may be lost. Maier [22, pp. 23, 24] points out that reconstructing a security incident in a distributed environment is time-consuming. It may require manually correlating logs from separate systems with different formats and time synchronizations.

In contrast, Event Sourcing treats state changes as an immutable sequence of events. Helland [28] emphasizes that the event log *is the truth*, and any application state is derived from it. Because the log is append-only, history is never overwritten.

This allows for deterministic replay of events, making any historic state perfectly reconstructible. Kleppmann [20, p. 531] also highlights that events typically capture user intent.

In terms of compliance and integrity, Kleppmann [20, p. 531] points out that the integrity of an event store can be verified through hashes, making event-sourced systems highly reliable.

While the reviewed literature establishes the superiority of event-sourced systems in terms of data integrity, no empirical research comparing the actual computational efficiency of state reconstruction between Event Sourcing and traditional CRUD audit logs could be found. Consequently, this thesis aims to fill that gap by quantifying the performance differences between these architectures when reconstructing historical states.

Chapter 4

Methodology

The goal of this thesis is to provide a comparison of CRUD and CQRS / ES architectures. At the core of this comparison, two prototypes of an identical application will be implemented and examined. These implementations will display the same interface and behavior.

To answer the primary research question based on the three defined sub-research questions, this thesis employs three phases. First, functional and non-functional requirements for the project and the applications will be described. Next, the applications can be implemented according to these requirements. The implementations can then be compared based on several criteria.

This chapter describes the method, detailing each phase and finally presenting the comparison methods used to assess performance, scalability, flexibility and traceability.

4.1 Phase 1: Requirement Analysis

Before implementing, it is necessary to conduct a requirement analysis. This requires defining functional requirements, regarding the domain and business logic of the application. Afterward, non-functional requirements are specified. These include performance goals (SLOs), auditability and observability of the applications.

The defined requirements then serve as a contract which both implementations adhere to, ensuring a fair comparison.

4.2 Phase 2: Implementation of prototypes

Once requirements for the application are defined, the prototypes can be implemented. As mentioned above, these prototypes should have an identical interface and exhibit identical state transitions and error behaviors.

4.3 Phase 3: Evaluation and Comparison

After implementing the prototypes, they can be compared and evaluated based on the defined research questions. This comparison can be separated into three aspects: performance and scalability, architectural flexibility and traceability.

4.3.1 Performance and Scalability

To answer RQ 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, environmental constraints, and the data collection methodology are described in this subsection. Finally, perspectives on scalability are presented, which can be derived by analyzing results of load tests.

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 generally characterized using "load parameters," which vary depending on the system's nature. For this thesis, 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 typically measure latencies by executing request filters which only run once the request starts being processed. Therefore, server-side latency metrics often exclude the "waiting time" a request spent in the queue, leading to an overly optimistic view of performance. Consequently, the client-side latencies are more relevant to measure user-perceived performance.

Kleppmann [20, p. 16] also mentions that the load-generating client must continuously send requests without waiting for previous ones to complete, to simulate realistic concurrent user behavior. In load-testing frameworks, this is often called an "open model". If a client waits for its request to complete before sending the next one, queues on the server are kept artificially short. This is typically called "closed model".

This thesis adopts the approach of keeping system resources constant while measuring performance fluctuations under varying load intensities, as opposed increasing system resources to keep unchanged performance. These two perspectives are described by Kleppmann [20, p. 13]

Abbott & Fisher [29, p. 264] mentions another way of looking at load testing: In positive testing, load is increased until an application is overwhelmed, while negative testing takes away system resources such as memory, threads or connections, until the application degrades.

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. Exactly 50% of requests experience the median latency or less, while the other half experiences a higher latency. However, to capture the experience of users facing high delays, it is critical to measure "tail latencys" via a high percentile like P95 or P99. 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]

Test Execution

To obtain significant results, each *test configuration* (the combination of a test script and target RPS) should be executed a sufficient number of times. [29, p. 259] Between each execution, the application and its dependencies should be restarted to ensure independent results.

Comparability and Environment

Section 4.3.1 describes that test configurations must be repeated an appropriate number of times to achieve accurate results. Additionally, the environment in which the tests are run must be controlled. It should be reproducible, identical for each test run and experience no disturbances like system updates while tests are being executed. [29, Chapter 17]

Collected Metrics

All metrics collected during load testing are listed in Table 4.1. 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 section 4.3.1, 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 a metric extracted from the event store (Axon Server). It is only recorded for the ES-CQRS application. *postgres_size*, the size of the relational database 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 4.1: All metrics collected during load testing

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.

4.3.2 Scalability Dimensions

Several aspects are relevant to the system's growth.

- Throughput scalability: handle increasing amount of operations

- Data volume: System behavior as state accumulates, audit logs and event store grow, millions of records
- Organizational scalability: how does architecture influence applications and their possibilities for scaling? CQRS might be more suited to be split up into small teams, microservice architecture, etc

Quantifying Scalability through a Productivity Metric

Jogalekar & Woodside [30] propose measuring scalability by looking at a system's "productivity" $F(k)$ at different scale factors k_i. They argue that productivity is a measure of value versus cost.

This is the proposed productivity metric:

$$F(k) = \lambda(k) \cdot f(k)/C(k)$$

Here, k is the scale factor. This may be the number of processors or number of users. $\lambda(k)$ is the system's throughput, measured in responses per second. $f(k)$ is the average value of each response. The value of a response must be determined per system. For this thesis, it will be based on SLOs defined in subsection 5.2.1. $C(k)$ is the system's running cost per second at scale k. The running cost can be defined through resource consumption, e.g. CPU and RAM usage. The authors mention that several factors can be used for the cost function, they also propose the actual cost of hardware, e.g. when running on the cloud.

From the given productivity metric, a scalability metric can be calculated. To determine if a system is scalable, its productivity at two different scales is compared.

$$\psi(k_1, k_2) = \frac{F(k_2)}{F(k_1)}$$

When ψ is close to or greater than 1, the system is considered to be scalable. [30, pp. 4–6]

Headroom and Resource Saturation

Abbott & Fisher [29, Chapter 11] defines headroom as the amount of free capacity existing within a system before it begins to suffer from performance degradation or outages. To accurately calculate headroom, one must subtract the current usage from the "ideal usage" of a component's maximum capacity, while also factoring in expected future growth and any planned optimization projects. This metric is used to ensure a system can handle future increases in load without failure.

Resource saturation occurs when a system approaches its maximum capacity, a state that should be avoided because system behavior becomes unpredictable near 100% utilization. As resources saturate, issues such as "thrashing" (excessive swapping of data between memory and disk) emerge, causing performance to plummet dramatically rather than degrade linearly. To prevent saturation and maintain stability, Abbott suggests adhering to an "ideal usage percentage," generally recommending that systems run at only 50% to 75% of their total capacity to account for demand variability and estimation errors.

Architecture that scales

Scalability is influenced by the alignment between the technology and the organization that builds it. Abbott & Fisher [29, p. 2] argues that scalability issues often originate with people and management rather than technology alone, as the organizational structure dictates communication flows and decision-making efficiency. Consequently, an architecture's ability to scale depends on whether the organization is structured to support growth without creating bottlenecks.

A primary factor in architectural decisions is the choice to "scale out" (horizontal scaling) rather than "scale up" (vertical scaling). *Scaling out* relies on adding more units of commodity hardware, which is cheaper, standard equipment, to handle increased load, whereas *scaling up* depends on purchasing larger, faster, and more expensive individual systems. According to Abbott & Fisher [29, pp. 203, 207], scaling out using commodity hardware should be preferred, because it demonstrates that a system is not viable through faster, more expensive hardware. Instead, a system designed to be horizontally split is more resilient as it does not rely on third-party technologies or technological progress.

Code architecture further impacts scalability through the implementation of fault isolation, asynchronous design, and statelessness. Fault isolative architectures, often described as "swim lanes," ensure that a failure in one component does not propagate to others, preserving the availability of the broader system. [29, Chapter 21] Additionally, asynchronous designs are more tolerant of slowdowns, as the speed of the entire system is not determined by its slowest component. This makes asynchronous systems easier to scale. [29, p. 205] Finally, stateless systems allow requests to be distributed freely across any available server, making them more scalable. Abbott & Fisher [29, p. 206] argues that stateful applications should be avoided, when possible.

The architectural factors going into scalability which were described in this section will be used to give a precise comparison between the systems when discussing results. TODO this sentence

Load Testing Scenarios

To accurately assess the performance of the architectures, a diverse suite of load tests should be employed. The selection of specific endpoints for testing follows the methodology outlined by Abbott & Fisher [29, pp. 260, 265], prioritizing scenarios based on usage volume, business criticality, and resource intensity.

When choosing which endpoints are relevant to test, Abbott advocates for a "Pareto Distribution" (80/20 rule) and argues that 20% of tests will provide developers with 80% of information. Endpoints can be ordered by criticality, a process which identifies the endpoints that are most likely to affect performance. Endpoints can also be ranked by usage of Input / Output (I/O) necessary, locking or synchronous calls. After ordering the endpoints as described, the 80/20 rule can be used to select one read and write per "complexity level", eliminating the need to load test every single endpoint.

4.3.3 Flexibility - Architectural Metrics

This section describes various architectural metrics established in literature which are used to assess the flexibility, quality and evolutionary potential of a software architecture. These metrics serve as a basis to answer RQ 2, using static analysis to compare the two architectures.

Coupling and Stability

Coupling metrics are based on graph theory. They are used to measure how dependent components are on each other. *Afferent coupling* measures the number of incoming connections to a code artifact, indicating how many other components depend on it. *Efferent coupling* calculates the number of outgoing connections to other artifacts, reflecting the component's dependency on external code. The concepts of afferent and efferent coupling were first introduced by Yourdon et al. [38] in 1978.

Instability assesses the volatility of a code base by calculating the ratio of efferent coupling to the sum of all coupling, indicating how easily a component breaks when changed. *Abstractness* determines the ratio of abstract artifacts, such as interfaces and abstract classes, to concrete implementations within a module. Finally, the "*Distance from the Main Sequence*" combines abstractness and instability to determine if a component is optimally balanced or falls into problematic "zones of uselessness or pain". [39, pp. 261–268]

MOOD metrics

Metrics for Object-Oriented Design (MOOD), introduced by Brito e Abreu & Carapuça [40], provide a summary of the overall quality of an object-oriented project. The *Method Hiding Factor (MHF)* represents the ratio of hidden methods to total methods, indicating the level of encapsulation. The *Attribute Hiding Factor (AHF)* measures the ratio of hidden attributes to total attributes, serving as indicator for the system's encapsulation. The *Method Inheritance Factor (MIF)* calculates the ratio of inherited methods to the total available methods. The *Attribute Inheritance Factor (AIF)* determines the ratio of inherited attributes to the total attributes available in the classes. The *Polymorphism factor (PF)* measures the actual number of different polymorphic situations relative to the maximum possible distinct situations. The *Coupling Factor (CF)* evaluates the ratio of actual couplings not related to inheritance against the maximum possible number of couplings in the system. The *Clustering Factor (CLF)* measures the ratio of independent class groups, or "Class Clusters," to the total number of classes, identifying disjoint subgraphs within the system. Finally, the *Reuse Factor (RF)* measures the ratio of code reused from library code or inheritance to the total code written. All MOOD metrics are expressed in percentages, meaning they take a value between 0 and 1.

Complexity Metrics

A common metric for code complexity is cyclomatic complexity. Cyclomatic complexity can give a measure for code complexity at the function, class or application level by measuring the number of decision paths through a given component. [41, pp. 79, 80] It is distinguished between essential complexity (complexity necessary for the domain) and accidental complexity introduced through poor coding practices.

Connascence

Connascence is a concept described by Page-Jones [42]. It refines afferent and efferent coupling and describes a relationship where two components are coupled such that a change in one requires a modification in the other to maintain system correctness. There are two types of connascence: static connascence, which can be analyzed at the source code level, and dynamic connascence, evaluated at runtime. [41, pp. 48–53]

Chidamber and Kemerer (CK) Metrics

CK are the most well-known object-oriented suite of metrics, according to Chawla & Kaur [43, p. 162]. They are described in a work from 1994 by Chidamber & Kemerer [44]. *Weighted Methods per Class (WMC)* calculates the sum of the complexities of all methods within a class to determine how much effort is required to maintain it. The authors deliberately do not give a specific complexity function to make the metric as general as possible. *DIT* measures the maximum length from a specific node to the root of the tree, where deeper trees imply higher complexity. *NOC* counts the immediate subclasses of a class, indicating the potential influence on the design and level of reuse. *CBO* counts the number of other classes to which a specific class is coupled, serving as an indicator of sensitivity to changes in other parts of the design. *RFC* counts the number of methods that can be executed in response to a message received by an object, reflecting the class's complexity. *LCOM* measures the degree to which methods within a class reference different instance variables, where high values indicate a class does too many unrelated things.

Usefulness of Static Analysis

So, how can these static analysis metrics help to answer RQ 2? They offer a quantitative method to identify the fundamental structural differences between architectures by measuring how components are organized and connected. Metrics such as *CBO* and *Afferent/Efferent coupling* reveal the degree of dependence between modules. High coupling suggests a structure where changes in one area may have affects on large parts of the system, while low coupling indicates better modularity and isolation, leading to easier maintenance. [43] Additionally, cohesion metrics like *LCOM* and complexity measures such as *WMC* help determine if an architecture is composed of focused, manageable components or large, complex classes that attempt to do too much. [43]

Regarding long-term flexibility and evolution, metrics that evaluate stability and abstraction provide insights into how easily a codebase may change over time. For example, calculating *Instability* and *Abstractness* allows architects to visualize their *"Distance from the Main Sequence,"* highlighting which parts of the system may be becoming too rigid to change or too abstract to be useful. [41, pp. 45–47]

Critiques of Static Analysis methods

Richards & Ford [41, p. 44] offer a strong critique of the LCOM metric. They argue that the metric possesses "serious deficiencies" because it only measures the structural lack of cohesion (how methods and fields physically interact). The metric cannot determine if the pieces of code logically fit together. They also criticize complexity metrics like cyclomatic complexity, calling them "blunt". Their primary critique is that complexity metrics can not distinguish between essential complexity (complexity necessary for the domain) and accidental complexity introduced through poor coding practices. [41, p. 48] Finally, the authors express their critique of connascence, mentioning that it focuses too much on the low-level details of software coupling. They argue that developers should care more about *how* components are coupled (e.g. synchronous or asynchronous). [41, p. 53]

Drotbohm [45], a Spring engineer, offers his critique on the *abstractness* metric, highlighting that it mistakes the existence of interfaces with actual, useful abstractions.

Schema Evolution TODO

TODO: Finally, talk about schema evolution a little bit. Can we determine methods to decide which architecture is more flexible regarding schemas?

4.3.4 Traceability

This section describes how traceability will be compared; it serves as a basis to answer RQ 3. The traceability aspect will be compared via two perspectives. First, the *accuracy* of historic reconstruction is compared. Then, *efficiency* of reconstructions is evaluated and compared.

Accuracy of reconstruction

Several qualitative criteria for the accuracy of an historic reconstruction can be defined. *Source of truth integrity* defines how the system guarantees that the historic records match the current state. When a system needs to write state changes to two separate data stores, this creates a problem called "dual-write" problem in distributed systems.

Intent preservation assesses the ability of a system to distinguish between different business reasons for the same data change. TODO example

Schema resilience examines how historical data survives the evolution of the application's data structure. As business rules change, entities and database schemas evolve. The history must remain readable without being corrupted by modifications to schemas.

Efficiency

The primary objective is to measure how each architecture handles "time-travel" queries. These are requests that require the system to reconstruct a specific state from the past. To ensure a fair comparison, both applications will be tested under identical load testing scenarios.

Two distinct scenarios of varying complexity will be evaluated: A "simple state reconstruction" scenario will query the historic state of one entity in the system. A more complex state reconstruction scenario will require a broader view of the system's state to be reconstructed. This requires the system to cross-reference and reconstruct multiple entity relationships as they existed at that moment.

The project's existing load-testing infrastructure will be used for these tests. This allows to capture the same data points, such as latencies and resource consumption, which can then serve as a basis to compare the efficiency of historic reconstruction in both applications.

4.4 Limitations of the method (TODO sources)

While the research design aims for a rigorous comparison, several limitations must be acknowledged. These constraints arise from the controlled environment and the scope of the prototype implementations.

The evaluation is conducted on a single machine running virtual machines, which consequently live on the same network. Therefore, the prototypes are not tested on distributed clusters, which would be necessary to observe the real effects of horizontal scaling. This setup also eliminates common distributed system challenges, such as network partitions, varying latency, and partial failures. By running all components in a controlled environment, the results may not fully reflect the resilience or overhead of the architectures when deployed across a distributed system.

In the CRUD implementation, the audit log resides within the same relational database as the operational data. In a production-grade system, these might be separated to prevent the audit log's growth from impacting query performance, with the audit log being an additional side-car. Furthermore, because all components share the same underlying hardware resources during testing, the performance of one architectural layer (e.g., the event store) may influence another in ways that would not occur if they were isolated on dedicated hardware.

The load testing scenarios are artificial and focused on specific endpoints rather than complete user journeys. Abbott & Fisher [29, p. 267] suggest that the most accurate load profiles are derived from real-world application or load-balancer logs. Without such data, the tests rely on the estimation of relevant scenarios. Additionally, the prototypes lack complex access control patterns and multi-tenancy logic, which in a real-world scenario could introduce additional, non-negligible overhead.

The comparison focuses on the initial implementation and short-term performance. While data volume is simulated, the real long-term effects of a growing event store or the maintenance of complex migration scripts for schema evolution are evaluated theoretically rather than through operational data. This may lead to an incorrect view of the maintainability of certain architectural patterns.

Chapter 5

Requirement Analysis

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.

5.1 Functional Requirements

A functional requirement describes a specific behavior that a product must exhibit under specific circumstances. These requirements specify what the system *does* by detailing the capabilities and functions the solution must possess to allow users to perform their tasks. [46, p. 4] To ensure clarity regarding exactly how the system should behave, functional requirements are often written using patterns that include the keyword "shall," such as "The system shall let the user do something". [46, p. 109]

5.1.1 Project description

The applications will implement a course enrollment and grading system which might for example be used in universities. Professors can create courses and lectures which students can enroll to. These lectures can have assignments, which professors enter grades for. Once a lecture is finished, final grades and awarded credits can be calculated. Students are able to view their enrollments, grades and credits.

5.1.2 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. (TODO: aktuell ist Semester irrelevant)

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.

5.1.3 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.

5.1.4 Resulting functional requirements

Based on the domain described above, the following list presents a selection of functional requirements. As this thesis focuses on an architectural comparison, not every functional requirement going into the application is listed.

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

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

5.2 Non-functional requirements

A non-functional requirement, often referred to as a *quality attribute*, describes the quality or performance characteristics of a solution. [46, p. 4] Rather than defining *what* the product does, these requirements focus on *how well it functions.* They establish specific goals or constraints for the design and implementation, such as targets for security, availability, or response time, to ensure the system satisfies user expectations. [46, p. 67]

The following non-functional requirements (quality attributes) are defined.

5.2.1 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. [47, pp. 63, 65] This thesis attempts to define realistic SLOs to establish a "breaking point" for each architecture.

Following Nielsen [48, 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 shall 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 shall be reflected in the PostgreSQL read-model within $\leq$100ms to ensure eventual consistency remains imperceptible. While the read-side is eventually consistent, the command-side (write-model) shall maintain immediate consistency to ensure business rules are validated against the latest state.
- **Reliability SLO**: Both implementations shall maintain a failure rate of $<$0.1% under stress.

5.2.2 Auditing

Both systems need to be fully auditable. Every change to an entity must be reflected as a historical record. Historic states should be accurately reconstructible.

5.2.3 Observability

The systems must expose observability endpoints. These should be able to present information about the system internals. Precisely, CPU and RAM usage, database connections and size of data stores should be available. Additionally, request latencies should be exposed as histograms.

5.2.4 Consistency

To ensure the integrity of the operations, the following consistency requirements apply:

- **Write Consistency**: Both architectures shall provide immediate (strong) consistency for write operations. This ensures that any command (e.g., enrolling a student) is validated against the most recent state of the aggregate to prevent violations of business rules.
- **Read Consistency**: The CRUD implementation shall provide immediate consistency for reads. The ES-CQRS implementation may utilize eventual consistency for its read-models according to the Freshness SLO.

5.2.5 Contract

Both implementations must follow the same contract regarding endpoints, request and response schemas and state transitions. To ensure this, an extensive test suite shall be set up. While the internals of the implementations will be vastly different architecturally, they will both have the same public API, making it possible to send requests and verify the responses. Therefore, one test suite shall be developed which can be executed on both applications. The test suite should include integration tests for all API endpoints covering both regular and edge-case (error) scenarios to ensure that both implementations behave identically.

Chapter 6

Implementation

After defining functional and non-functional requirements, the two applications can be implemented. The implementation phase will be detailed in this chapter. After describing the utilized technologies, the contract tests ran on both applications are outlined. Next, implementation details of CRUD and ES-CQRS are given. Finally, the load tests are presented.

6.1 Endpoints

TODO: is this needed?

Table Table 6.1 presents a feature matrix, mapping endpoints to their functionality. As this thesis focuses not on the functionality of an application, but instead an architectural comparison, not all implemented endpoints are listed.

Endpoint	**Description**	**Response**
GET /lectures	Get lectures a student is enrolled in	200
POST /courses/create	Used by professors to create a course	201

Table 6.1: Selected endpoints implemented in the applications

6.2 Technologies

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

6.2.1 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" [49]. 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 Java Beans specification, a standard for creating reusable software components developed by Sun Microsystems [50], Spring extends this concept by taking full responsibility for the lifecycle and configuration of these objects [51, 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. [52, Chapter 1]. Spring offers support for several programming paradigms: reactive, event-driven, microservices and serverless. [49]

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. [53]

SpringBoot 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 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. [51, Chapter 1.1.2]

6.2.2 PostgreSQL

PostgreSQL [3] 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. [54] PostgreSQL is designed for a wide range of workloads and can handle many tasks thanks to its extensibility and large

[1]`https://spring.io/projects/spring-boot`
[2]`https://spring.io/projects/spring-framework`
[3]`https://www.postgresql.org/`

suite of extensions, such as the popular PostGIS extension for storing and querying geospatial data. [55]

6.2.3 JPA

Jakarta Persistence API (JPA),[4] formerly *Java Persistence API* is a Java specification which provides a mechanism for managing persistence and object-relational mapping. 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, supported by annotations, 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. TODO cite

When using JPA with SpringBoot by including the `spring-boot-starter-data-jpa` dependency, *Hibernate*[5] is used as implementation of the JPA standard. [56, Chapter 1]

6.2.4 Hibernate Envers (TODO)

Present Hibernate Envers here.

6.2.5 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 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 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. [57], [58]

[4]`https://jakarta.ee/specifications/persistence/`

[5]`https://hibernate.org/orm/`

[6]`https://github.com/FasterXML/jackson`

6.2.6 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. [59], [60]

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. [61], [62]

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.

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. [63], [64]

[7]`https://www.axoniq.io/framework`
[8]`https://www.axoniq.io/server`

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`. [65]

Aggregates

An aggregate is a core concept of 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. [66]

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. [67]

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 6.2.6. 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. [68]

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. [69]

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. [70]

Event processors take care of the technical part aspects of event processing. Axon's `EventBus` implementations support both subscribing and tracking event processors. [70] 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. [71]

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. [72]

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. [73]

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. [73] 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. [74]

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`.

6.2.7 SpringBoot Actuator

Spring Boot Actuator [9] 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 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. [75]

[9] `https://docs.spring.io/spring-boot/reference/actuator/index.html`

6.2.8 Prometheus

Prometheus [10] 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. [76], [77]

To collect and export Actuator metrics specifically for Prometheus, the `micrometer-registry-prometheus` dependency must be included in the classpath. [78] 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. [79]

6.2.9 Docker

Docker [11] 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. [80]

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. [81]

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. [82]

[10] `https://prometheus.io/docs/introduction/overview/`
[11] `https://docs.docker.com/`

6.2.10 k6

Grafana k6 [12] 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. [83]

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. [84]

6.3 Contract Test Implementation

To ensure the implementations adhere to the contract, a test suite is implemented in a separate module called `test-suite`[13], according to subsection 5.2.5. The test classes use the `JUnit 5`[14] testing framework and `REST Assured`[15] to send and assert HTTP requests. The test classes are abstract and must be extended by each prototype. Each test creates seed data through functions which are implementation-specific, meaning they have to be implemented by the respective implementation.

Necessary infrastructure for these tests 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. [3]

6.4 Shared Base Module

The `api` module serves as base module for both implementations[16]. This module defines DTOs, implemented using Java records, and controller interfaces exposing these record classes as part of their API. These interfaces are then implemented by each application. Additionally, the base `api` module contains utilities and shared

[12]`https://grafana.com/docs/k6/latest/`
[13]`test-suite/src/test/java/karsch.lukas`
[14]TODO link
[15]`https://rest-assured.io/`
[16]`api/src/main/java/karsch.lukas`

logic, for example Actuator endpoints to control the application's internal clock.[17]

6.5 CRUD implementation

This section presents the relevant aspects of the CRUD implementation,[18] mainly focusing on relational modeling using JPA and the audit log implementation. TODO focus is weird

6.5.1 Architectural Overview

The CRUD application is built upon a traditional Layered Architecture (described in section 2.2 and section 2.4), using on Controller, Service, and Repository layers. This classic separation of concerns ensures that responsibilities are clearly defined: Controllers handle incoming requests and responses, Services encapsulate the core business logic, and Repositories manage data persistence operations. This layered approach allows for independent development and testing of each layer.

While adhering to this layered structure, *feature slicing* was additionally applied to enhance modularity. The application's components are grouped into logical, domain-specific modules such as `lectures`, `users`, `courses`, and `stats`. This slicing enables all relevant code for a particular feature to reside within its dedicated package. This approach aims to reduce coupling between functional areas of the application and makes it easier to locate and modify code related to a specific feature.

6.5.2 Data Modeling

The CRUD implementation uses a normalized database in the Third Normal Form. TODO!! this still shows auditing class. Create a new diagram; also note that Envers' created auditing tables are not shown here. TODO: explain that modeling "data" here means entity / relational modeling. Make the whole section less detailed

Figure 6.1 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.

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:

[17]`api/src/main/java/karsch.lukas.time.DateTimeActuatorEndpoints`
[18]`impl-crud/src/main/java/karsch.lukas`

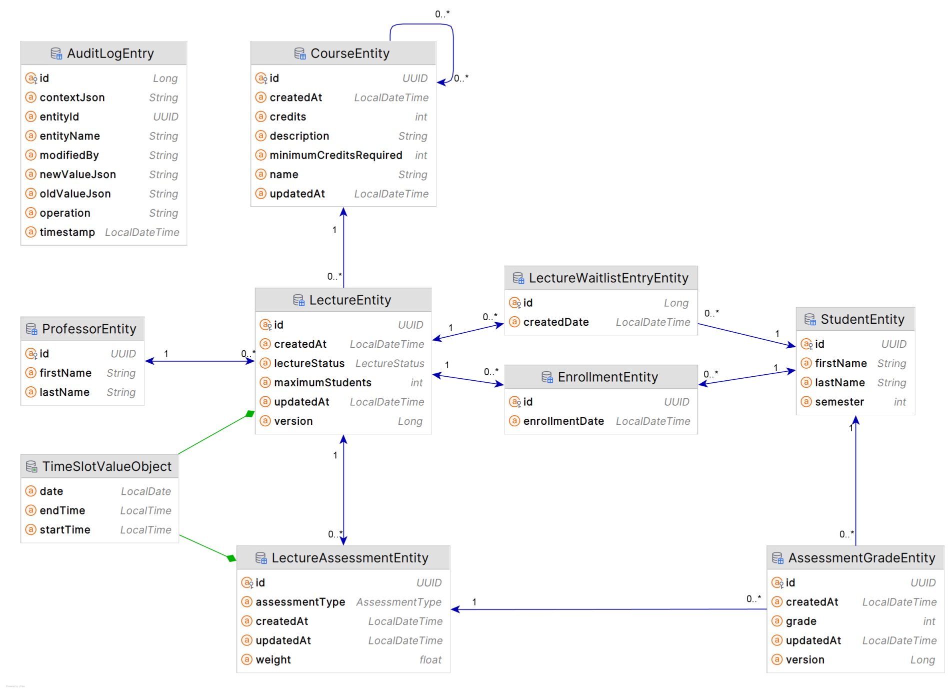

Figure 6.1: Entity Relationship Diagram for the CRUD App

$1 \longleftrightarrow 0..1$ 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 6.1 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 `LectureWaitlist-EntryEntity`, `EnrollmentEntity` and `LectureAssessmentEntity`. `LectureWaitlist-EntryEntity` 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.

These entities are implemented using SpringBoot's JPA integration. For example, an entity with a "One to Many" relationship can be implemented as presented in Listing 6.1.

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

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

Listing 6.1: 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.

6.5.3 Encapsulation and API Boundaries

The design of the CRUD prototype aims to create a clear separation between the internal logic and the external API. This is primarily achieved through the shared `api` module, described in section 6.4, which defines the DTOs that form the public contract for all communication. The intention is that internal data structures like database entities are always converted to these DTOs before being exposed, encapsulating the implementation details. This creates a boundary between clients and service internals.

However, achieving perfect encapsulation in a Layered Architecture proves challenging. While feature slicing was applied and DTOs provide a boundary at the controller level, the underlying entity classes are often referenced across different modules, especially when they have relationships with each other (for example, the `LectureEntity` and `CourseEntity`). Furthermore, services and repositories are frequently injected and called throughout the application, leading to a tightly coupled system where the boundaries between modules can become blurred. This may make it difficult to modify one part of the system without impacting others.

6.5.4 Auditing and Temporal Queries

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 simple code example is presented in Listing 6.2.

```
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 6.2: 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 [85, 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. **JPA Entity Listeners**: JPA's lifecycle events (`@PrePersist`, `@PreUpdate`, etc.) can be used to intercept changes. Inside event handling functions

designed for those events, it is possible to capture the changes and persist them in separate auditing tables. 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. Additional context can also be added through thread-local or request-scoped variables. [56, Section 13.2]

4. **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. These tables store historical data whenever a transaction is committed. Custom revision entities and change listeners can be implemented to capture additional context. [86] (TODO: is it true that Envers builds on JPA listeners?)

5. TODO: talk about CDC (change data capture) here or in subsection 2.8.1

The audit log in the CRUD prototype is implemented using Hibernate Envers. This solution was chosen because it seamlessly integrates with existing JPA entities to manage historical versions of data in dedicated audit tables.

Enabling Auditing on Entities

To track changes for a specific entity, it must be annotated with `@Audited`. In this implementation, a common base class `AuditableEntity`[19] is used to handle basic auditing metadata such as creation and modification timestamps using Spring Data JPA annotations. Listing 6.3 presents the state of an entity after enabling Envers auditing. Apart from the `@Audited` annotation, no changes are necessary, unless developers wish to exclude certain fields from auditing, in which case `@NotAudited` can be used.

```java
@Entity
@Audited
public class CourseEntity extends AuditableEntity {
    @Id @GeneratedUuidV7
    private UUID id;
    // all fields remain unchanged
}
```

Listing 6.3: Auditing configuration for the Course entity (`impl-crud/src/main/java/karsch.lukas.courses.CourseEntity`
)

[19] `impl-crud/src/main/java/karsch.lukas.audit.AuditableEntity`

Custom Revision Entity and Listener

While Envers provides a default revision table (storing only a revision ID and timestamp), a custom implementation is required to capture application-specific context, such as the user responsible for the change and a descriptive, optional context which allows capturing additional information about a change.

As shown in Listing 6.4, the `CustomRevisionEntity` extends Envers' `DefaultRevisionEntity` to include the fields `revisionMadeBy` and `additionalContext`.

```java
@Entity
@RevisionEntity(UserRevisionListener.class)
public class CustomRevisionEntity extends DefaultRevisionEntity {
    private String revisionMadeBy;
    private String additionalContext;
}
```

Listing 6.4: Custom Envers revision entity (`impl-crud/src/main/java/karsch.lukas.audit.CustomRevisionEntity`)

The association between a transaction and this metadata is handled by the `UserRevisionListener`. This listener intercepts the creation of a new revision and populates the fields by accessing the current request scope and a custom `AuditContext` bean. Its implementation is detailed in section 6.5.4.

Capturing Request-Scoped Context

To ensure the audit log contains meaningful information about why or by whom a change was made, the implementation utilizes Spring's `@RequestScope`. This annotation can be placed on beans, which will then be request-scoped, meaning they are re-created for each request. This annotation is used on two beans: `RequestContext`, holding information about the current user, and `AuditContext`, which is a bean able to capture additional context for auditing purposes. As `UserRevisionListener` is a Hibernate specific class living outside of Spring's managed environment, a static `getBean` method is used to access the relevant Spring beans.

```java
public class UserRevisionListener implements RevisionListener {
    @Override
    public void newRevision(Object revisionEntity) {
        CustomRevisionEntity rev = (CustomRevisionEntity)
            revisionEntity;

        if (isInsideRequestScope()) {
            RequestContext ctx =
                SpringContext.getBean(RequestContext.class);
            AuditContext audit =
                SpringContext.getBean(AuditContext.class);
            rev.setRevisionMadeBy(ctx.getUserType() + "_" +
                ctx.getUserId());
            rev.setAdditionalContext(audit.getAdditionalContext());
        } else {
            rev.setRevisionMadeBy("SYSTEM");
        }
    }
}
```

Listing 6.5: Implementation of the Revision Listener (`impl-crud/src/main/java/karsch.lukas.audit.UserRevisionListener`)

Global Auditing Configuration

Finally, the `AuditingConfig`[20] configuration class connects the application's custom time provider to the JPA auditing infrastructure. This ensures that both the standard `createdAt` fields and the Envers revision timestamps are synchronized with the application's internal clock, which is essential for consistent testing. Additionally, the configuration connects the application's request context to the auditing infrastructure, providing information about the current user. In a full-grade application, Spring security would provide the user context, though for this project, a simpler solution was preferred, as described in (TODO).

Reconstructing Historic State (TODO refine language)

Envers stores its revision data and historic records in specific auditing tables. While those hold the information necessary to reconstruct historic state, it is interesting to examine how well this state can *actually* be recreated. Imagine a service method with the purpose of returning the history of grade changes made to one grade. Envers provides a specific API which can be queried to reconstruct historical state. The mentioned service method is implemented in `StatsService`, as shown in Listing 6.6.

First, JPA's entity manager is used to obtain an instance of the AuditReader class, which provides methods to create historic queries. Using the `reader.createQuery()` method, it is possible to create a query instance by matching a specific class for which revisions shall be fetched, as well as adding a filter to match the relevant entity using its ID.

In this use-case, a date filter is part of the API. Envers enables developers to add additional matchers based on revision properties. In this case, the revision

[20]`impl-crud/src/main/java/karsch.lukas.audit.AuditingConfig`

property timestamp is used to define lower and upper date bounds, inside which the changes are relevant.

Once the query is built, the result list can be fetched, which is a list containing arrays of objects. More precisely, though not reflected by the type, each list entry is a tuple. Its first value is the historic entity, its second value is the revision entity which was created for this specific revision. Because a custom revision entity is registered, the type of this revision entity is CustomRevisionEntity.

```java
public GradeHistoryResponse getGradeHistory(
    UUID studentId, UUID assessmentId) {
    var assessment = fetchAssessment(assessmentId);
    var grade = fetchGrade(assessmentId, studentId);

    AuditReader reader = AuditReaderFactory.get(entityManager);

    AuditQuery query = reader.createQuery()
            .forRevisionsOfEntity(AssessmentGradeEntity.class,
                false, true)
            .add(AuditEntity.id().eq(grade.getId())); // match by
                entity ID

    if (startDate != null) {
        query.add(AuditEntity.revisionProperty("timestamp").gt(
                startDate.toEpochMilli())
        );
    }
    if (endDate != null) {
        query.add(AuditEntity.revisionProperty("timestamp").le(
                endDate.toEpochMilli())
        );
    }

    List<Object[]> results = query.getResultList();

    var gradeChanges = results.stream()
            .map(result -> {
                AssessmentGradeEntity entity =
                    (AssessmentGradeEntity) result[0];
                CustomRevisionEntity revision =
                    (CustomRevisionEntity) result[1];

                return new GradeChangeDTO(
                        lectureAssessmentId,
                        entity.getGrade(),
                        revision.getTimestamp()
                );
            })
            .toList();
    return new GradeHistoryResponse(gradeChanges);
}
```

Listing 6.6: Reconstructing historic state using Envers, simplified code example adapted from `impl-crud/src/main/java/karsch.lukas.stats.StatsService`

6.5.5 Tracing Request Flow

Figure 6.2 presents the flow of two requests through the CRUD implementation. The first request is a `POST` request, meaning it writes something. The client sends their request to the web controller, which calls a service using the request body and, optionally, additional information about the requesting user. The service uses its own or external service methods to validate the request. If validation passes, the service creates a JPA entity and persists it to its JPA repository. Afterward, the service returns the `UUID` of the created entity to the controller, which finally responds to the client by wrapping the service result in an HTTP response.

As reads pass through the same components as writes, a subsequent read request is also present in the diagram. The client sends a `GET` request, which is received by the controller that then calls a service method to fetch the requested data. The service may apply additional logic, e.g. filters. Then, it selects the data from the repository, maps the result to a DTO and returns it to the controller. The controller sends an HTTP response with status code 200, containing the response body.

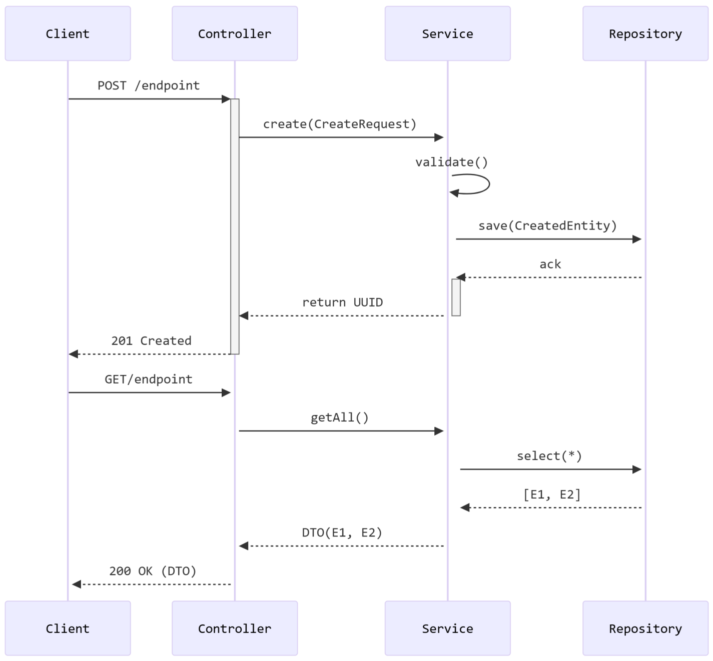

Figure 6.2: Reads and writes in CRUD / Layered Architecture. Data Store omitted

6.6 ES/CQRS implementation

6.6.1 Architectural Overview

The architecture of the `impl-es-cqrs` application [21] 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 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.

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

6.6.2 Data Modeling

TODO

6.6.3 Request Handling

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` [22]. 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`,[23] which is injected into the aggregate's command handler. The validator employs set-based validation as described in section 6.2.6. Once the prerequisite courses are validated, a `CourseCreatedEvent` is emitted. Otherwise, a specific `MissingCoursesException` is thrown, indicating that command handling was rejected.

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

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

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

this event is received, the saga sends an `AwardCreditsCommand` to the `Enrollment-Aggregate`. 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`).

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 simple. 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.

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.

6.6.4 Encapsulation and API Boundaries

Like the CRUD application, this prototype also implements Controller interfaces defined by the shared `api` module. However, each feature slice contains its own `api` package that is shared between web controllers, command side and read side. This "internal" API maps the application's public interface to CQRS / ES internals by defining specific Command and Query classes which target Aggregates and projections. These feature-specific `api` packages are the only public packages in a feature slice, meaning communication across package boundaries is only possible using the defined Commands and Queries.

The public API of the application is exposed through its Controllers, which only interact with the `CommandGateway` and `QueryGateway`. This ensures that all interactions with the system internals go through the proper channels and that underlying implementations can be changed without affecting the clients.

6.6.5 Auditing and Temporal Queries

As described in section 2.7, no additional audit log has to be implemented when using Event Sourcing. The application's state can be reconstructed using the Event

Stream by rehydrating Aggregates (Command-side) and by replaying projections (Read-side), if desired.

Temporal queries are implemented differently than in the CRUD prototype. Using Envers, it is possible to select a specific range of revisions based on indexed columns, for example a "date" column. When using Event Sourcing, all events have to be replayed from the Event Stream. Axon offers a method to read all events emitted by one Aggregate. The Event Stream returned from this function then has to be filtered to match the relevant events. These events can be applied to temporary projections or collected into a list. Listing 6.7 presents how this workflow is used to query the grade history for a student.

```
@QueryHandler
public GradeHistoryResponse getGradeHistory(
    GetGradeHistoryQuery query
) {
    final UUID enrollmentId = getEnrollmentIdFromQuery(query);

    final List<GradeChangeDTO> gradeChanges = eventStore
            .readEvents(enrollmentId.toString())
            .filter(msg -> eventTypeMatches(msg, query))
            .filter(msg -> eventIdMatches(msg, query))
            .filter(msg -> matchesDateFilter(msg, query))
            .asStream() // turn into Java Stream
            .map(msg -> {
                LocalDateTime changedAt = getEventTimestamp(msg);
                GradeAssignedEvent payload = msg.getPayload();
                return new GradeChangeDTO(payload.assessmentId(),
                    payload.grade(), changedAt);
            })
            .toList();

    return new GradeHistoryResponse(
            query.studentId(),
            query.lectureAssessmentId(),
            gradeChanges
    );
}
```

Listing 6.7: Simplified code for a Temporal Query in the ES-CQRS implementation. Adapted from `impl-es-cqrs/src/main/java/karsch.lukas.features.stats.queries.gradeHistory.GradeHistoryProjector`

6.6.6 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.

Because reads and writes take different paths through the application, two separate diagrams are presented.

Command Request Flow

Figure 6.3 illustrates the flow of a command through the system. Upon receiving a request, the controller constructs a specific `Command` object containing the request data and dispatches it through the `CommandGateway`. This gateway is responsible for routing the command to the appropriate destination, typically an `Aggregate` constructor or another method annotated with `@CommandHandler`. The command handler verifies that the command is allowed to be executed by performing validation logic and business rule checks. If the validation is successful, the aggregate triggers a state change by applying a corresponding `Event` 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 aggregate's `@EventSourcingHandler` is executed, updating the aggregate's internal state. It is worth noting that only the fields necessary for identifying the aggregate or maintaining its consistency are typically stored in the aggregate state, while other properties may be ignored on the command side. Any read-side projectors with `@EventHandlers` for the event are also executed, usually asynchronously, after the event is applied to update the projection databases.

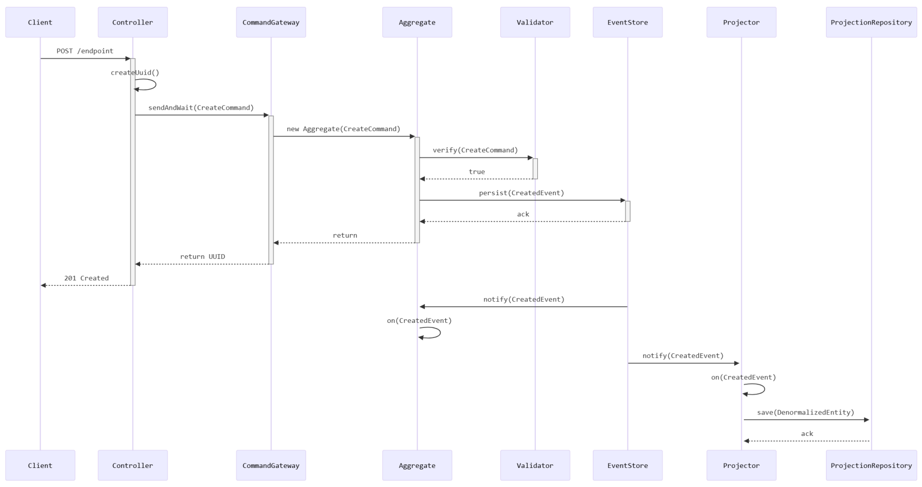

Figure 6.3: Sequence Diagram: Command Flow inside the ES-CQRS application

Query Request Flow

Figure 6.4 illustrates the flow of a query through the application. The request is received by a REST controller, which creates a `Query` instance and sends it to Axon's `QueryGateway`. The gateway routes the query to the appropriate `@QueryHandler` method responsible for that specific query type. The query handler accesses its respective projection repository to fetch the required data, maps the entities to DTOs, and returns the result. The `QueryGateway` hands this result back to the controller, which returns the data to the client.

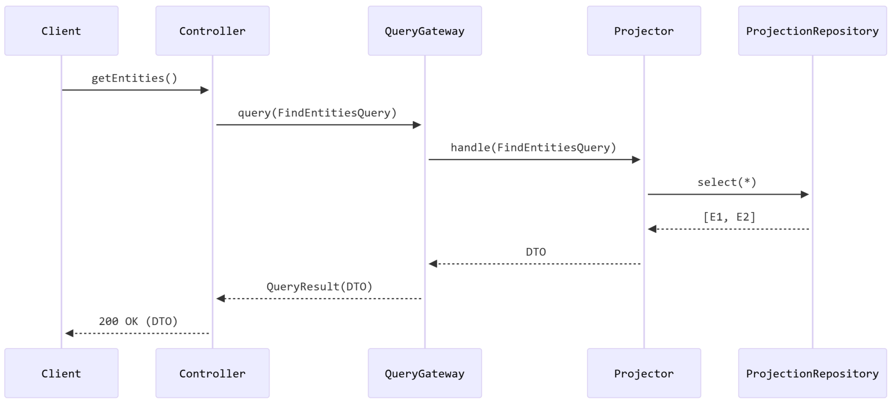

Figure 6.4: Sequence Diagram: Query Flow inside the ES-CQRS application

6.7 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 Virtual Machine (VM) provisioning setup for performance testing, as well as an integration testing strategy using Testcontainers, described in section 6.3.

6.7.1 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 `Dockerfiles`. 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.

6.7.2 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`,[24] 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 [25] configuration file that handles the provisioning of the VM upon its first boot. [26]

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.

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 6.2.

The physical host provisions two 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

[24] `performance-tests/vm/scripts/create-vm.sh`
[25] `https://cloudinit.readthedocs.io/en/latest/`
[26] `performance-tests/vm/scripts/cloud-init.yml`

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 6.2: Hardware specifications for the performance evaluation machine

negligible, the "queueing delay" remains measurable at the client level, allowing for the identification of request queues building up on the server, indicating bottlenecks.

6.8 Load Tests

This section describes the implementation of load tests.

6.8.1 k6 Scripts

The core of the load testing suite are the load-generating scripts developed using k6. Listing 6.8.1 illustrates the implementation of a typical k6 script using the creation of courses with prerequisites as an example. [27]

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 Requests Per Second (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 section 4.3.1. 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.

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

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

6.8.2 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` [28] and `many_runs.py` [29]. 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 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

[28] `performance-tests/perf_runner.py`

[29] `performance-tests/many_runs.py`

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`.

6.8.3 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_p*95, measured in seconds. Precisely, k6's *med*, *avg* and percentile latency metrics are mapped to the Prometheus equivalent, laid out in Table 4.1. Performing this normalization step immediately after the test run means the collected data can easily be compared and visualized later.

6.8.4 Testing "Freshness": Time to Consistency

To assess the eventual consistency of the ES-CQRS architecture, a specialized test for the Freshness SLO was developed.[30] 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 HTTP 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

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

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.

Chapter 7

Results (TODO)

7.1 Performance

This section describes results of load testing.

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

Chapter 8

Discussion (TODO)

8.1 Analysis of results

8.2 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] 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`

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

[4] 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/`

[5] 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`

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

[7] 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`

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

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

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

[11] 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

[12] E. A. Brewer, "Towards robust distributed systems (abstract)," en, in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00, New York, NY, USA: Association for Computing Machinery, Jul. 2000, p. 7, ISBN: 978-1-58113-183-3. DOI: 10.1145/343477.343502 Accessed: Dec. 28, 2025. [Online]. Available: https://doi.org/10.1145/343477.343502

[13] 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

[14] 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

[15] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, & B. B. Welch, "(PDF) Session guarantees for weakly consistent replicated data," en, 1994. DOI: 10.1109/PDIS.1994.331722 Accessed: Feb. 12, 2026. [Online]. Available: https://www.researchgate.net/publication/3561300_Session_guarantees_for_weakly_consistent_replicated_data

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

[17] 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`

[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] P. Maier, *Audit and Trace Log Management: Consolidation and Analysis*, en, 1st ed. 2006, ISBN: 978-0-8493-2725-4. Accessed: Nov. 14, 2025. [Online]. Available: `https://www.routledge.com/Audit-and-Trace-Log-Management-Consolidation-and-Analysis/Maier/p/book/9780849327254`

[23] 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`

[24] 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`

[25] 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

[26] 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

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

[28] P. Helland, "Immutability Changes Everything," en, Asilomar, California, USA., Jan. 2015.

[29] M. L. Abbott & M. T. Fisher, *The Art of Scalability. Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, eng. Addison-Wesley, Dec. 2009, ISBN: 978-0-13-703042-2.

[30] P. Jogalekar & M. Woodside, "Evaluating the Scalability of Distributed Systems," en, *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 6, pp. 589–603, Jun. 2000. Accessed: Feb. 2, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/862209

[31] S. Jayaraman & R. Mishra, "Implementing Command Query Responsibility Segregation (CQRS) in Large-Scale Systems," en, *International Journal of Research in Modern Engineering & Emerging Technology*, vol. 12, no. 12, pp. 49–73, Dec. 2024, ISSN: 23206586. DOI: 10.63345/ijrmeet.org.v12.i12.3 Accessed: Dec. 28, 2025. [Online]. Available: https://ijrmeet.org/implementing-command-query-responsibility-segregation-cqrs-in-large-scale-systems/

[32] D. Hruzin & O. Lytvynov, "ON THE MIGRATION OF DOMAIN DRIVEN DESIGN TO CQRS WITH EVENT SOURCING SOFTWARE ARCHITECTURE," *Information Technology Computer Science Software Engineering and Cyber Security*, vol. 1, pp. 50–60, Jun. 2024. DOI: 10.32782/IT/2024-1-7

[33] A. Singh, R. Bhatia, & A. Singhrova, "Object Oriented Coupling based Test Case Prioritization," *International Journal of Computer Sciences and Engineering*, vol. 6, pp. 747–754, Sep. 2018. DOI: 10.26438/ijcse/v6i9.747754

[34] V. R. Basili, L. Briand, & W. L. Melo, "A validation of object-oriented design metrics as quality indicators," en, *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, Oct. 1996, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/32.544352 Accessed: Feb. 11, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/544352/

[35] M. Overeem, M. Spoor, S. Jansen, & S. Brinkkemper, "An empirical characterization of event sourced systems and their schema evolution — Lessons from industry," en, *Journal of Systems and Software*, vol. 178, p. 110 970, Aug. 2021, ISSN: 01641212. DOI: 10.1016/j.jss.2021.110970 Accessed: Nov. 3, 2025. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121221000674

[36] V. Monagari, "Demystifying Event-Driven Microservices in Cloud-Native FinTech Applications," *Journal of Information Systems Engineering and Management*, vol. 11, no. 1s, pp. 184–197, Jan. 2026, ISSN: 2468-4376. DOI: 10.52783/jisem.v11i1s.14061 Accessed: Feb. 2, 2026. [Online]. Available: https://jisem-journal.com/index.php/journal/article/view/14061

[37] S. D. Gantz, *The Basics of IT Audit*, en. Elsevier, 2014, ISBN: 978-0-12-417159-6. Accessed: Nov. 13, 2025. [Online]. Available: https://www.oreilly.com/library/view/the-basics-of/9780124171596/

[38] E. Yourdon, L. L. Constantine, & L. L. Constantine, *Structured design: fundamentals of a discipline of computer program and systems design*, eng, 2. ed. New York, NY: Yourdon Pr, 1978, ISBN: 978-0-917072-11-6.

[39] R. C. Martin, *Agile software development: principles, patterns, and practices* (Alan Apt series), eng. Upper Saddle River, NJ: Prentice Hall/Pearson Education, 2003, ISBN: 978-0-13-597444-5.

[40] F. Brito e Abreu & R. Carapuça, "Object-Oriented Software Engineering: Measuring and Controlling the Development Process," en, *Pro-*

ceedings of "4th Int. Conf. on Software Quality", no. Revised Version, 1994.

[41] M. Richards & N. Ford, *Fundamentals of software architecture: an engineering approach*, eng, First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2020, ISBN: 978-1-4920-4345-4 978-1-4920-4340-9 978-1-4920-4342-3.

[42] M. Page-Jones, *What every programmer should know about object-oriented design*, eng. New York, NY: Dorset House Publ, 1995, ISBN: 978-0-932633-31-6.

[43] S. Chawla & G. Kaur, "Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development," *International Journal of Advanced Computer Science and Applications*, vol. 4, Oct. 2013. DOI: `10.14569/IJACSA.2013.040925`

[44] S. Chidamber & C. Kemerer, "A metrics suite for object oriented design," en, *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: `10.1109/32.295895` Accessed: Feb. 5, 2026. [Online]. Available: `https://ieeexplore.ieee.org/document/295895/`

[45] O. Drotbohm, *The Instability-Abstractness-Relationship — An Alternative View*, Sep. 2024. Accessed: Feb. 6, 2026. [Online]. Available: `https://odrotbohm.de/2024/09/the-instability-abstractness-relationsship-an-alternative-view/`

[46] K. Wiegers & C. Hokanson, *Software Requirements Essentials: Core Practices for Successful Business Analysis*, en. Addison Wesley, 2023, ISBN: 978-0-13-819028-6.

[47] 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.

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

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

[50] Sun Microsystems, *JavaBeans Specification*, 1997. Accessed: Feb. 9, 2026. [Online]. Available: https://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf?AuthParam=1770635266_3cf6e1c3e76abcca79c696cbebc8fd48#page=7.08

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

[52] 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

[53] 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

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

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

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

[57] 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

[58] 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

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

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

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

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

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

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

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

[66] 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/

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

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

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

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

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

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

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

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

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

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

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

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

[79] 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

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

[81] 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/

[82] 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/

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

[84] 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/`

[85] 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/`

[86] 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 applications, performance tests and markdown notes, is available at the following locations:

- `https://gitlab.mi.hdm-stuttgart.de/lk224/thesis`
- TODO: GitHub link

The repository contains `README` files with instructions on how to launch the applications, execute load tests and how to reproduce the VM environments.

Appendix B

Nutzung von KI-Tools

Kapitel / Codemodul	**Tool(s)**	**Beschreibung und Begründung Einsatzzweck**
Alle Kapitel	Gemini	LaTeX Syntax-Unterstützung: Skelette für Tabellen, figures, listings
Alle Kapitel	DeepL Write	Grammatikalische und stilistische Überarbeitung von Textsegmenten
Alle Kapitel	DeepL Translate	Übersetzung von Wörtern und Redewendungen
Kapitel 7 (Results)	Gemini	Generieren von Snippets zur Visualisierung der Ergebnisse (matplotlib, pandas, seaborn)

Table B.1: AI tools used throughout the thesis