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

Abstract

This is my abstract.
Keywords: CQRS, CRUD, Architecture, Load Testing, Scalability, Traceability, Static Analysis

Das ist mein Abstrakt. Es geht um einen Vergleich zwischen Softwarearchitekturen: typische Schichtenarchitektur vs. CQRS und Event Sourcing.
Keywords: CQRS, CRUD, Architektur, Stresstests, Skalierbarkeit, Rückverfolgbarkiet, Statische Analyse

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. 4, 29, 31, 37, 43, 47

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, 12

Contract Test A contract test verifies that services implement a shared interface by testing their interactions against an explicitly defined contract. 30

Docker Open platform for developing, shipping and running containerized applications. 36, 50, 51, 53

Dockerfile A text document containing a series of instructions used to assemble a Docker image. 36

HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 [2]. 4, 30, 36–38, 40, 44, 53, 54, 71

Layered Architecture divides applications into *horizontal layers*, with each layer performing a specific role. Typical layers include Presentation, Business, Persistence. v, vii, 5–7

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. 14, 23, 24, 75

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. 14, 23, 24, 75, 79

REST stands for *Representational State Transfer*. It is an architectural style for distributed hypermedia systems. 3, 4

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

Schema Evolution The process of managing how data structures, such as events or database tables, change over time. 3, 20, 25, 90

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. 14, 58

Test Configuration A test configuration is the combination of a test script and the target RPS.. 23

Testcontainer Testcontainers are a way to declare infrastructure dependencies as code using Docker [3]. 37, 50

Tuple An ordered, immutable sequence of elements used to store a collection of data.. 43

Acronyms

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

AHF Attribute Hiding Factor. 16, 81

AIF Attribute Inheritance Factor. 16, 81

BASE Basically Available, Soft State, Eventual Consistency. 8

CBO Coupling Between Objects. 17, 18, 20, 82, 87, 91

CF Coupling Factor. 16, 81

CK Chidamber and Kemerer Metrics. 17, 20, 56, 82, 87

CLF Clustering Factor. 17, 56

CQRS Command Query Responsibility Segregation. 1–3, 8–10, 12, 19, 20, 22, 27, 29, 30, 32, 33, 35, 47, 54, 75, 85–89, 92, 93

CQS Command And Query Separation. 8

CRUD Create Read Update Delete. 1–3, 7, 12, 21, 22, 26, 27, 29, 30, 38, 40, 41, 44, 47, 60, 75, 87–90

DAO Data Access Object. 5

DDD Domain Driven Design. vii, 3, 5–7, 10, 19, 32, 33

DIT Depth of Inheritance Tree. 17, 20, 82

DTO Data Transfer Object. 9, 37, 40, 44, 49

ES Event Sourcing. v, 1–3, 10–12, 19–22, 27, 29, 30, 32, 47, 75, 86–89, 92, 93

HTML HyperText Markup Language. 4, 5

I/O Input / Output. 14, 90, 93

IPS Iterations per Second. 71, 72, 86

IQR The "middle" of a dataset, representing the 50% of data between the 25th and the 75th percentile.. 60, 64, 65, 68, 70–72, 74–77, 79–81, 87

JMX Java Management Extensions. 36

JPA Jakarta Persistence API. 31, 32, 38–40, 43, 44, 87

JSON JavaScript Object Notation. 4, 5, 32, 46

L Load Test. 55, 56, 58, 85, 92

LCOM Lack of Cohesion in Methods. 17, 18, 20, 82

MHF Method Hiding Factor. 16, 81

MIF Method Inheritance Factor. 16, 81

MOOD Metrics for Object-Oriented Design. 16, 17, 56, 81, 87

NOC Number of Children. 17, 20, 82

ORM Object-relational Mapper. 31

PF Polymorphism factor. 16, 81, 82

POJO Plain Old Java Object. 32

RF Reuse Factor. 17, 56

RFC Response for a Class. 17, 20, 82

RPS Requests Per Second. v, 23, 24, 51, 52, 58, 60–75, 84, 86

RQ Research Question. 19, 23–25, 75, 90

SLA Service Level Agreement. 28

SLO Service Level Objective. 1, 15, 22, 28, 29, 54, 60, 90

SSH Secure Shell. 53

URI Uniform Resource Identifier. 4

VM Virtual Machine. 2, 50, 51, 53, 85, 89, 103

VU Virtual User. 37, 53, 59, 89

WMC Weighted Methods per Class. 17, 18, 82

WWW World Wide Web. 4

XML Extensible Markup Language. 5, 12

Chapter 1

Introduction

1.1 Motivation

Many enterprise applications are built as classic CRUD systems with a Layered Architecture, because this structure is widely used and straightforward to implement [4, Chapter 1]. Despite its simplicity, many requirements could arise that require a more careful selection of the architecture. The chosen architecture affects how fast the system stays under load, how easy an application is to change, and how well it can explain its past. This thesis therefore compares two different ways to build the same backend: a traditional CRUD system with an independent audit log, and an Event Sourcing system combined with CQRS.

A big factor in selecting the appropriate architecture is traceability. In many domains, applications must keep a verifiable, ordered history of actions for auditing. This requirement often emerges from legal requirements. In a CRUD system, this can be addressed by adding an audit log that records changes over time. Audit logs can be simple to write, but they are harder to read and process when you want to reconstruct complex historical states [5], and they can become fragile when the system has to write to both the main database and the log [6, pp. 452, 453].

Event Sourcing approaches the same problem from the opposite direction: instead of overwriting state, it stores every state change in an immutable, append-only event stream that becomes the system's *primary source of truth* [7]. This can preserve the intent behind changes through meaningful event types and enables temporal queries that reconstruct past states from the recorded history. When paired with CQRS, reads and writes are separated, which allows read models to be updated asynchronously.

Previous scientific work reports that CQRS (often together with Event Sourcing) can improve throughput and response times in certain scenarios. However, it can also increase infrastructure complexity and introduce new challenges such as eventual consistency.

These trade-offs motivate this thesis, which employs several quantitative methods to provide an accurate comparison of performance, architectural flexibility, and traceability under identical conditions and requirements.

To make the comparison concrete and fair, the thesis implements two prototypes that expose the same public API and follow the same functional contract: a course enrollment and grading system with deliberately complex relationships and validation rules. Both systems must be fully auditable, and the evaluation uses explicit SLOs to define meaningful breaking points. Finally, the work combines load testing for performance and scalability with static analysis metrics to discuss architectural flexibility and evolution.

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

This section defines the scope of the thesis by outlining the specific objectives to be achieved, as well as boundaries that will not be addressed.

1.3.1 Project Goals

The primary objective of this thesis is to develop two distinct prototype implementations that share an identical interface and follow the same functional contract. One application follows a traditional CRUD-based, layered architecture, while the other utilizes Command Query Responsibility Segregation (CQRS) and Event Sourcing (ES). Both systems are built using Spring Boot and PostgreSQL, with the ES/CQRS implementation additionally using the Axon Framework and Axon Server. A core goal is to implement both applications according to their respective industry best practices to ensure a fair comparison.

The comparison focuses on providing quantitative measurements regarding performance and scalability under increasing loads. Furthermore, the thesis evaluates architectural flexibility using established static analysis metrics and uses existing literature to form a reasoned opinion on the traceability of both systems. To ensure the results represent typical developer experiences, the performance comparisons rely on out-of-the-box Spring Boot configurations rather than custom-tuned environments. A major focus of the performance evaluation lies on the reproducibility and reliability of these results, which is achieved through reproducible, code-only Virtual Machine (VM) configurations, repeatable test scripts, and executing many test iterations to minimize statistical outliers.

1.3.2 Non-Goals

There are several areas that fall outside the scope of this research to keep the focus on the core architectural comparison. While the ES/CQRS application uses domain-driven principles, it is not a goal to achieve perfect Domain Driven Design (DDD) semantics. Similarly, the thesis does not aim to find the absolute maximum performance of either system through fine-tuning or specialized infrastructure configurations.

From a technical standpoint, the implementations do not include mechanics to support schema evolution or data migration over time. Security is also not a primary focus. While basic authorization is handled through a custom RequestContext, the applications do not follow full-grade production security best practices. Finally, the project aims to create a functional backend for testing purposes rather than a "real" usable application, which means no graphical user interface (GUI) is developed.

Furthermore, it is not a goal of this thesis to demonstrate real horizontal scaling or to run the applications on distributed servers, as the study focuses on architectural behavior within a controlled environment.

1.4 Structure of the thesis

This thesis is divided into eight chapters, going from the basics of REST APIs and server architectures to a concrete implementation, finally closing with a comparison of the two architectures and a conclusion to the given research questions.

After this initial introduction that defines the research goals and questions, chapter 2 (Basics) establishes the technical groundwork by explaining principles, Layered Architecture, DDD, and the mechanics of Event Sourcing and CQRS. chapter 3 reviews Related Work to situate the study within the existing knowledge regarding performance, flexibility, and traceability in distributed systems.

The practical work begins in chapter 4 (Methodology), where the study's three-phase research design is described. After the Requirement Analysis in chapter 5, chapter 6 describes the technical implementation of both the traditional CRUD-based system and the ES-CQRS prototype. Finally, chapter 7 presents the results of load tests and static analysis, which are then interpreted in chapter 8 (Discussion) to provide a holistic answer to the research questions and present opportunities for future work.

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 [8].

Web are interfaces that enable applications to communicate. They use HTTP as a network-based [9, p. 138]. Modern typically follow REST principles. REST stands for "Representational State Transfer" and describes an architectural style for distributed hypermedia systems [9, p. 76].

REST 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:

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

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.

In REST, Resources are decoupled from their representations. Clients can request different

representations of a resource, depending on their needs [9]. For example, a web browser might request HTML, while another server or application might request XML or JSON.

2.2 Layered Architecture

Richards [4] 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 [4, 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 [4, 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 [4, 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 [4, Chapter 1].

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

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

2.3 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.

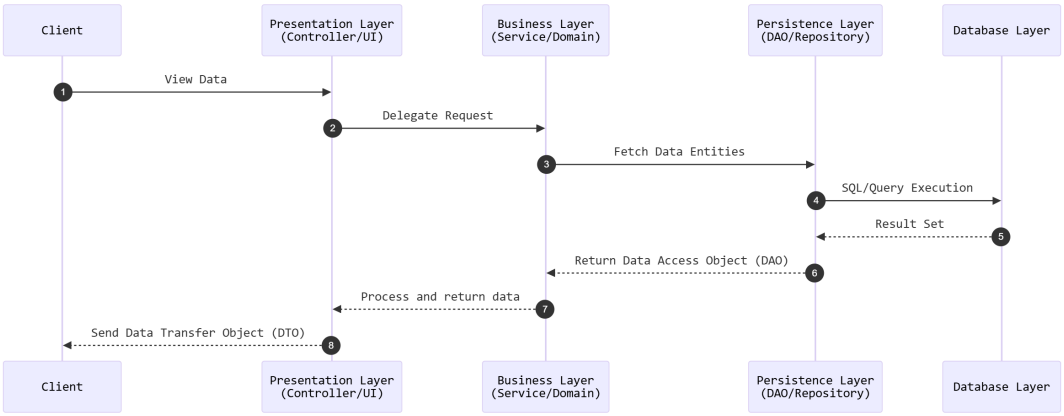


Figure 2.1: Layered Architecture sequence diagram, adapted from [4, p. 6].

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

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

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

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

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 [4, 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 [11, p. 80].

Using a Rich Domain Model does not mean that there should be no layers, the opposite is the case. Evans [11] advocates for using layers in domain driven designs. He proposes the layers presented in Table 2.3.

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

Value Objects are elements that describe the nature or state of something and have no conceptual identity of their own. They are interesting only for their characteristics. While two entities with the same characteristics are considered as different from each other, the system does not care about "identity" of a value object, since only its characteristics are relevant. Value objects should

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 [11, p. 53]

be used to encapsulate concepts, such as using an "Address" object instead of distinct "Street" and "City" attributes. Value objects should be immutable. They are never modified, instead they are replaced entirely when a new value is required [11, pp. 70–72].

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 [11, p. 89].

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

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

2.4 CRUD and ACID

Layered Architectures often follow a *Create Read Update Delete (CRUD)* architecture. CRUD is an acronym coined by Martin [12, 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 [12, 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 [12, p. 203].

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

2.5 Eventual Consistency

Gray et al. [13] 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 [14] 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 [15], [16]. This creates an inconsistency window in which data is not consistent across the system. During this window, stale data may be read [16]. This concept is visualized in Figure 2.2.

Consistency guarantees described by Terry et al. [17] 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 eventually achieves consistency through successful synchronization.

2.6 CQRS Architecture

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 [18, 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 [19, p. 17].

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

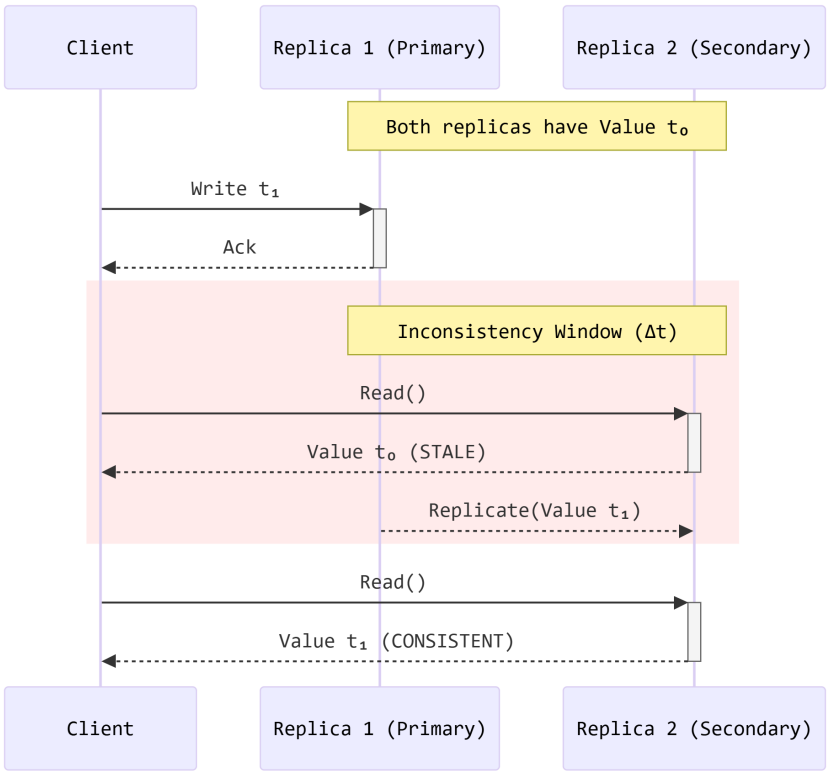


Figure 2.2: Sequence diagram illustrating the inconsistency window (Δt) in eventually consistent systems where read results depend on the selected replica. Based on [16]

- 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 [19, 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 [19, 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 [19, p. 23].

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

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 [20]. This

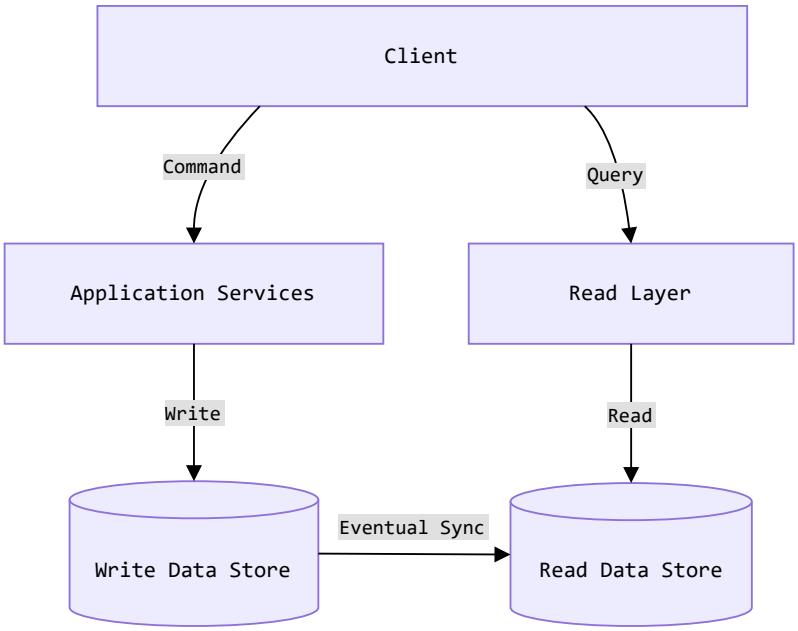


Figure 2.3: CQRS Architecture, adapted from [19, p. 24].

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 [21].

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 [6, 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 [21]. 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 [22], [6, pp. 461, 462]. This 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 [19, p. 50]. This architecture frequently applies DDD principles, where commands are processed by Aggregates. The combination of CQRS, ES, and DDD is central to Axon Framework, which serves as the technical foundation for this study's ES/CQRS prototype. Consequently, the following sections adopt Axon's semantics when describing the interaction between these patterns.

Figure 2.4 demonstrates the process of state reconstruction in an event-sourced CQRS system using 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.

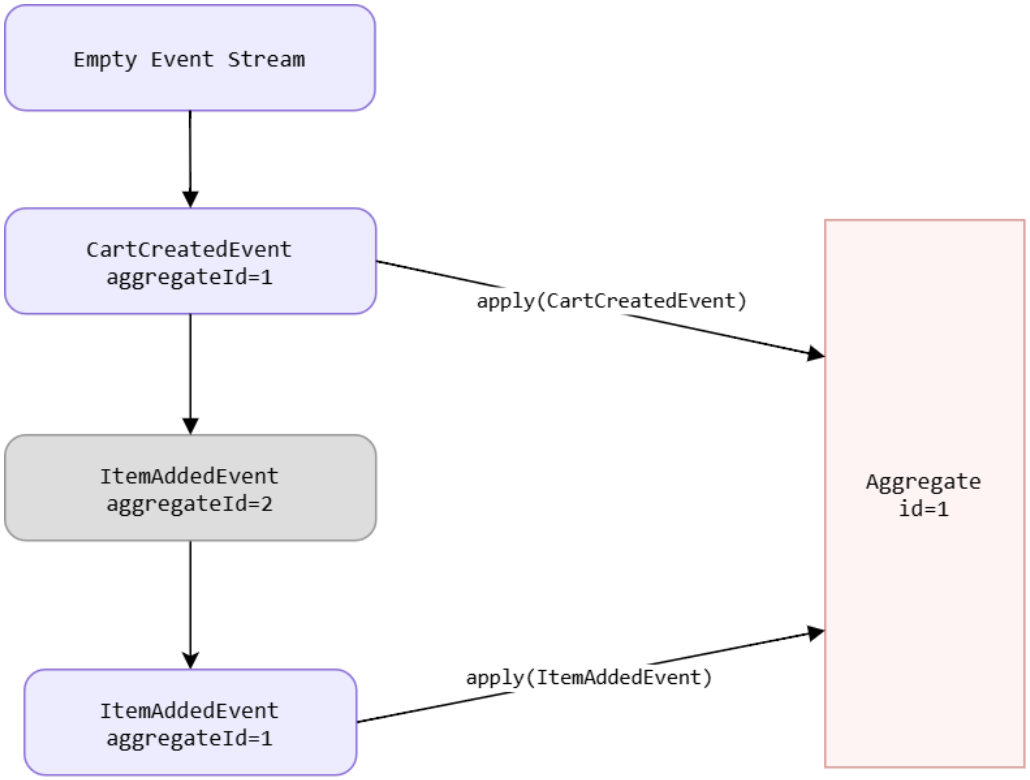


Figure 2.4: Applying events to rehydrate Aggregates in Event Sourcing.

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 [19, 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.

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 [23, 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 [24].

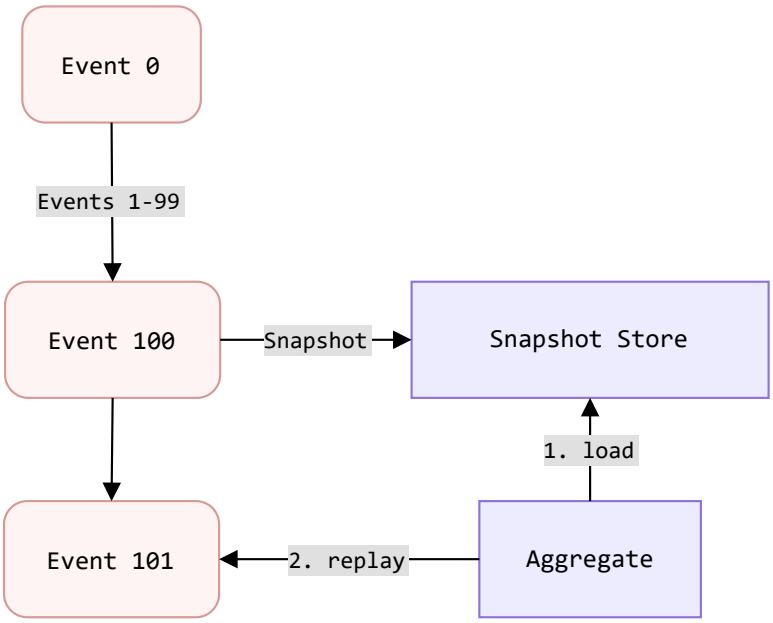


Figure 2.5: Snapshots in Event Sourcing / CQRS . Only the events between the most recent snapshot and the most recent event need to be applied to rehydrate the Aggregate [19, p. 20]

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 [25]. In information security, the audit log stores a record of system activities, enabling the reconstruction of events [26]. 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 [27, p. 266].

Fowler [5] describes an audit log as simple and effective way of storing temporal information. Changes are tracked by writing a record indicating *what* changed *when*. A basic implementation of an audit log can have many forms, for example a text file, database tables or XML documents. Fowler also mentions that while the audit log is easy to write, it is harder to read and process. While occasional reads can be done by eye, complex processing and reconstruction of historical state can be resource-intensive.

In distributed environments or complex application architectures, the implementation of an audit log often introduces the "dual-write" problem. This occurs when an application is responsible for updating the primary database (the current state) and simultaneously emitting a record to a separate audit log or messaging system. As Kleppmann [6, 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 [7].

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 [6, 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 [21] 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 [6, 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 [6, 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) [19]. If an event is not recorded, the state change never occurred [6, p. 460]. This guarantees that the log and the system state are consistent by design, ensuring traceability.

2.9 Evaluating Performance through Load Testing

2.9.1 Choosing Load Testing Scenarios

To accurately assess the performance of an architecture, a diverse suite of load tests should be employed. The selection of relevant endpoints to test should prioritize scenarios based on usage volume, business criticality, and resource intensity [28, pp. 260, 265].

When identifying relevant endpoints to load test, Abbott and Fisher advocate for a "Pareto Distribution" (80/20 rule) and argue 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 [28, pp. 260, 265].

2.9.2 Load Testing Theory

According to the methodology defined by Kleppmann [6, pp. 10–17], the load on a system is generally characterized using "load parameters," which vary depending on the system's nature. Kleppmann [6, 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, client-side latencies are more relevant when measuring user-perceived performance.

Furthermore, Kleppmann [6, p. 16] 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 a "closed model." Regarding increasing load, Kleppmann [6, p. 13] describes two perspectives: keeping system resources constant while measuring performance fluctuations under varying load, or increasing resources to maintain unchanged performance.

To evaluate results, the arithmetic mean is often avoided as it obscures the experience of typical users and the impact of outliers. Instead, Kleppmann [6, pp. 14–16] suggests using percentiles. The median (P50) serves as the metric for "typical" response time. To capture the experience of users facing high delays, it is critical to measure "tail latencies" via high percentiles like P95 or P99. These identify the performance of outliers which, despite being a numerical minority, often represent the most valuable or complex operations.

2.10 Scalability Dimensions

This section outlines several dimensions of scalability, ranging from a mathematical productivity metric to architectural design choices.

2.10.1 Quantifying Scalability through a Productivity Metric

Jogalekar & Woodside [29] 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.

Their proposed productivity metric is presented in Equation 2.1.

$$F(k) = \lambda(k) \cdot f(k)/C(k) \tag{2.1}$$

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. For example, they propose using the actual cost of hardware as cost function, e.g. when running on the cloud.

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

$$\psi(k_1, k_2) = \frac{F(k_2)}{F(k_1)} \tag{2.2}$$

When ψ is close to or greater than 1, the system is considered to be scalable. [29, pp. 4–6]

2.10.2 Headroom and Resource Saturation

Abbott & Fisher [28, 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.

2.10.3 Architecture That Scales

Scalability is influenced by the alignment between the technology and the organization that builds it. Abbott & Fisher [28, p. 2] argue 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.

Generally, there are two ways to scale a system: "Scaling out" (horizontal scaling) and "scaling 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 [28, 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 split horizontally is more resilient as it does not rely on high-performance 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 [28, 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 [28, p. 205]. Finally, stateless systems allow requests to be distributed freely across any available server, making them more scalable. Abbott & Fisher [28, p. 206] argue that stateful applications should be avoided, when possible.

2.11 Static Analysis 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.

2.11.1 Coupling Metrics

Coupling metrics are based on graph theory. They measure how dependent components are on each other. *Afferent coupling* (C_a) measures the number of incoming connections to a code artifact, indicating how many other components depend on it. *Efferent coupling* (C_e) 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. [30] in 1978.

2.11.2 Instability and Abstractness

Instability assesses the volatility of a code base by calculating the ratio of efferent coupling to the sum of all coupling (Equation 2.3), indicating how easily a component breaks when changed.

$$I = \frac{C_e}{C_e + C_a} \tag{2.3}$$

Abstractness determines the ratio of abstract artifacts, such as interfaces and abstract classes, to concrete implementations within a module.

$$A = \frac{AbstractClasses + Interfaces}{TotalClasses} \tag{2.4}$$

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" [31, pp. 261–268]. Martin considers a stable, concrete package to fall into the "Zone of Pain", while instable, highly abstract packages fall into the "Zone of Uselessness". The Main Sequence is illustrated in Figure 2.6.

2.11.3 MOOD Metrics

Metrics for Object-Oriented Design (MOOD), introduced by Brito e Abreu & Carapuça [32], 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

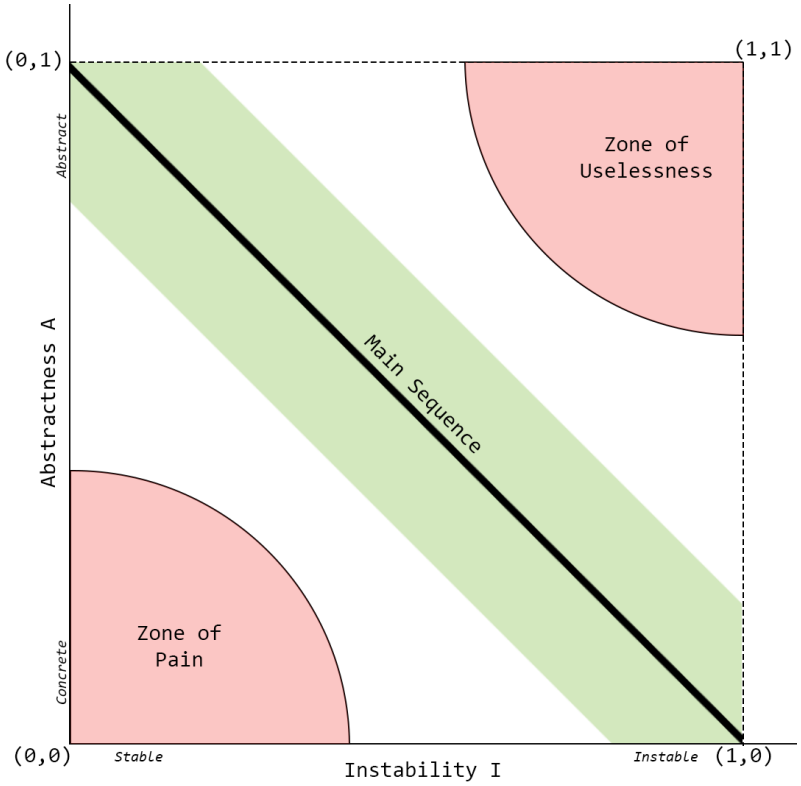


Figure 2.6: Main Sequence, Zones of Pain and Uselessness. Adapted from [31, p. 266]

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 sub-graphs 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.

2.11.4 Chidamber and Kemerer (CK) Metrics

Chidamber and Kemerer Metrics (CK) are the most well-known object-oriented suite of metrics, according to Chawla & Kaur [33, p. 162]. They are described in a work from 1994 by Chidamber & Kemerer [34]. *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. *Depth of Inheritance Tree (DIT)* measures the maximum length from a specific node to the root of the tree, where deeper trees imply higher complexity. *Number of Children (NOC)* counts the immediate subclasses of a class, indicating the potential influence on the design and level of reuse. *Coupling Between Objects (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. *Response for a Class (RFC)* counts the number of methods that can be executed in response to a message received by an object, reflecting the class's complexity. *Lack of Cohesion in Methods (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.

2.11.5 Usefulness of Static Analysis

The described metrics offer a quantitative method to identify 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 [33]. Additionally, cohesion metrics like *LCOM* and complexity measures such as *WMC* help to determine if an architecture is composed of focused, manageable components or large, complex classes that attempt to do too much [33].

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 [35, pp. 45–47].

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 chapter 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 [6] 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 [6, Chapter 1]. To quantify scalability characteristics, Jogalekar & Woodside [29] 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 [36] 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 [37] 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 appears to be 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 [36], [37], [38].

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 [37], 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, some of which were described in section 2.11. Singh et al. [39] employed statistical approaches, including hypothesis testing and linear regression, to correlate the CK suite with software quality. Their analysis established that metrics such as DIT and NOC impact quality, but CBO demonstrated the strongest negative correlation with software quality. Basili et al. [40] 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, RFC and NOC were most significant for predicting defects in a program, while LCOM was shown to be insignificant.

However, different critiques on these static analysis methods can be found in literature. Richards & Ford [35, 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 [35, p. 48]. Finally, the authors express their critique of coupling metrics, mentioning that they focus too much on the low-level details of software coupling. The authors argue that developers should care more about *how* components are coupled (e.g. synchronous or asynchronous) [35, p. 53].

Drotbohm [41], a Spring engineer, offers his critique on the *abstractness* metric, highlighting that it mistakes the existence of interfaces with actual, useful abstractions.

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 [6, pp. 461, 462].

While Kleppmann emphasizes the architectural freedom to evolve views on existing data, Overeem et al. [42] discuss 19 event-sourced systems and point out that many engineers struggle with *schema evolution* in Event Sourcing, which 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. While worth mentioning, this aspect will not be taken into consideration when providing a final comparison of the two architectures.

In terms of maintainability, while Jayaraman & Mishra [36] 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 [6, pp. 457, 531] mentions that using ES makes it easier to reproduce bugs and diagnose unexpected behaviors by replaying the event log. Monagari [38] 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 used in this study could not be verified during literature review.

Gantz [43] defines IT auditing as the process of validating controls to protect assets and information. Maier mentions that historically, audit logs were often viewed as "disposable" data, overwritten regularly to save space [23, p. 3]. However, he argues that regulatory pressure and the need for accountability have made log retention critical for event reconstruction and forensic analysis [23, pp. 4, 17].

Effective auditing requires "evidence": information that auditors can verify against established criteria [43, p. 155]. Gantz emphasizes that the reliability of a system depends on its ability to produce accurate evidence of past states and operations [43, p. 4].

The standard audit logging approach employed in traditional CRUD systems has several downsides which can be found in the literature. Kleppmann [6, 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 [23, 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 [7] 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 [6, p. 531] also highlights that events typically capture user intent.

In terms of compliance and integrity, Kleppmann [6, 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 attempts to quantify 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.

Load Testing

Based on the theoretical foundations laid out in section 2.9, this thesis performs load testing on both implementations. The primary load parameter used is the number of concurrent requests to the web server [6, p. 11]. The tests measure **Requests Per Second (RPS)**, and server- and client-side response times to ensure that queueing delays are accurately captured and user-perceived performance is reflected.

Following the "open model" principle, the load-generation setup ensures that requests are sent continuously to simulate realistic concurrency. This thesis adopts the approach of keeping system resources constant while measuring performance fluctuations under varying load intensities.

In the analysis of the results, the arithmetic mean is avoided in favor of percentiles. The median (P50) is used to determine the typical response time, while P95 and P99 percentiles are utilized to measure tail latency and identify the performance of outliers.

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 [28, 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 [28, 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 **??**, 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 mentioned primarily when visualizing and describing 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.

axon_event_size and *axon_snapshot_size* are metrics extracted from the event store (Axon Server). They are 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 for denormalized projections and lookup tables. 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_p50	50th percentile Latency (median)	Server, Client
latency_p95	95th percentile Latency	Server, Client
latency_p99	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_event_size	Size of Axon's Event store	Axon Server
axon_snapshot_size	Size of Axon's Snapshot store	Axon Server
axon_storage_size	$axon\_event\_size + axon\_storage\_size$	Axon Server
dropped_iterations_rate	Dropped iterations per second	Client
failure_rate	Rate of failed requests (in %)	Client

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.

Scalability

As outline above, load tests are used to collect latencies and system metrics. From these results, section 2.10 serves as a basis to define a scalability function specific to this thesis. Results from the scalability function can then be used to assess the scalability of each application.

4.3.2 Flexibility

section 2.11 described a set of static analysis metrics. These serve as a basis to answer RQ 2, using the results to compare the applications' architectures and their potential for flexibility and evolution over time.

4.3.3 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.

Another interesting perspective in this scope would be *schema resilience*, which 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 schema modifications. However, as schema evolution is not a goal of the study, this perspective will not be pursued further.

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

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. These challenges, which are bypassed by this study's setup, are known as the eight fallacies of distributed systems [44], a term first coined by P. Deutsch in 1994.

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 data store, creating a "dual-write" problem [6, Chapter 11]. 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 [28, 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, which may obscure long-term operational complexities. While data volume is simulated, the actual challenges of managing a growing event store, specifically the evolution of the schema over time, are evaluated theoretically rather than through years of operational data. As Young [45] notes, the permanence of events in an immutable log necessitates complex strategies like upcasting or transformation to handle changing business requirements. Without observing these "versioning" cycles over a multi-year lifecycle, an incorrect view of the maintainability of event-sourced architectures may be presented.

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.

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 and System Constraints

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.

Based on the domain described in subsection 5.1.1, the following list presents a selection of constraints and rules which are central to the system. As this thesis focuses on an architectural comparison, not every functional requirement going into the application is listed explicitly.

- Referential Integrity: The system shall verify the existence of all referenced entities during request handling. Requests involving non-existent entities shall be rejected.
- The system shall prevent conflicts such as time slot overlaps.
- 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. For example, enrolling shall only be possible during a lifecycle of "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:

SLO 1 **Latency SLO**: All endpoints shall maintain a client-side P95 latency of $\leq$100ms to ensure the system feels "instant" for 95% of requests.

SLO 2 **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.

SLO 3 **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 executed on both applications are outlined. Next, implementation details of CRUD and ES-CQRS are given. Finally, the load tests are presented.

6.1 Endpoints

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, only the endpoints which will later be load-tested are listed.

Endpoint	Description	Response
`GET /lectures`	Returns all lectures a student is enrolled or wait-listed in	200
`GET /lectures/all`	Returns details for all lectures	200
`POST /courses`	Creates a course	201
`POST /lectures/{id}/enroll`	Enroll to a lecture	201
`GET /stats/credits`	Returns a student's current credits	201
`GET /stats/grades/history`	Returns the grade history for an assessment	201

Table 6.1: Endpoints which will be load-tested

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 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 mapper between the relational world of SQL databases and the object-oriented world of Java [56, Chapter 1].

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. Using an

[1] `https://spring.io/projects/spring-boot`
[2] `https://spring.io/projects/spring-framework`
[3] `https://www.postgresql.org/`
[4] `https://jakarta.ee/specifications/persistence/`

`EntityManager`, these entities can be persisted and queried [56, Chapter 3].

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

Envers[6] is a Hibernate project designed for audit logging and versioning of historic data. When using Envers, a copy of data (a "revision") is stored in specific revision tables managed by Envers [56, Chapter 13.3].

Envers offers an API, primarily accessed through the `AuditReader`, to query historic versions of data [56, Chapter 13.3].

To use Hibernate Envers, the `org.hibernate.orm:hibernate-envers` dependency must be added to the project's classpath.

6.2.5 Jackson

Jackson[7] 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].

6.2.6 Axon

Axon Framework[8] 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 [59].

Axon also supports Domain Driven Design by providing tools to manage entities and domain logic [59], [60].

Axon Server[9] 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].

[5] `https://hibernate.org/orm/`
[6] `https://hibernate.org/orm/envers/`
[7] `https://github.com/FasterXML/jackson`
[8] `https://www.axoniq.io/framework`
[9] `https://www.axoniq.io/server`

This subsection describes core concepts of Axon. Diagrams illustrating the data flow through an application built with Axon are available in subsection 6.7.6.

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` [63].

`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 [63].

In general, command handling functions return `null` if handling was successful, except for command handlers which *create* Aggregates. These return the Aggregate identifier. 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 [63], [64].

Query Handling

Axon dispatches Queries 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`. When no query handler is found, an exception is thrown [65].

The `QueryGateway` includes different dispatching methods. For regular "point-to-point" queries, the `query` method can be used. For large result sets, streaming queries should be used. All query methods are asynchronous by nature and return Java's `CompletableFuture` [65].

Subscription Queries provide an initial result and continuous updates as data changes. These queries work well with reactive programming. By using a subscription query inside a web controller, a synchronous interface can be provided to clients, while keeping the internal command and event handling process asynchronous. While this approach does provide the desired synchronous user experience, it has the downside of coupling the client to the event processing flow. Alternatively, developers might employ WebSockets or other client-side notification mechanisms to inform the user about the result of their action asynchronously, after the initial request was accepted [65].

Aggregates

Aggregates are a core concept of DDD, which was described in section 2.3. They define command handlers using the `@CommandHandler` annotation. 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 [66].

Every Aggregate is 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 [66].

By default, Axon uses Event-sourced Aggregates. This means that Aggregates are not persisted as a simple 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 [66].

Axon also supports multi-entity Aggregates, where 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 [67].

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 [68].

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 [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].

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 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 an optimized and consistent dataset, such as a registry of unique IDs, which can be queried during the validation phase of a command [73].

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 [70], [71].

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 helps to enforce validation across individual aggregates and allows for global state verification [73].

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 [74].

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 [74]. 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 [75].

6.2.7 SpringBoot Actuator

Spring Boot Actuator[10] 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

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

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 [76], [77].

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 [78].

Actuator can be enabled in a Spring Boot project by including the `spring-boot-starter-actuator` dependency [79].

6.2.8 Prometheus

Prometheus[11] 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 [80], [81].

To collect and export Actuator metrics specifically for Prometheus, the `micrometer-registry-prometheus` dependency must be included in the classpath [82]. 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 [77].

6.2.9 Docker

Docker[12] 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 [83].

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 [84].

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

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

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 [85].

6.2.10 k6

Grafana k6[13] 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 [86].

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 [87].

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`[14], according to subsection 5.2.5. The test classes use the `JUnit 5`[15] testing framework and `REST Assured`[16] to send and assert HTTP requests. The test classes are abstract and must be extended by both application to add implementation-specific functionality, such as teardown and seeding functions.

Necessary infrastructure for these tests is spun up by the subclasses using Testcontainers[17]. 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`[18] module serves as base module for both implementations. This module defines DTOs, which 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 logic, for example Actuator endpoints to control the application's internal clock.[19]

[13] `https://grafana.com/docs/k6/latest/`
[14] `test-suite/src/test/java/karsch.lukas`
[15] `https://docs.junit.org/5.14.3/overview.html`
[16] `https://rest-assured.io/`
[17] `https://testcontainers.com/`
[18] `api/src/main/java/karsch.lukas`
[19] `api/src/main/java/karsch.lukas.time.DateTimeActuatorEndpoints`

6.5 Authentication and Authorization

Instead of implementing real authentication using a technology like Spring Security[20], a simple header-based mechanism was used to simulate an authenticated state. This approach relies on a custom HTTP header to attach session information to requests.

This mechanism consists of two main components: a request-scoped `RequestContext`[21] bean that acts as a container for user data throughout the lifecycle of a single HTTP request; and a custom `UserFilter`[22] that intercepts incoming requests, parses the header, and populates the context.

The filter expects a header named `customAuth` with the format `<type>_<uuid>`. The `type` and `uuid` are extracted from the header and set in the `RequestContext`.

This solution removes the additional implementation complexity of cryptographically secure authentication while allowing for user-dependent logic, such as creating courses and enrolling.

6.6 CRUD implementation

This section presents the relevant aspects of the CRUD implementation,[23] mainly focusing on relational modeling using JPA and the audit log implementation.

6.6.1 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.

6.6.2 Data Modeling

All data in the CRUD implementation is present in the form of database / JPA entities. Other objects like services and controllers are *stateless*. The application uses a normalized database in the Third Normal Form.

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. It should be noted that the auditing table created by Hibernate Envers are not present in the diagram. The diagram highlights that `LectureEntity` is the core entity in the system, having direct a relationship to many other entities.

All entities and value objects 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.

[20] `https://spring.io/projects/spring-security`
[21] `api/src/main/java/karsch.lukas.context.RequestContext`
[22] `api/src/main/java/karsch.lukas.context.UserFilter`
[23] `impl-crud/src/main/java/karsch.lukas`

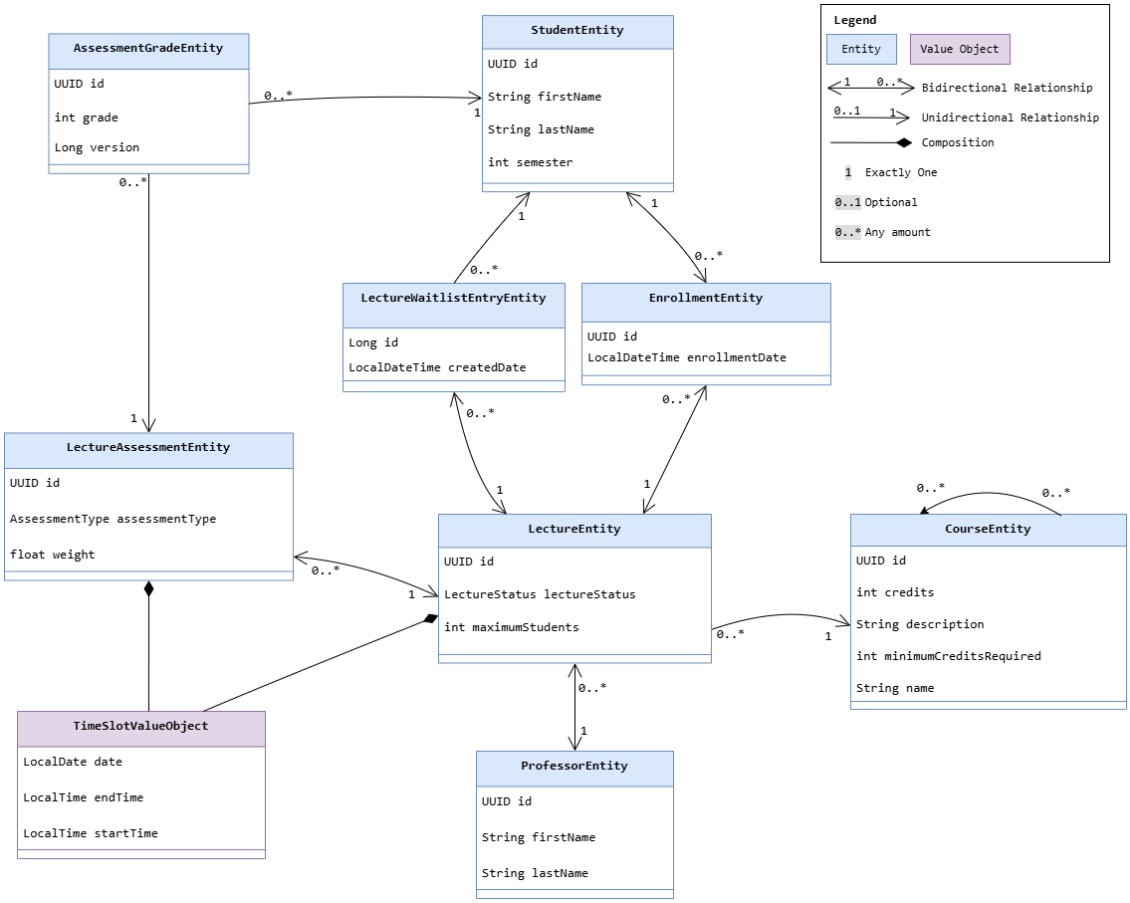


Figure 6.1: Entity Relationship Diagram for the CRUD App

```
@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[24] or Liquibase[25] 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.

[24] https://github.com/flyway/flyway
[25] https://www.liquibase.com/

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.

6.6.3 Request Handling

The request handling process in the CRUD application follows the flow described in section 2.2. A request reaches one of the web controllers (classes annotated with `@Controller`). The controller delegates the request to a service object. The service applies business logic to create, update or retrieve entities from the database using one or several repositories. Finally, the service returns a DTO, which the controller wraps in a HTTP response that is sent back to the client.

6.6.4 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 packages, 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.

6.6.5 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. Adapted from [5]

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 [88, 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 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 [89].
5. **Change Data Capture (CDC)** is the process of extracting all changes to a data store into a form that can be replicated to other systems. For example, a stream of changes can continuously be applied to a search index [6, Chapter 11].

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`[26] 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.

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

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

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

```
)
```

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`.

```
@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 6.6.5.

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.

```
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`[27] 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 section 6.5.

Reconstructing Historic State

Envers stores its revision data and historical records in special auditing tables. These tables should contain all necessary information to reconstruct historic state. Envers provides a specific API which can be queried to reconstruct historical state. This API was used to implement the "reconstruction" use-case in the application which allows users to query their grade history for a specific assessment. The implementation of this service method is outlined 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. Beyond filtering revisions by ID, Envers enables developers to add additional matchers based on revision properties. Here, the revision property `timestamp` is used to define the relevant date bounds.

Once the query is built, the result list can be fetched. The result is a list containing arrays of objects. More precisely, each list entry is a *Tuple*. The first value is the historic entity, and the second value is the revision entity created for this specific revision. Because a custom revision entity is registered, the type of this revision entity is `CustomRevisionEntity`.

```
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
```

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

```
    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`

6.6.6 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.

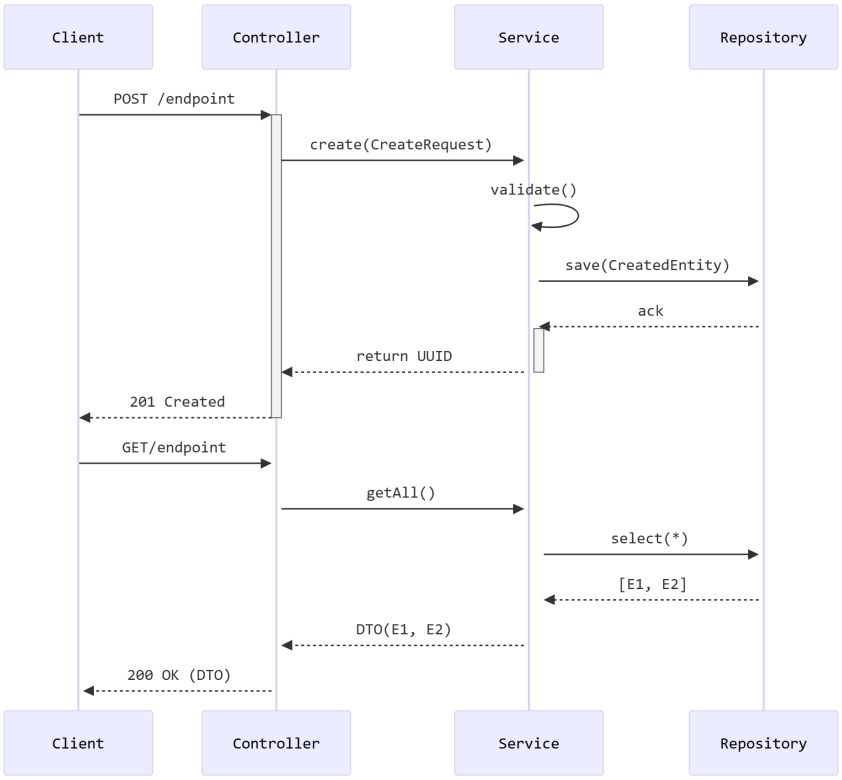


Figure 6.2: Reads and writes in CRUD / Layered Architecture. Data Store omitted

6.7 ES-CQRS implementation TODO: refine language

6.7.1 Architectural Overview

The architecture of the `impl-es-cqrs` application[28] 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.

An architecture like this 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. It should be mentioned that in many "vertical slice" architectures, the separation goes even further than described here, creating separate packages for each use-case, such as `GetLecture`, `CreateCourse`, and so on.

6.7.2 Data Modeling

In CRUD, *Data Modeling* refers to relational modeling. All data is present in the form of database tables which are written to and read from by all the components. In the ES-CQRS application, data is separated into writes and reads. The "data model" in ES-CQRS consists not of tables and rows, but of event types and an event stream, such as `CourseCreatedEvent`.

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

Data flow between components is decoupled through the use of Events, Commands and Queries. These message types facilitate all communication in the application.

The ES-CQRS application's Aggregates, which are central to Command handling in CQRS, are `CourseAggregate`[29], `LectureAggregate`[30], `EnrollmentAggregate`[31], `StudentAggregate`[32] and `ProfessorAggregate`[33]. Unlike CRUD's ER model where `LectureEntity` has a foreign key to a professor, the `LectureAggregate` stores only a `professorId`. This ensures that each aggregate can be loaded and versioned independently in the Event Store.

Sagas act as process managers, but they are also data. For each process requiring a Saga, a new instance of that Saga is created. The Sagas are long-running and serializable; they store all data that is relevant to their business logic.

Events, which carry facts about what happened in the system, are emitted by the command side and can be received and processed by the read-side, which then builds its projections. Projections can be seen as "derived data" which is built from events. Each Query use-case can and should create its own projections which contain exactly the data that is relevant to answer the respective Query.

The projections store data relevant to their responsibilities. To further improve efficiency, these projections are denormalized. Relationships are not mapped to database tables using foreign keys. Instead, the relevant relationships are directly stored in a JSON column on the projection table.

An example of a projector that stores data in such a way is the `LectureProjector`[34]. It demonstrates the fact that each projector maintains its own view of the system. Projectors must not use 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 single `SELECT` statement. The same concepts apply to all other projectors in the ES-CQRS implementation.

6.7.3 Request Handling

This section describes how different components of the ES-CQRS application handle requests. Diagrams illustrating the flow of both commands and queries are available in subsection 6.7.6.

Command Side

Command (or write) requests are received by a SpringBoot controller. The controller transforms the request into a Command message which is dispatched to the Command side using the `Command-`

[29] `impl-es-cqrs/src/main/java/karsch.lukas.features.course.commands.CourseAggregate`
[30] `impl-es-cqrs/src/main/java/karsch.lukas.features.lectures.commands.LectureAggregate`
[31] `impl-es-cqrs/src/main/java/karsch.lukas.features.enrollment.commands.Enrollment-Aggregate`
[32] `impl-es-cqrs/src/main/java/karsch.lukas.features.student.commands.StudentAggregate`
[33] `impl-es-cqrs/src/main/java/karsch.lukas.features.professor.commands.ProfessorAggregate`
[34] `impl-es-cqrs/src/main/java/karsch.lukas.features.lectures.queries.LectureProjector`

`Gateway`. The Command side is responsible for handling all state changes in the application. It is implemented using the Aggregates and Sagas described in subsection 6.7.2.

Most command handling occurs inside the Aggregates, however the enrollment of students is a use-case spanning across two Aggregates. This required the external Command handler `EnrollmentCommandHandler` to be implemented.

`AwardCreditsSaga` is a Saga which was manages the process of awarding credits to students. The Saga is initiated when an `EnrollmentCreatedEvent` occurs. It then waits for a `Lecture-LifecycleAdvancedEvent` with the status `FINISHED`. Once this event is received, the saga sends an `AwardCreditsCommand` to the `EnrollmentAggregate`. The saga ends when it receives a `Credits-AwardedEvent`. This ensures that credits are only awarded after a lecture is finished, and all assessments have been graded. Unlike the CRUD application, which calculates awarded credits based on the current state of a lecture, the ES-CQRS implementation makes the fact that credits are awarded explicit through an event. Even when changing the logic of 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

Read requests (Queries) are also received by a SpringBoot controller, which creates a Query object and dispatches it to the `QueryGateway`. The queries are routed to query handlers which can efficiently fetch data from their dedicated projections, usually without `JOINs`. 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.

6.7.4 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.

6.7.5 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`

6.7.6 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 — the message may be handled inside the same JVM or on a different machine [59].

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 `@Event-SourcingHandler` 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.

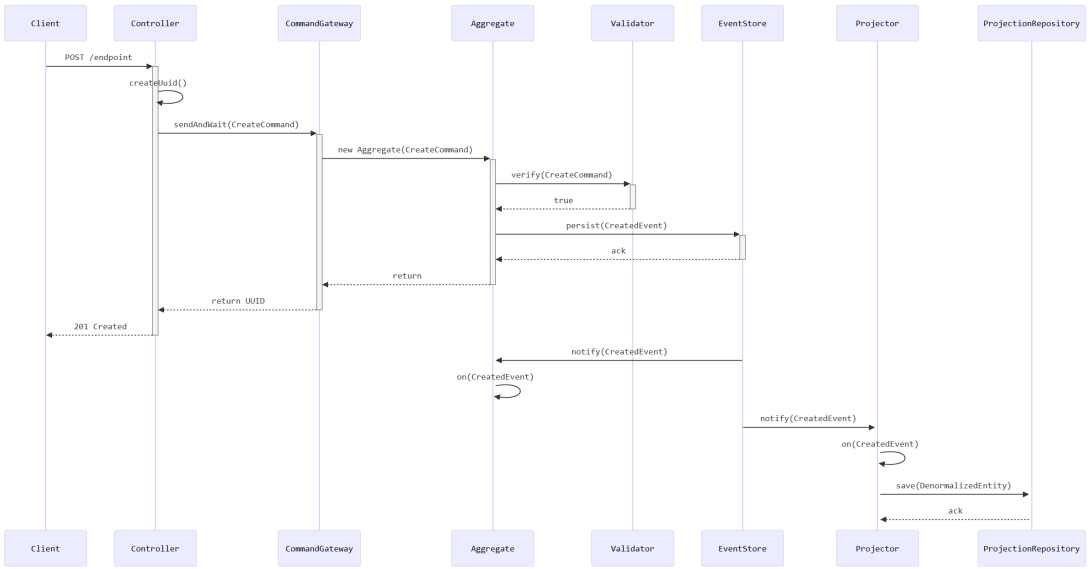


Figure 6.3: Sequence Diagram: Command Flow inside the ES-CQRS application

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.

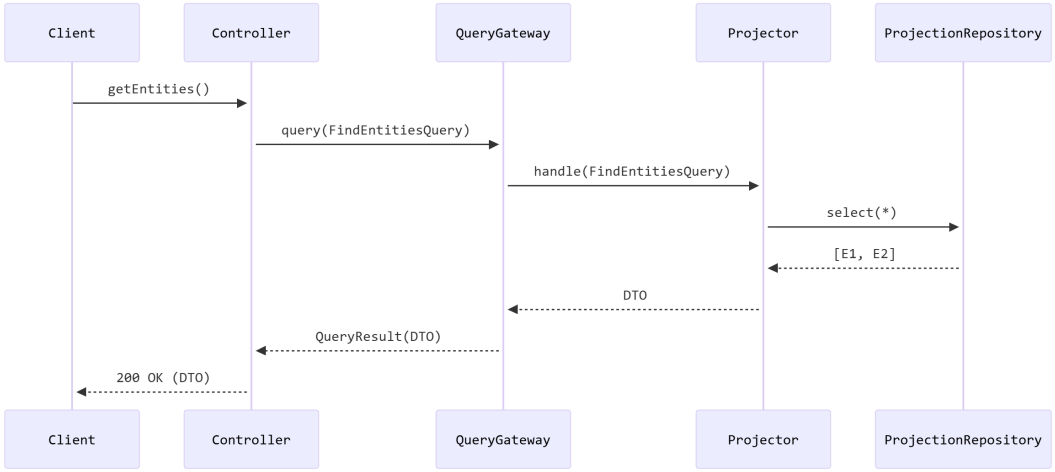


Figure 6.4: Sequence Diagram: Query Flow inside the ES-CQRS application

6.8 Infrastructure

The project's infrastructure is designed for consistency and reproducibility across development and testing environments. It is composed of a containerized environment for running the applications and their dependencies, an automated VM provisioning setup for performance testing, as well as an integration testing strategy using Testcontainers, described in section 6.3.

6.8.1 Containerized Services

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

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

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

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

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

6.8.2 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`,[35] 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[36] configuration file that handles the provisioning of the VM upon its first boot.[37]

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.

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

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

6.9 Performance Evaluation through Load Tests

This section describes the implementation of load tests.

6.9.1 k6 Scripts

The core of the load testing suite are the load-generating scripts developed using k6. Listing 6.9.1 illustrates the implementation of a typical k6 script using the creation of courses with prerequisites as an example.[38]

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

[35] `performance-tests/vm/scripts/create-vm.sh`
[36] `https://cloudinit.readthedocs.io/en/latest/`
[37] `performance-tests/vm/scripts/cloud-init.yml`
[38] `performance-tests/k6/writes/create-course-prerequisites/create-course-prerequisites.js`

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 ??. Stages in a scenario define the "timeline" of RPS. In the given example, RPS are increased from 0 to the target RPS over a duration of 20 seconds. This RPS is then held for a duration of 80 seconds, before decreasing RPS back to 0 over a span of 20 seconds.

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

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

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

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

6.9.2 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`[39] and `many_runs.py`[40]. 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 `VU` environment variable defines the arrival rate for the `ramping-arrival-rate` executor. Unlike a fixed concurrency model, this approach ensures a consistent load by triggering a specific number of iterations per second. By decoupling the request rate from the response time of individual HTTP calls, the script maintains consistent pressure on the system even if latency fluctuates during the test.

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

Once all data is extracted, the system is ready for the next run. To prepare the environment, all containers need to be stopped first. That is done by running `docker compose down -v` inside the Docker remote context, with the `-v` argument explicitly removing all docker volumes. This

[39] `performance-tests/perf_runner.py`
[40] `performance-tests/many_runs.py`

ensures a clean state by purging the persistent data stores of PostgreSQL and Axon Server.

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`.

6.9.3 Post Processing Test Results

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

6.9.4 Testing "Freshness": Time to Consistency

To assess the eventual consistency of the ES-CQRS architecture, a specialized test for the Freshness SLO was developed.[41] 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 freshnessthreshold defined in the SLO 2. After this threshold, the application is expected to have synchronized the write- and read-side. Once the script wakes up again, 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.

6.9.5 A Function for Scalability

Following [29], a function to determine the scalability of applications is established in this subsection. Their scalability metric, previously described in subsection 2.10.1, serves as a basis for the scalability metric used in this study.

The cost function described in Equation 6.1 is used.

$$C(k) = w_1 * \mathrm{CPU} + w_2 * \mathrm{S} + w_3 \left(\frac{T}{200}\right)^2 + w_4 \left(\frac{D}{10}\right)^2 \tag{6.1}$$

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

This cost function is weighted, with w_i being the weights for each metric. S corresponds to storage size. More precisely, this value is calculated using a ratio of the two applications: the application with the larger storage consumption receives a cost penalty (e.g., 2.0 for twice the usage), while the smaller footprint is "rewarded", e.g. a value of 0.5 for half the usage. Furthermore, T corresponds to $tomcat\_threads$, and D corresponds to $hikari\_connections$. These metrics are normalized to their respective ceilings.

When testing time to consistency ("freshness"), the following term gets added to the cost function: $w_5(1 - R)^4$, with R corresponding to $read\_visible\_rate$.

The concrete weights are as following: $w_1 = 1$, $w_2 = 0.5$, $w_3 = 1$, $w_4 = 1$, $w_5 = 3$. Storage consumption is weighed less, $read\_visible\_rate$ is weighed more.

Using the above cost function $C(k)$ and the following throughput function Equation 6.2 and the value function Equation 6.3, a scalability metric can be calculated.

$$\lambda(k) = RPS - dropped\_iterations\_rate \tag{6.2}$$

$$f(k) = \frac{1}{latency\_p95} \tag{6.3}$$

The implementation of this scalability function is available in `scalability_function.py`[42].

6.9.6 All Implemented Load Tests

Each Load Test (L) is listed in table Table 6.3.

[42] `performance-tests/scalability_function.py`

Test	Endpoint(s)	Name	Seed Data
L1	POST /courses	Create Courses Simple	N/A
L2	POST /courses	Create Courses Prerequisites	10 courses
L3	POST /lectures/{lectureId}/enroll	Enrollment	Professor, one student per RPS, 100 courses and lectures
L4	GET /lectures	Read lectures for student	Professor, 50 courses and lectures, 100 students. Enroll each student in a lecture
L5	GET /lectures/all	Read all lectures	Same as L4
L6	GET /stats/credits	Get credits	Professor, 50 students, 10 courses and lectures. Enroll each student in each lecture & assign grade twice (one update)
L7	POST /lectures/create, GET /lectures/{lectureId}	Time to consistency / Create lecture, then read	Professor, 50 courses
L8	GET /stats/grades/history	Grade history	Same as L6

Table 6.3: All load tests

6.10 Static Analysis

The IntelliJ plugin *MetricsReloaded*[43] was used to collect static analysis metrics from both applications.

The afferent (C_a) and efferent (C_e) coupling metrics, previously described in subsection 2.11.1, can be calculated by MetricsReloaded on a per-package basis. Furthermore, the plugin can calculate metrics for incoming and outgoing dependencies on a class-basis. These can be mapped to the afferent and efferent coupling metrics, as shown in Table 6.4. Additionally, MetricsReloaded calculates transitive (indirect) dependencies, allowing for a broader view of the system's dependencies, describing the number of packages a class depends on or the number of packages depending on a class.

Other metrics outlined in section 2.11 are also supported by MetricsReloaded, such as MOOD, CK and the stability metrics described in [31]. It should be noted that the MOOD metrics are only partly implemented, missing calculations for CLF and RF, which is why no results will be presented for those metrics.

[43] `https://plugins.jetbrains.com/plugin/93-metricsreloaded`

Plugin Metric	**Theoretical Metric**	**Definition**
Dpt	C_a (per-class)	Incoming dependencies
Dcy	C_e (per-class)	Outgoing dependencies
*Dpt**	transitive C_a	Indirect incoming dependencies
*Dcy**	transitive C_e	Indirect outgoing dependencies
PDcy	N/A	Outgoing dependencies to other packages
PDpt	N/A	Packages depending on this class

Table 6.4: Class-based dependency metrics

Chapter 7

Results

7.1 Performance

This section describes results of load testing. Each L is presented in order, describing the kind of endpoint and test that was employed.

For all load tests, the primary results are visualized through diagrams like latency-vs-load line plots. The raw numerical data, including confidence intervals and significance levels for all tested load levels, are provided in Appendix B.

7.1.1 Significance

The significance levels indicated in the tables inside the appendix are calculated using the *Mann-Whitney U* significance test. This test is used because it doesn't require normally distributed samples and compares the *rankings* of results rather than the averages, making it more resistant to outliers[90, Chapter 13.3]. In load testing, outliers (tail latencies) can create *skewing* that would incorrectly bias a standard average-based test.

7.1.2 Steady-state performance

To capture the system's median CPU usage under load, the initial usage spike ("transient"), exhibited across all runs, was excluded. This approach mitigates the influence of startup spikes and resource initialization overhead, instead focusing on the steady-state performance. This process is called *transient removal* [91].

7.1.3 Ratios

When comparing the CRUD and ES-CQRS applications' performance, a "speedup" factor or ratio is calculated per metric and RPS. This ratio may be mentioned for textual descriptions of the resulting graphs, and is always present in the tables inside the appendix. The term "speedup" is used when comparing latencies, while the term "ratio" is used for other metrics. The given values are calculated like this: $\frac{metric_{crud}}{metric_{es\_cqrs}}$. Therefore, a $latency\_p95$ of 100ms for the CRUD application and a $latency\_p95$ of 50ms for the ES-CQRS application would result in a speedup of 2x for that specific metric at the respective RPS. Ratios involving the value zero for either application are marked as *N/A*.

7.1.4 Dropped Iterations

In some tests, iterations were dropped, meaning that k6 skipped iterations and did not send requests to the server. When iterations take more than 1 second, eventually no more Virtual Users (VUs) (which act as request processors) are available to k6. The fact that iterations were dropped during testing is mentioned when describing results, and the rate of dropped iterations per second (*dropped_iterations_rate*) is available in the results tables inside the appendix.

7.1.5 Failure Rate

The *failure_rate* metric was recorded for all tests. However, in most cases, the failure rate was 0%. Therefore, the values of this metric are not present in most tables. The only exception to this is section 7.1.10.

7.1.6 Threadpool

Each load test shows the threadpool usage (*tomcat_threads*) of the application under test. The maximum value this metric can take in these tests is 200, as it SpringBoot's default configuration.

7.1.7 Database Connections

Each load test presents the number of used database connections of the application under test — *hikari_connections*. The maximum value this metric can take in these tests is 10, which can again be attributed to SpringBoot's default configuration.

It is important to clarify that a median value of zero active database connections does not imply a lack of database activity. Instead, this result is an artifact of the metric collection frequency and the statistical properties of the median. Resource consumption metrics were sampled at two-second intervals. In scenarios where database interactions completed within milliseconds, the majority of these snapshots captured the connection pool in an idle state. Consequently, while the database was used during request handling, a high rate of zero-value samples drives the median to zero.

7.1.8 Data Store Size

The size of the data store is calculated differently for both applications. The CRUD application uses PostgreSQL as its only data store. Therefore, *postgres_size* equals the application's data store size.

The total data store size for the ES-CQRS application is defined as the sum of the PostgreSQL projection size and the allocated Axon storage: $postgres\_size + axon\_storage\_size$. It is important to note that Axon Server allocates storage in fixed-size segments (or "pages") of 4MB. Because these segments are allocated eagerly, the total storage includes a minimum overhead of 8MB (one 4MB segment each for events and snapshots), regardless of the actual data density within those blocks. Consequently, the measured size represents the allocated capacity rather than the literal byte-count of the stored records.

7.1.9 Graphs

In graphs that show latencies, the shaded areas in the line graphs represent the *confidence interval (CI)* of the latency measurements. The confidence interval is the range of values that is likely to contain the true answer. It estimates the level of uncertainty by providing a margin of error

around a specific result. Furthermore, latency graphs use a logarithmic y-axis, unless mentioned otherwise.

Graphs showing resource consumption typically show the area between the 25th and 75th percentile in the shaded area, also called *Interquartile Range (IQR)*. This shows the typical range of resource usage across runs.

The lines between the measured data points are interpolated, representing an estimation of performance trends across the full range of measurements. These lines help visualize the transition between different loads, though the actual values were only recorded at the marked intervals on the x-axis. In some cases, the x-axis omits intermediate labels for improved readability. For a complete view of every value, refer to the full results provided in Appendix B.

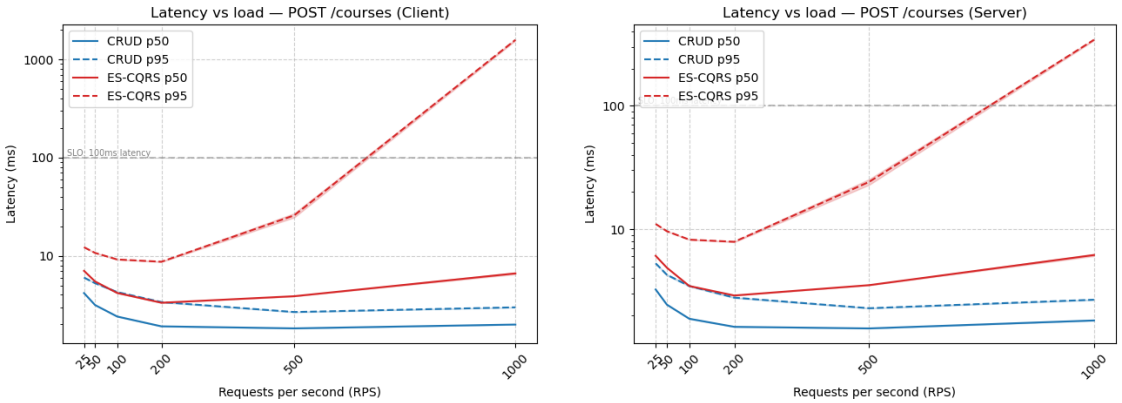
7.1.10 Write Performance

L1: Create Courses Simple

Professors can create courses using the `POST /courses` endpoint. The "simple" test case tests creation of courses *without prerequisites*. This means that no validation is necessary to create a new entity, making this test the raw insertion performance.

Figure 7.1 presents line graphs comparing the endpoint's latency under increasing load. The graphs describe the observed client-side and server-side latencies under varying loads of both applications by showing their *latency_p50* (median) and *latency_p95* (tail latency). SLO 1 defined the threshold for *latency_p95* at 100ms, meaning the ES-CQRS application failed to satisfy the SLO between 500 and 1000 RPS. Additionally, its *dropped_iterations_rate* reaches a value of 52 at 1000 RPS. Meanwhile, the CRUD application achieves a sub 10ms *latency_p95* until at least 1000 RPS.

It can be noted that the latency curve follows the same pattern in both the measurements made on the client and on the server, except for the ES-CQRS *latency_p95* which exhibits a value of over 1000ms on the client, but only around 300ms on the server.



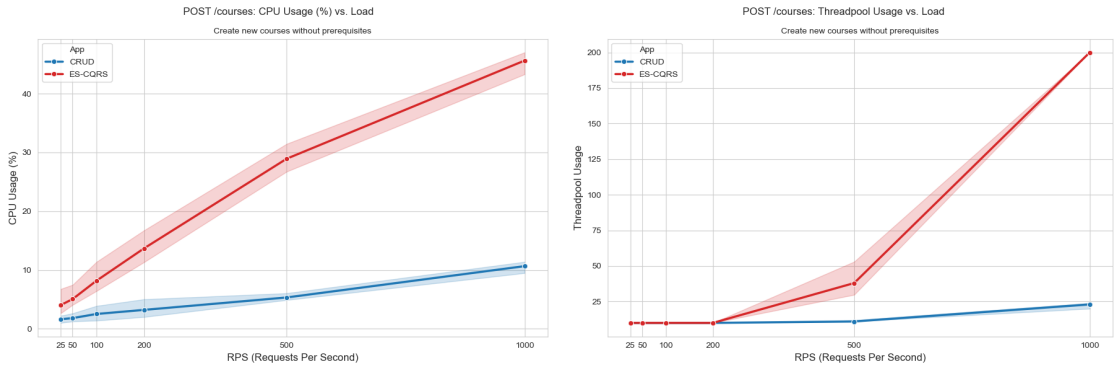
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.1: L1 Performance Metrics: Load measured in RPS. Detailed results in B.1.

Figure 7.2a presents median CPU usage of the endpoint under increasing load. The ES-CQRS application generally has a higher CPU usage, exhibiting a value of ≈45% at 1000 RPS, while the CRUD application uses ≈10%. The CRUD application's CPU usage rises linearly, while the ES-CQRS application's CPU usage rises slower past 500 RPS.

The threadpool usage of both applications is visualized in Figure 7.2b. At 500 RPS, ES-CQRS starts using more threads, finally reaching threadpool saturation — meaning that all 200 possible threads of the threadpool are used — at 1000 RPS. The CRUD application uses at most around 25 threads at 1000 RPS.

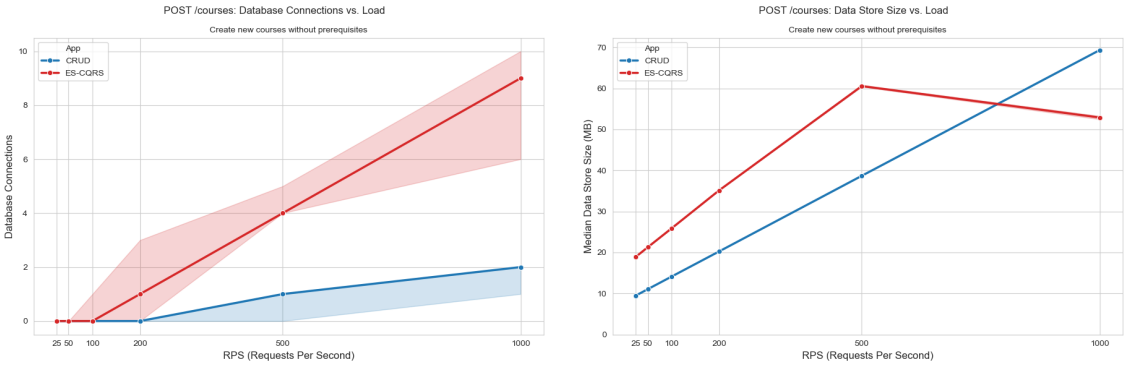


(a) CPU Usage (%) vs. Load

(b) Threadpool Usage vs. Load

Figure 7.2: L1 Resource Usage: Load measured in RPS. Detailed results in B.1.

In Figure 7.3b, the size of the data store under increasing load is presented. The CRUD application's data store, consisting only of the PostgreSQL database, exhibits a linear growth, reaching a size of ≈70MB at 1000 RPS. With 100,000 requests sent at 1000 RPS, this comes out to around 0.7kB per persisted course. The ES-CQRS graph also grows linearly. At 500 RPS, the size of the data store is about 60MB, a 1.5x higher storage consumption than the CRUD application. This equals around 1kB per persisted course. However, at 1000 RPS, the recorded storage size is lower than in the previous test. This can be attributed to a projection lag caused by the eventual consistency present in the application and is explained further in subsection 8.1.3. The true storage consumption of the ES-CQRS application can be assumed to be around 120MB, once all projections are updated.



(a) Database Connections vs. Load

(b) Data Store Size vs. Load

Figure 7.3: L1 Database Usage: Load measured in RPS. Detailed results in B.1.

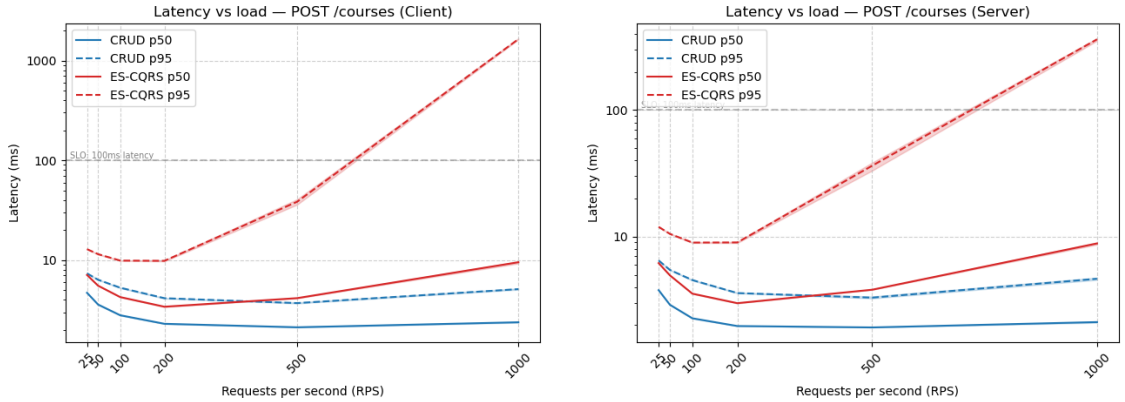
These values correspond to the following scalability metrics for the CRUD application: $\psi(200, 500) \approx 5.4$, $\psi(500, 1000) \approx 1.4$; and the following scalability metrics for the ES-CQRS application: $\psi(200, 500) \approx 0.6$, $\psi(500, 1000) \approx 0.01$. Note that the prior assumption of 120MB storage size was used to calculate the value for ES-CQRS's productivity at 1000 RPS.

L2: Create Courses Prerequisites

This test also evaluates the performance of the endpoint described in section 7.1.10. However, before executing the load generation, a set of "prerequisite" courses are generated. Each iteration, a random number of these courses (0 to 5) are selected which are referenced during load generation. This creates the necessity to do additional checks on existing data, verifying whether the referenced courses actually exist.

The endpoint's performance, presented in Figure 7.4, is similar to the observations made in L1. Once the load exceeds 500 RPS, the ES-CQRS application fails to satisfy SLO 1 with a $latency\_p95$ crossing the threshold of 100ms. Additionally, its $dropped\_iterations\_rate$ reaches a value of around 60. The CRUD application, on the other hand, is able to fulfill the latency threshold up to at least 1000 RPS without dropping iterations.

It can be noted that the latency curve follows the same pattern in both the measurements made on the client and on the server, except for the ES-CQRS $latency\_p95$ which exhibits a value of over 1000ms on the client, but only ≈360ms on the server at 1000 RPS.



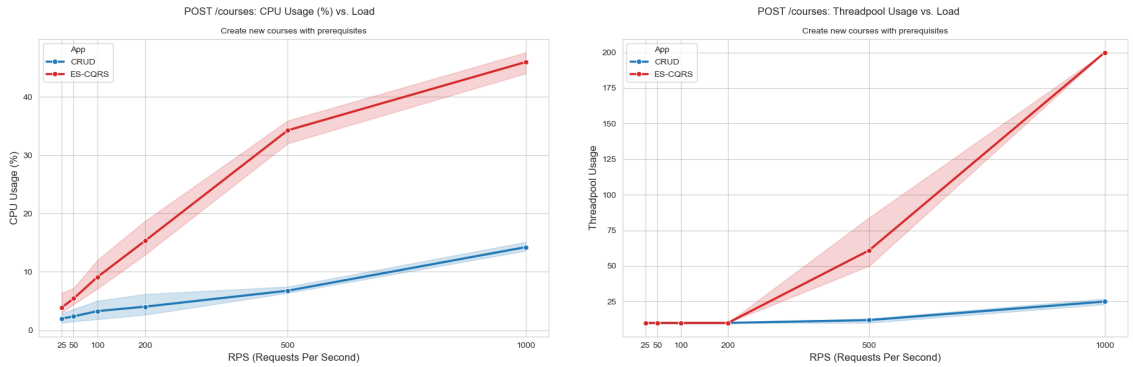
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.4: L2 Performance Metrics: Load measured in RPS. Detailed results in B.2.

Figure 7.5a presents the median cpu_usage of the endpoint under increasing load. The ES-CQRS application has a higher CPU usage, exhibiting a value of ≈45% at 1000 RPS, while the CRUD application uses ≈10%. The CRUD application's CPU usage rises linearly, while the ES-CQRS application's CPU usage rises slower past 500 RPS.

The threadpool usage of both applications is visualized in Figure 7.5b. At 200 RPS, ES-CQRS starts using more threads, finally reaching threadpool saturation at 1000 RPS. The CRUD application uses at most around 25 threads at 1000 RPS.



(a) Resource Usage (%) vs. Load

(b) Threadpool Usage vs. Load

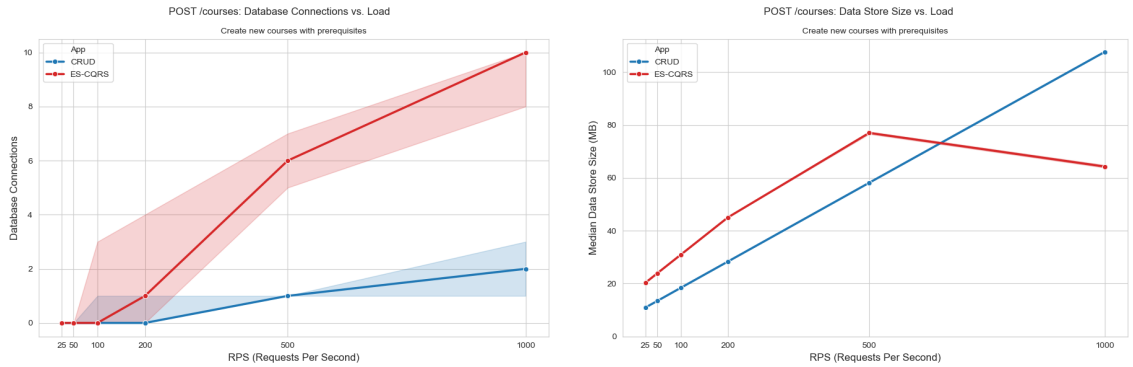
Figure 7.5: L2 Resource Usage: Load measured in RPS. Detailed results in B.2.

Figure 7.6 presents database usage statistics of both applications under increasing load.

The median used database connections are shown in Figure 7.6a. At 1000 RPS, the ES-CQRS application uses all available connections, while the CRUD application utilizes around 2.

In Figure 7.6b, the size of the data store under increasing load is presented. The CRUD application's data store exhibits a linear growth, reaching a size of ≈107MB at 1000 RPS. With 100,000 requests sent at 1000 RPS, this comes out to roughly 1.1kB per persisted course — around 60% higher than when creating courses without prerequisites.

The ES-CQRS application's storage consumption initially follows a linear growth pattern. At 200 RPS, the data store occupies approximately 45MB, representing a 1.6x increase compared to the CRUD application—roughly 2.3kB per persisted course. However, as the load increases beyond 500 RPS, the observed storage growth decelerates, eventually showing a decline at 1000 RPS. Again, this can be attributed to projection lag. Once the system updates all projections, the true storage consumption of the ES-CQRS application is estimated to be approximately 225MB.



(a) Database Connections vs. Load

(b) Data Store Size vs. Load

Figure 7.6: L2 Database Usage: Load measured in RPS. Detailed results in B.2.

These values correspond to the following scalability metrics for the CRUD application: $\psi(200, 500) \approx 2.5$, $\psi(500, 1000) \approx 1.1$; and the following scalability metrics for the ES-CQRS application: $\psi(200, 500) \approx 0.4$, $\psi(500, 1000) \approx 0.02$.

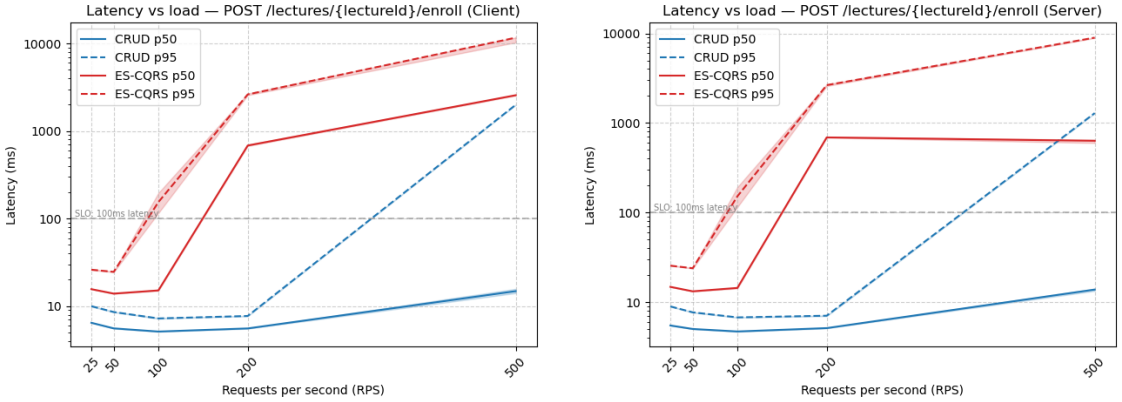
L3: Enrollment

The endpoint `POST /lectures/{lectureId}/enroll` enrolls students to a lecture. Its performance under load up to 500 RPS is visualized in Figure 7.7.

The ES-CQRS application's *latency_p95* rises above 100ms between 50 and 100 RPS, violating SLO 1. The CRUD application's *latency_p95* and *latency_p50* remain below 10ms until 200 RPS, representing more than a 100x slowdown for the ES-CQRS application. Between 200 and 500 RPS, CRUD's P95 latency rises to over 1000ms, violating SLO 1. At 500 RPS, ES-CQRS' *latency_p95* exceeds 10.000ms.

It is worth noting that starting from 200 RPS, the ES-CQRS's load tests started dropping iterations. *dropped_iterations_rate* reached a peak value of roughly 240 at 500 RPS, at which point the CRUD application also exhibited dropped iterations with a rate of around 36.

Additionally, ES-CQRS also shows a failure rate of around 6% at 500 RPS. This can be attributed to the endpoint's implementation. First, the enrollment command is sent to the query gateway. After waiting for the command to complete, a subscription query is issued. If the read-side fails to respond to this query within a 10 second threshold, the request fails.



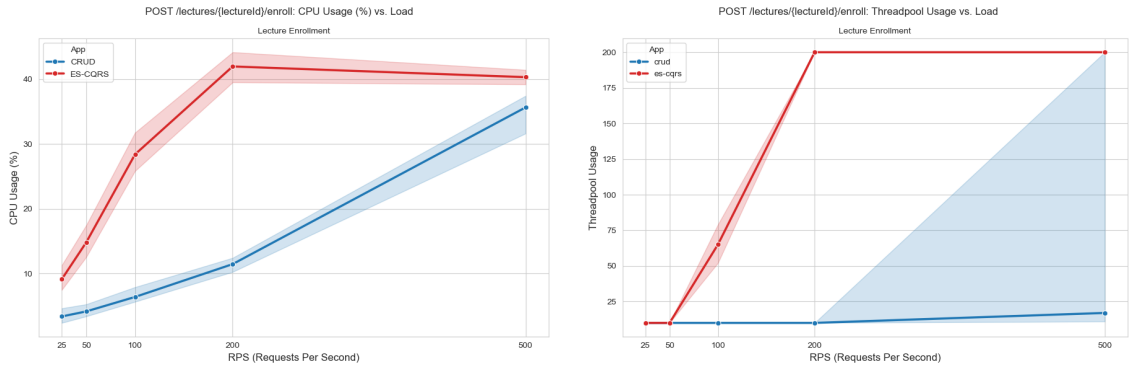
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.7: L3 Performance Metrics: Load measured in RPS. Detailed results in B.3.

Figure 7.8a compares CPU utilization by both applications as the load increases. The ES-CQRS application consistently requires more CPU than the CRUD application across all tested loads. At lower intensities (below 200 RPS), the ES-CQRS CPU usage grows, before peaking at around 42% at 200 RPS and remaining at that level for 500 RPS. The CRUD application shows a steady increase in CPU consumption as load increases, reaching aroud 36% at 500 RPS.

Figure 7.8b illustrates the number of active threads utilized by each application to process the incoming load. At 25 and 50 RPS, both applications maintain an identical, stable baseline of 10 threads. Beyond 50 RPS, the utilized threads for ES-CQRS begin to increase. It reaches a maximum capacity of 200 threads at in between 100 and 200 RPS and remains saturated at that limit through 500 RPS. In contrast, the CRUD application maintains its baseline of 10 threads until 200 RPS. Even at the maximum load of 500 RPS, its median thread usage remains at around 17 threads. However, it should be noted that the IQR at 500 RPS spans up to 200 threads, indicating a high variance in the measured results.



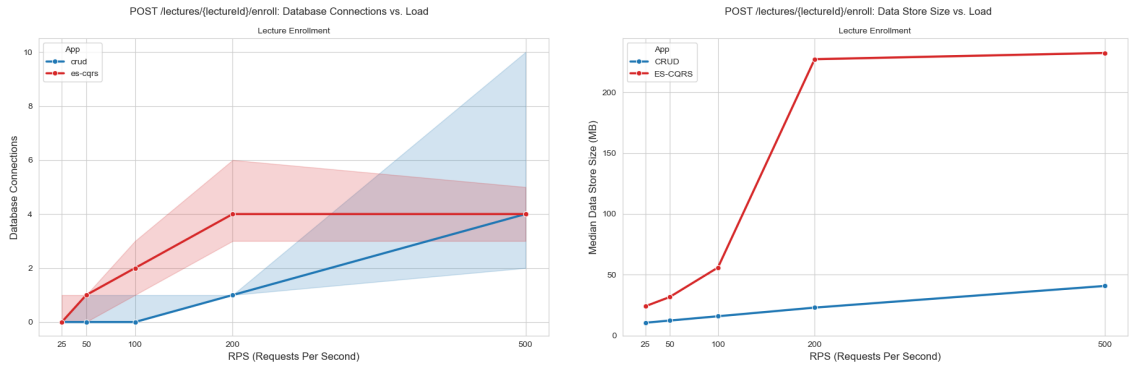
(a) Resource Usage (%) vs. Load

(b) Threadpool Usage vs. Load

Figure 7.8: L3 Resource Usage: Load measured in RPS. Detailed results in B.3.

Database usage under increasing load is visualized in Figure 7.9. Figure 7.9a shows the median active database connections. It can be seen that the ES-CQRS application reaches its peak at around 4 connections simultaneously at 200 RPS, plateauing for the final measured load of 500 RPS. On the other hand, the CRUD application uses a median of 0 connections until 100 RPS. At this point, the graph starts rising, reaching a median of 4 at 500 RPS with a wide IQR spanning from 2 to 10.

Size of the data store is presented in Figure 7.9b. The CRUD application shows a linear growth, reaching a final size of 41MB. In contrast, ES-CQRS shows a sharper growth. At 100 RPS, its size is already around 56MB, at 200 RPS, its size is 227MB. Doubling the amount of requests increased the data store's size by a factor of almost 4! At 500 RPS, the recorded size is only slightly higher than at 200 RPS, reaching a final value of 232MB. Once again, this indicates a projection lag.



(a) Database Connections vs. Load

(b) Data Store Size vs. Load

Figure 7.9: L3 Database Usage: Load measured in RPS. Detailed results in B.3.

These values correspond to the following scalability metric for the CRUD application: $\psi(50, 100) \approx 2.7$; and the following scalability metric for the ES-CQRS application: $\psi(50, 100) \approx 0.2$.

7.1.11 Read Performance

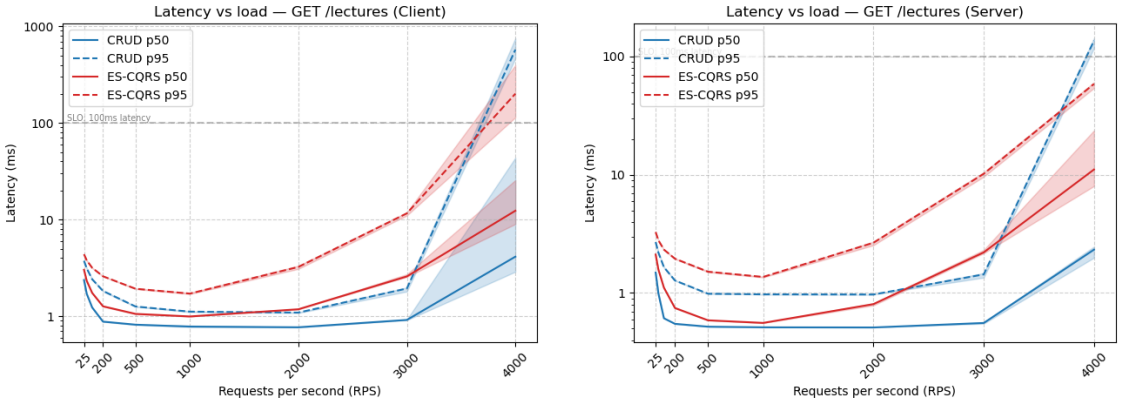
When measuring read performance, no data is created during load generation. Therefore, the visualizations of data store size compared to load are replaced by simple tables, as each test creates the exact same amount of data.

L4: Read Lectures for Student

`GET /lectures` returns all lectures a student is enrolled or waitlisted in. Figure 7.10 presents client-side and server-side latencies for this endpoint.

In the client-side graph, shown in Figure 7.10a, both applications maintain a *latency_p50* and *latency_p95* of 10ms or less up to 3000 RPS. At 3000 RPS, the ES-CQRS application exhibits an around 6x higher *latency_p95* than the CRUD app. Beyond 3000 RPS, though, the CRUD's *latency_p95* overtakes ES-CQRS. At 4000 RPS, the ES-CQRS application exhibits a *latency_p95* of around 200ms, which is around 2.9x faster than the CRUD application at 620ms. However, CRUD's *latency_p50* stays lower than the one of ES-CQRS, even though the data shows a large confidence interval, meaning the results are unstable and exhibit high variance. Both applications violate SLO 1 beyond 3000 RPS.

The server-side graph, presented in Figure 7.16b, initially shows a similar pattern. Once exceeding 3000 RPS, the latencies also start increasing, however, the observed increase is not as strong as in the client-side latencies. At 4000 RPS, the ES-CQRS application shows a *latency_p95* of around 60ms, the CRUD application has a *latency_p95* of around 130ms. This indicates a queueing delay which occurs before the request starts being processed on the server.



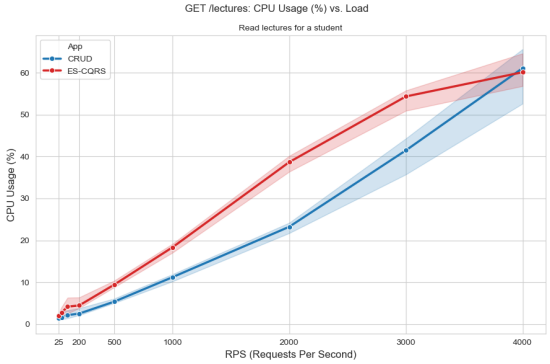
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

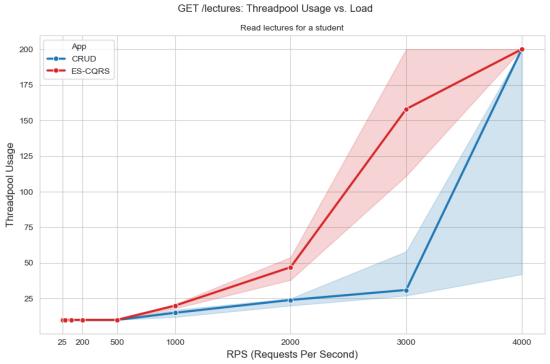
Figure 7.10: L4 Performance Metrics: Load measured in RPS. Detailed results in B.4.

Regarding CPU usage, pictured in Figure 7.11a, both applications exhibit a linear increase as the load intensifies. Between 25 and 3000 RPS, the ES-CQRS application consistently maintains a higher median CPU usage compared to the CRUD application, reaching a peak relative difference at 100 RPS where it is 1.9x higher. As the load reaches 4000 RPS, the two applications converge at approximately 60% usage, with the CRUD application showing a slightly higher median of 61% and a wider confidence interval.

Figure 7.11b visualizes the application's threadpool usage. Both applications remain stable at a baseline of 10 threads for loads between 25 and 500 RPS. As the load increases to 1000 RPS, the ES-CQRS application begins using more threads, climbing to 47 threads at 2000 RPS and 158 threads at 3000 RPS, which is 5.1x higher than the CRUD application at that same point. The CRUD application maintains a lower threadpool usage until it reaches 3000 RPS. Finally, at the maximum load of 4000 RPS, both applications reach an identical ceiling of 200 threads.



(a) Resource Usage (%) vs. Load

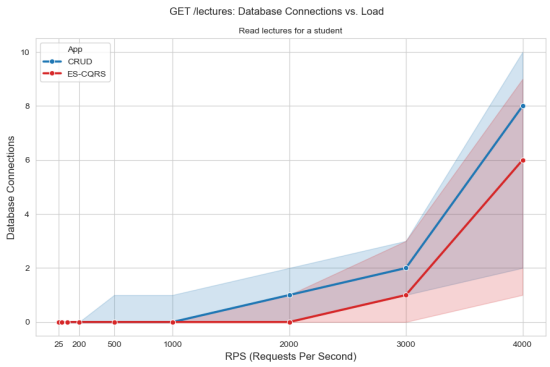


(b) Threadpool Usage vs. Load

Figure 7.11: L4 Resource Usage: Load measured in RPS. Detailed results in B.4.

Active database connections are visualized in Figure 7.12a. Both the CRUD and ES-CQRS applications maintain a median of zero active connections for loads ranging from 25 to 1000 RPS. The CRUD application begins to utilize more connections first, reaching a median of 1 at 2000 RPS and 2 at 3000 RPS. During this same interval, the ES-CQRS application remains at a median of zero connections until 3000 RPS, where it records a median of 1, making it 2x lower than the CRUD application. At the maximum load of 4000 RPS, database connection usage increases for both, with the CRUD application reaching a median of 8 and the ES-CQRS application reaching a median of 6.

In terms of data store size for seed data, shown in Figure 7.12b, the ES-CQRS application requires significantly more storage than the CRUD application. Specifically, the CRUD application uses around 9MB, while the ES-CQRS application uses around 17MB, a 2x higher storage value.



(a) Database Connections vs. Load

	CRUD (ms)		ES-CQRS (ms)			
RPS	Median	CI ±	Median	CI ±	Ratio	Significance
All	8.67MB	0.01	17.31MB	0.01	2.0x Higher	***

(b) Data Store Size

Figure 7.12: L4 Database Usage: Load measured in RPS. Detailed results in B.4.

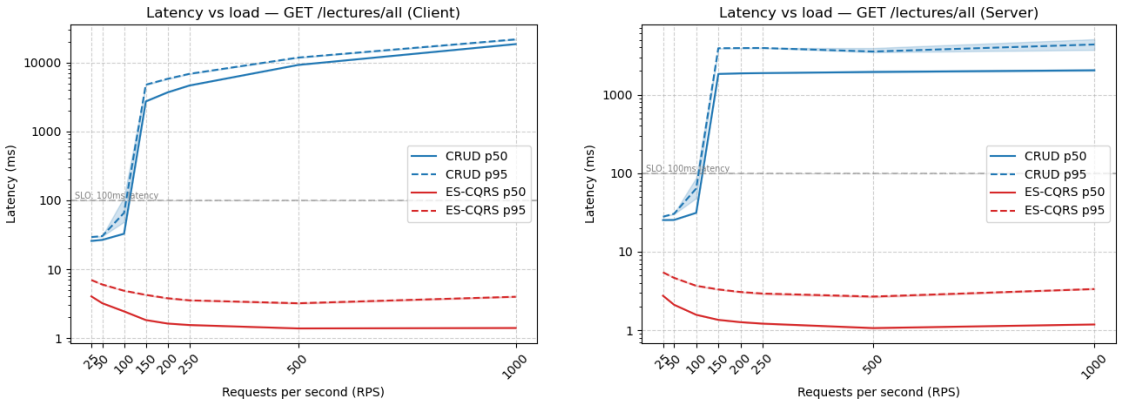
These values correspond to the following scalability metrics for the CRUD application: $\psi(2000, 3000) \approx 0.6$, $\psi(3000, 4000) \approx 0.00$; and the following scalability metrics for the ES-CQRS application: $\psi(2000, 3000) \approx 0.3$, $\psi(3000, 4000) \approx 0.05$.

L5: Read All Lectures

The `GET /lectures/all` endpoint returns a list of all lecture details. 7.13a shows the client-side latency of the endpoint under increasing load. The ES-CQRS application consistently maintains a

sub 10ms *latency_p95*, while the CRUD application violates item SLO 1 at 100 RPS and higher, reaching latencies beyond 10.000ms.

At 150 RPS and higher, the CRUD application starts dropping iterations. At 1000 RPS, *dropped_iterations_rate* reaches a value of more than 560, meaning that 56% of all iterations were not executed.

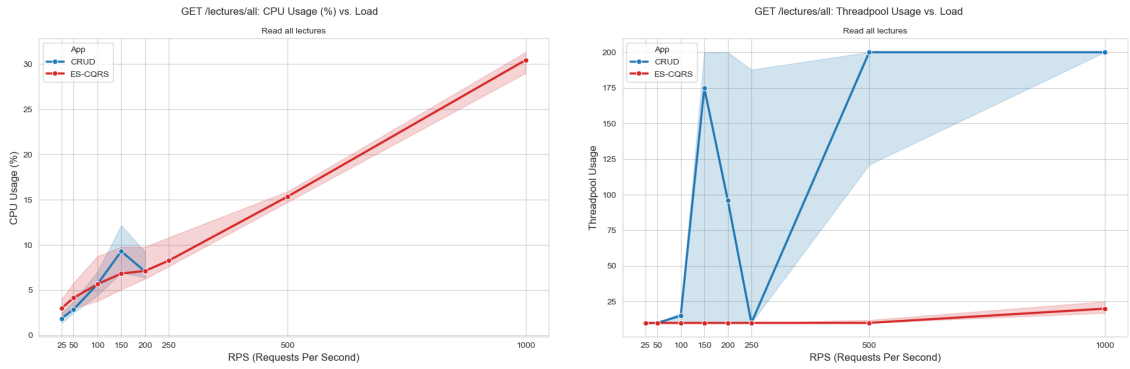


(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.13: L5 Performance Metrics: Load measured in RPS. Detailed results in B.5.

In Figure 7.14a, CPU usage of the SpringBoot application is presented under increasing load. The ES-CQRS application shows a linear increase. In CRUD, the CPU usage also starts to increase linearly, before a smaller value is observed from 150 RPS to 200 RPS. After 200 RPS, no further values could be collected using Prometheus. Potential reasons for this are mentioned in subsection 8.1.2. Figure 7.14b shows threadpool usage of both applications. The ES-CQRS application remains at 10 threads for most of the load, reaching its highest value of 20 at a load of 1000 RPS. In contrast, the CRUD application shows a sharp increase in thread usage at 150 RPS, before the values at 200 RPS and 250 RPS show a sharp decline, reaching the minimum of 10 threads. At the same time, the IQR (shaded blue area) shows that many of the samples still fell close to the ceiling of 200 threads. At 500 RPS and further, the median of *tomcat_threads* stabilizes at 200.



(a) Resource Usage (%) vs. Load

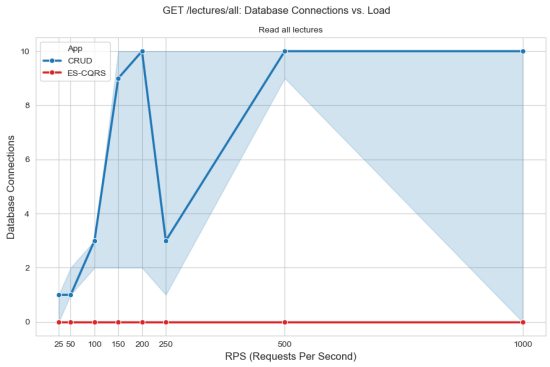
(b) Threadpool Usage vs. Load

Figure 7.14: L5 Resource Usage: Load measured in RPS. Detailed results in B.5.

Figure 7.15a presents the median of used database connections (*hikari_connections*) under increasing load. The ES-CQRS application maintains a median of 0, while CRUD quickly rises to

the ceiling of 10 concurrent connections, reaching its peak at 200 RPS. However, similarly to the threadpool usage, the number of database connections is lower at 250 RPS before increasing back to 10 at 500 RPS.

In terms of data store size, presented in Figure 7.15b, the ES-CQRS application uses around 17MB to store its events and projections, twice as much storage as CRUD at 8MB.



(a) Database Connections vs. Load

	CRUD (ms)		ES-CQRS (ms)			
RPS	Median	CI ±	Median	CI ±	Ratio	Significance
All	8.4	0.0	16.84	0.01	2.0x Higher	***

(b) Data Store Size

Figure 7.15: L5 Database Usage: Load measured in RPS. Detailed results in B.5.

The observed collapse in resource consumption in the CRUD application between 200 RPS and 250 RPS represents an anomaly, as the metrics drop substantially despite the increasing load. These results were confirmed through repeated test executions. An attempt at further analysis will be given in subsection 8.1.2.

L6: Get Credits

The `GET /stats/credits` endpoint retrieves a student's total collected credits. Figure 7.16 presents the endpoint's client-side and server-side latencies.

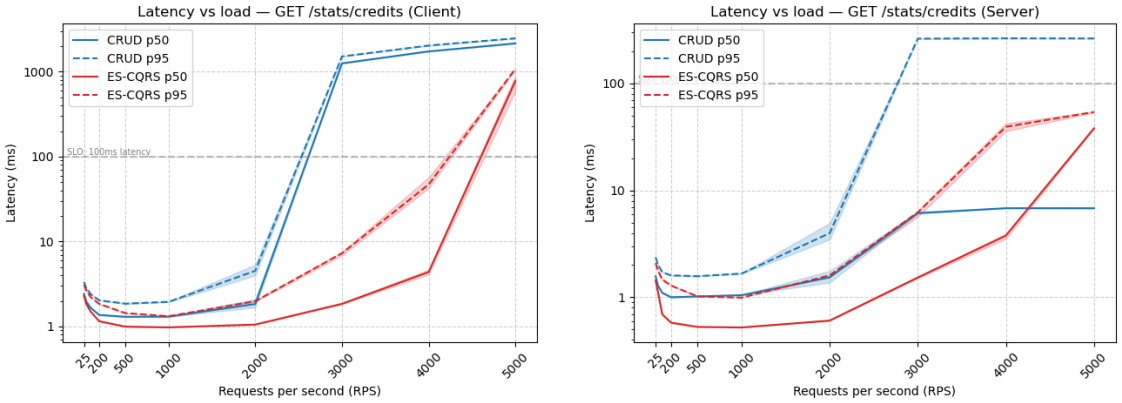
Figure 7.16a presents the client-side latency under increasing load. It can be seen that the $latency\_p95$ remains below 10ms for both applications until around 2000 RPS.

Between 2000 RPS and 3000 RPS, a performance divergence occurs: the CRUD application's $latency\_p50$ and $latency\_p95$ increase to more than 1000ms, violating SLO 1 beyond 2000 RPS. On the other hand, the ES-CQRS application still exhibits a $latency\_p95$ of less than 10ms at 3000 RPS. This represents a speedup of over 200x. At 4000 RPS, the ES-CQRS application exhibits a $latency\_p95$ of around 50ms, a 40x speedup. Beyond 4000 RPS, the ES-CQRS app also begins to violate SLO 1, reaching latencies around 1000ms at 5000 RPS.

It can be noted that starting at 3000 RPS, the CRUD application's latencies seem to reach a plateau, with the observed latencies at 3000, 4000 and 5000 RPS being very similar. Meanwhile, the ES-CQRS application's latencies keep increasing up to a load of 5000 RPS. At this load, the ES-CQRS application's $latency\_p95$ is about 2.4x lower than the CRUD application's.

At 3000 RPS, the CRUD application has a $dropped\_iterations\_rate$ of around 360. Until 5000 RPS, the value reaches 1581. Dropping iterations artificially keeps queues on the server shorter, which could explain the observed latency plateau in CRUD. It should also be noted that at 5000 RPS, ES-CQRS begins to drop some iterations with $dropped\_iterations\_rate \approx 38$.

The server latencies are displayed in Figure 7.16b. At higher loads, the same plateaus can be observed, with the ES-CQRS application also appearing to reach a plateau with a server-side $latency\_p95$ just below 100ms.



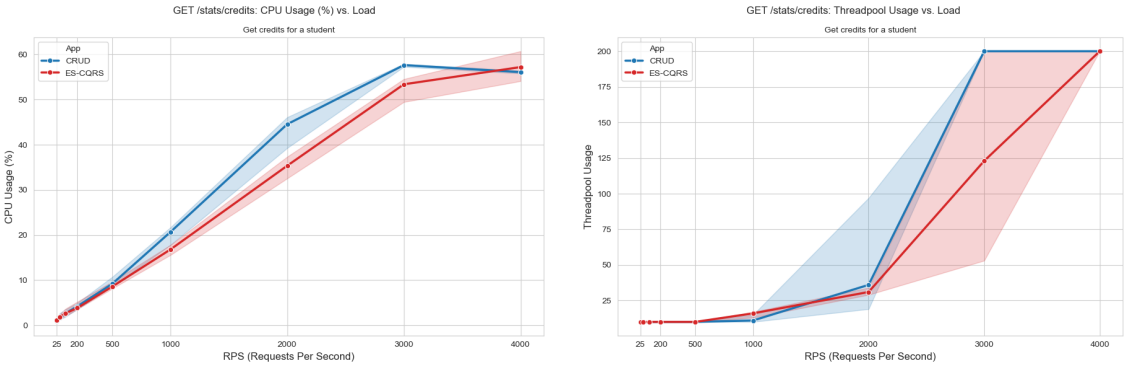
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.16: L6 Performance Metrics: Load measured in RPS. Detailed results in B.6.

Figure 7.17a shows CPU usage of both applications from 25 to 4000 RPS. At the lowest load of 25 RPS, both applications exhibit a median *cpu_usage* close to 0. As the load increases to 1000 RPS, the CRUD median reaches around 21% compared to ≈17% for ES-CQRS. At 2000 RPS, the gap widens with CRUD at 45% and ES-CQRS at 35%, a 1.3x lower value for ES-CQRS. Both applications peak near 4000 RPS, where CRUD records 56% and ES-CQRS records 57%. The IQR remains narrow for both applications across most data points.

Threadpool usage is visualized in Figure 7.17b. From 25 to 500 RPS, both CRUD and ES-CQRS maintain a constant median threadpool usage of 10. At 1000 RPS, the values begin to diverge, with CRUD at 11 and ES-CQRS at 16. At 2000 RPS, CRUD usage rises to 36 while ES-CQRS is at 31. An increase occurs at 3000 RPS, where CRUD reaches a median of 200 utilized threads, while ES-CQRS reaches 123. By 4000 RPS, both applications reach a maximum median value of 200. The graph shows a wide IQR for ES-CQRS at 3000 RPS. In contrast, CRUD has a very tight IQR beyond 3000 RPS, indicating that the median of 200 utilized threads remains constant across most runs.



(a) Resource Usage (%) vs. Load

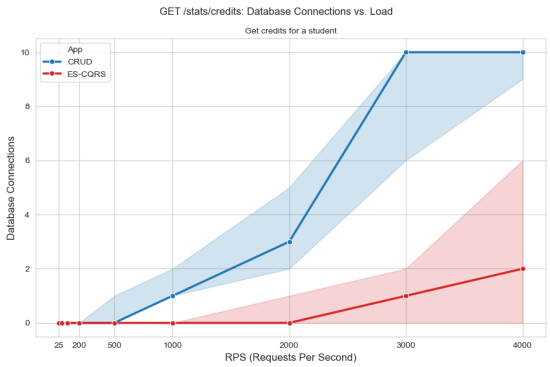
(b) Threadpool Usage vs. Load

Figure 7.17: L6 Resource Usage: Load measured in RPS. Detailed results in B.6.

Figure 7.18 illustrates the median number of active database connections under increasing load. Until 500 RPS, both applications maintain a value of 0. Beyond 500 RPS, the CRUD application's value increases, reaching a ceiling of 10 active connections at 3000 RPS. In contrast, the ES-CQRS application's value starts rising linearly only at 2000 RPS, reaching its maximum of

2 active connections at 4000 RPS, with an IQR spanning from 0 to 6.

The size of the data store is presented in Figure 7.18b. At 19.6MB, the ES-CQRS application uses twice as much storage to store the seed data than the CRUD application, which uses around 9.6MB.



(a) Database Connections vs. Load

	CRUD (ms)		ES-CQRS (ms)			
RPS	Median	CI ±	Median	CI ±	Ratio	Significance
All	9.55	0.0	19.58	0.03	2.1x Higher	***

(b) Data Store Size

Figure 7.18: L6 Database Usage: Load measured in RPS. Detailed results in B.6.

These values correspond to the following scalability metrics for the CRUD application: $\psi(1000, 2000) \approx 0.4$, $\psi(2000, 3000) \approx 0.00$; and the following scalability metrics for the ES-CQRS application: $\psi(1000, 2000) \approx 1.1$, $\psi(2000, 3000) \approx 0.3$.

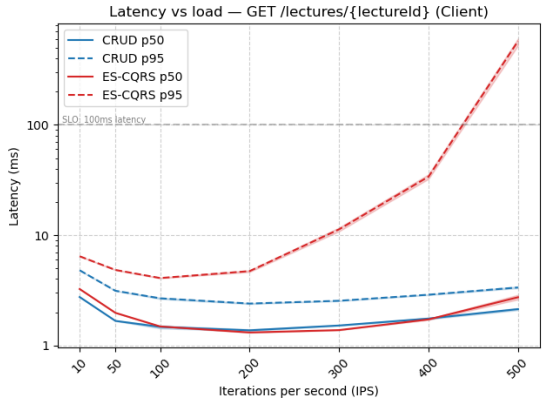
7.1.12 Time to Consistency / Freshness

SLO 2 defined a threshold of 100ms inside which all writes shall be reflected on the read-side. This "freshness" is measured in the following test.

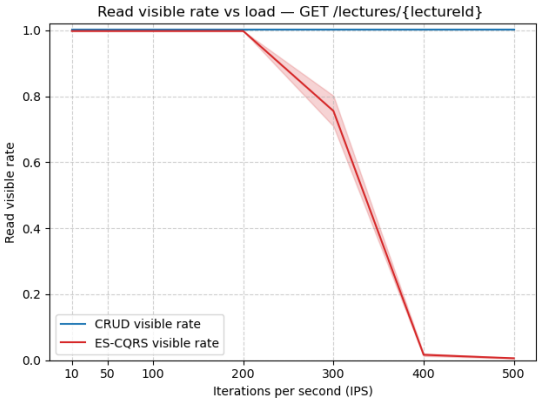
L7: Create Lecture, then Read

This load test differs from others in the fact that each iteration executes 2 HTTP requests. The first request creates a lecture. After sleeping for 100ms — the consistency threshold defined in SLO 2 — the script executes a request to `GET` the created lecture. If status code 404 is returned, the write was not reflected in the read model in time. The rate of successful reads is recorded as a metric called *read_visible_rate*, presented in Figure 7.19b. It can be seen that once exceeding 200 Iterations per Second (IPS), the ES-CQRS application failed to synchronize the read-side fast enough. Figure 7.19a shows the latencies under varying loads. *latency_p50* remains similar for both applications, however the ES-CQRS application's *latency_p95* increases with rising IPS, finally violating the threshold of 100ms defined in SLO 1 once exceeding 400 IPS.

The CRUD application consistently maintains a 100% *read_visible_rate*.



(a) Latency vs. Load, visualized for the subsequent GET request

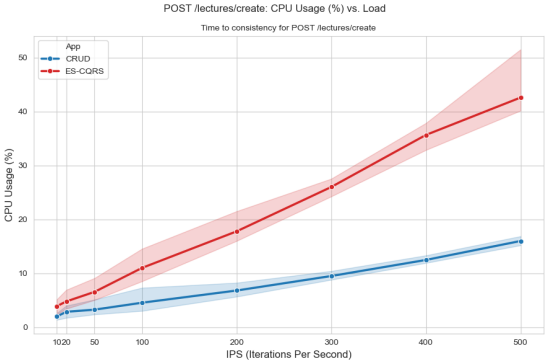


(b) Rate of visible reads

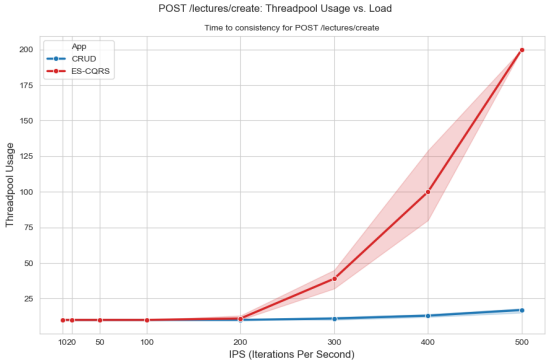
Figure 7.19: L7 Performance Metrics, load measured in IPS with 2 requests per iteration. Detailed results in B.7

cpu_usage of both applications is plotted in Figure 7.20a. A linear increase in usage can be observed in both cases, with the ES-CQRS application consistenly utilizing more CPU. At the final tested load of 500 RPS, ES-CQRS reaches a value above 40%, while CRUD utilizes around 16%.

Threadpool usage of both applications is visualized in Figure 7.20b. In both cases, the value remains very close to the base value of 10. Beyond 200 RPS, ES-CQRS starts to use more threads, reaching a utilization of 100 threads at 400 RPS, and reaching the ceiling of 200 threads at the final tested load of 500 RPS. CRUD remains below 20 utilized threads consistently.



(a) Resource Usage (%) vs. Load



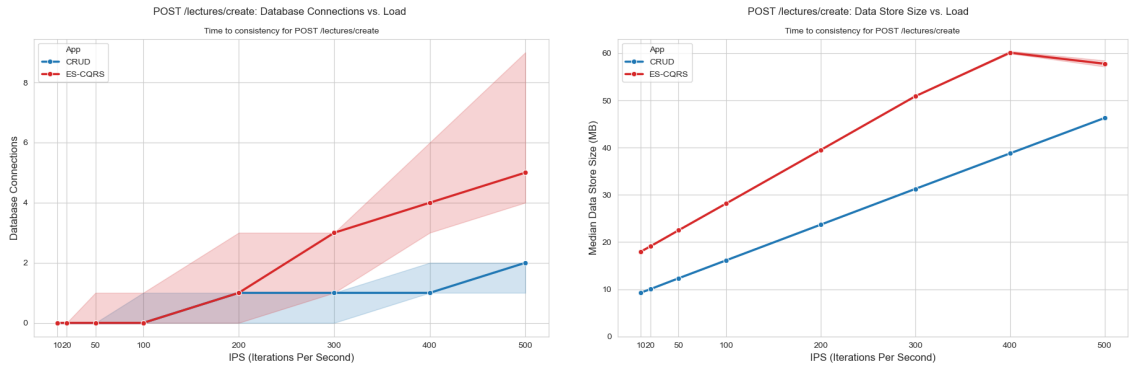
(b) Threadpool Usage vs. Load

Figure 7.20: L7 Resource Usage: Load measured in RPS. Detailed results in B.7.

Active database connections are visualized in Figure 7.21a. Both the CRUD and ES-CQRS applications maintain a median of zero active connections for loads up to 100 RPS, though the IQR shows that ES-CQRS uses some database connections. At 200 RPS, both applications use a median of 1 database connections. CRUD stays at this level until 500 RPS, where it utilizes 2 connections. ES-CQRS, in contrast, begins claiming more connections, reaching a final value of 5 at 500 RPS.

In terms of data store size, shown in Figure 7.21b, ES-CQRS again utilizes more storage. Both applications show a linear increase. The CRUD application shows a final data store size of around 46MB. This comes out to ≈1kB per created lecture. ES-CQRS reaches its peak at 400 RPS with

a value of 60MB, around 1.5kB per lecture — 1.6x higher than CRUD. However, at 500 RPS, the recorded storage size drops slightly compared to 400 RPS, indicating a projection lag.



(a) Database Connections vs. Load

(b) Data Store Size vs. Load

Figure 7.21: L7 Database Usage: Load measured in RPS. Detailed results in B.7.

These values correspond to the following scalability metric for the CRUD application: $\psi(300, 400) \approx 1.2$; and the following scalability metric for the ES-CQRS application: $\psi(300, 400) \approx 0.3$. It should be noted that the latency used for calculation was the sum of both requests' $latency\_p95$.

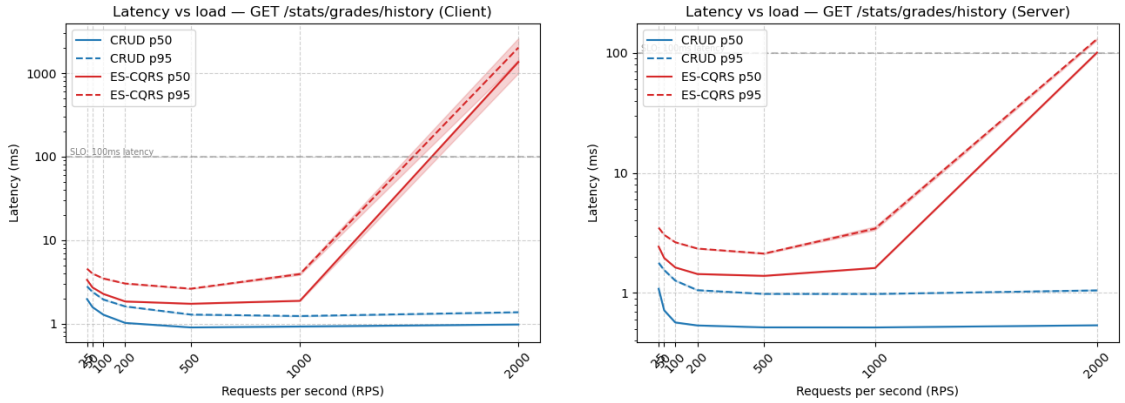
7.1.13 Historic Reconstruction

This subsection presents load tests which evaluate the performance of endpoints reconstructing historical state.

L8: Grade History

The endpoint `GET /stats/grades/history` returns the historical states of a grade. Its client-side and server-side latencies are presented in Figure 7.22. Up to a load of 1000 RPS, both applications maintain a sub-10ms $latency\_p50$ and $latency\_p95$. Beyond that point, though, the ES-CQRS application's latencies increase sharply, reaching values above 1000ms. This marks the point at which the ES-CQRS application fails SLO 1. The CRUD application, on the other hand, remains at a $latency\_p95$ of just around 2ms at the final tested load of 2000RPS.

The server latencies, visualized in Figure 7.22b, paint a similar picture. The CRUD application's server-side response times stay at around 1ms, while the ES-CQRS application's response time increases to over 100ms at 2000 RPS.



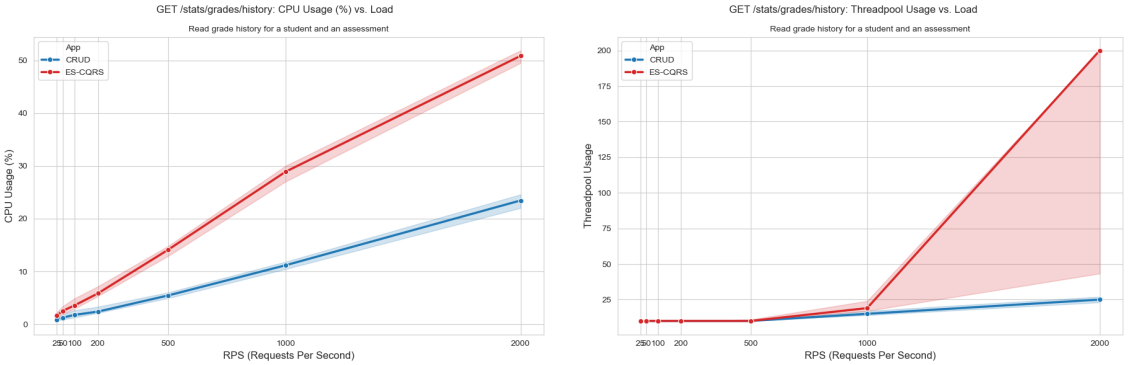
(a) Latency vs. Load measured on the client

(b) Latency vs. Load measured on the server

Figure 7.22: L8 Performance Metrics, load measured in RPS. Detailed results in B.8

The applications' *cpu_usage* is depicted in Figure 7.23a. Both implementations exhibit a linear growth, with the CRUD application reaching its maximum at around 23% and ES-CQRS reaching a maximum *cpu_usage* of 51%, which is a 2.2x increase.

tomcat_threads is visualized in Figure 7.23b. Both applications show a similar thread usage up to 1000 RPS, staying below a median value of 25. At 2000 RPS, the ES-CQRS application uses a median of 200 threads, with a wide IQR reaching from around 40 to 200. The CRUD application stays at a median thread utilization of 25.



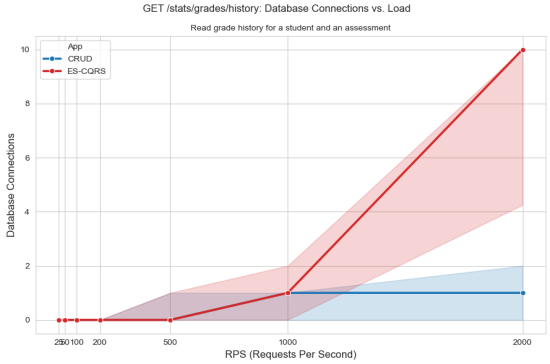
(a) Resource Usage (%) vs. Load

(b) Threadpool Usage vs. Load

Figure 7.23: L8 Resource Usage: Load measured in RPS. Detailed results in B.8.

Figure 7.24a shows the median number of utilized database connections — *hikari_connections* — under increasing load. Up to 1000 RPS, both applications show identical behavior, staying at a median of 0 until 500 RPS, before increasing to 1 at 1000 RPS. Afterward, however, the ES-CQRS application shows a sharp increase, reaching a median of 10 at 2000 RPS with a wide IQR spanning from 4 to 10. At this load, the CRUD application still remains at a median of 1.

Figure 7.24b shows the storage taken up by the test's seed data. The ES-CQRS application takes up twice the amount of storage at ≈19.7MB, while CRUD uses around 9.8MB.



(a) Database Connections vs. Load

	CRUD (ms)		ES-CQRS (ms)			
RPS	Median	CI ±	Median	CI ±	Ratio	Significance
All	9.82	0.0	19.65	0.02	2.0x Higher	***

(b) Data Store Size

Figure 7.24: L8 Database Usage: Load measured in RPS. Detailed results in B.8.

These values correspond to the following scalability metrics for the CRUD application: $\psi(500, 1000) \approx 2.0$, $\psi(1000, 2000) \approx 1.3$; and the following scalability metrics for the ES-CQRS application: $\psi(500, 1000) \approx 1.2$, $\psi(1000, 2000) \approx 0.00$.

7.2 Static Analysis

RQ 2 attempts to evaluate the architectural flexibility of the architectural styles CRUD and ES-CQRS. In section 2.11, several static analysis methods were established. This section presents results and visualizations for these metrics.

7.2.1 Graphs

Boxplots are used to visualize the results of static analysis. They are a standardized way of displaying the distribution of data based on a five-number summary: minimum, first quartile (25th percentile), median, third quartile (75th percentile), and maximum. The central box represents the IQR, which encompasses the middle 50% of data points. The *whiskers* extend from the edges of the box to indicate the variability outside the upper and lower quartiles, showing the full range of the data excluding extreme values. Finally, individual points plotted beyond the whiskers are outliers, representing data points that fall further from the median than the rest of the population.

7.2.2 Coupling Metrics

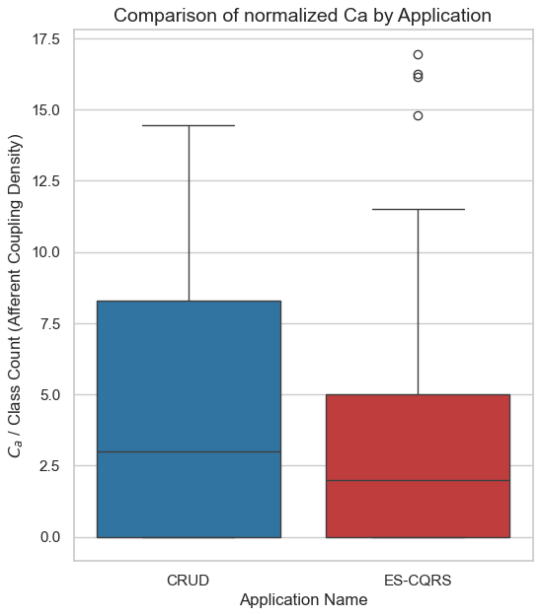
Afferent coupling (C_a, describing incoming connections) and Efferent coupling (C_e, describing outgoing connections), which were described in subsection 2.11.1, are calculated by MetricsReloaded on a package-basis. The results are visualized in a boxplot. Package size influences the value of C_a and C_e, which is why the results were normalized by class count. Therefore, the plots show this data:

$$C_a\_norm = \frac{C_a}{Class\_count} \tag{7.1}$$

$$C_e\_norm = \frac{C_e}{Class\_count} \tag{7.2}$$

Figure 7.25 presents the normalized C_a per application. The CRUD application generally has higher Afferent coupling across its packages (a 33% higher median and higher 75th percentile).

Most packages in the ES-CQRS architecture have low Afferent coupling, but some packages areas are more coupled than any package found in the CRUD app.



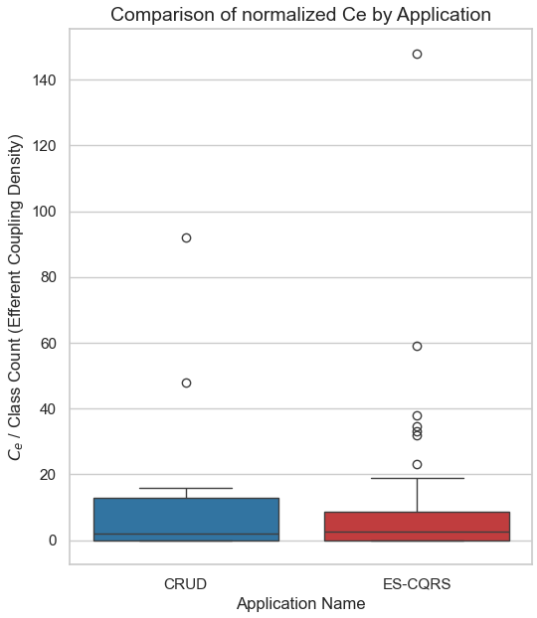
(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.0	3.0	8.3	14.4	0
ES-CQRS	0.0	0.0	2.0	5.0	17.0	4

(b) Descriptive statistics

Figure 7.25: Comparison of C_a (Afferent coupling) by Application.

Regarding Efferent coupling, the two architectures show more similarity. The medians are almost equal at values of 2.1 (CRUD) and 2.5 (ES-CQRS). CRUD's IQR is wider, ranging from 0 to around 13, while the ES-CQRS IQR ranges from 0 to around 9. However, the ES-CQRS architecture displays more outliers (7, versus 2 in CRUD) reaching a value of 148, while the maximum value of the CRUD architecture is 92.



(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.0	2.1	12.9	92.0	2
ES-CQRS	0.0	0.0	2.5	8.6	148.0	7

(b) Descriptive statistics

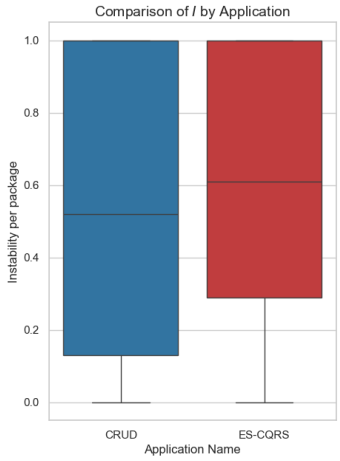
Figure 7.26: Comparison of C_e (Efferent coupling) by Application.

7.2.3 Instability and Abstractness

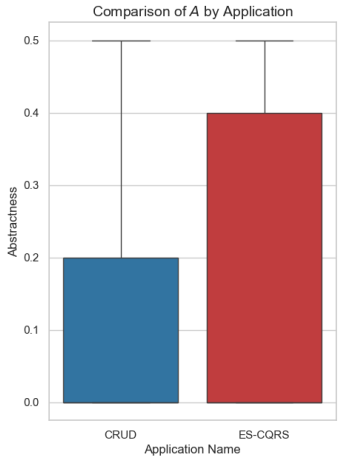
Instability I is defined as the ratio of Efferent coupling to the total coupling of a package, as shown in Equation 2.3. Therefore, it can take values between 0 and 1. It is visualized using a boxplot in Figure 7.27a. The plot highlights a wide IQR and a median value around 0.5 to 0.6 for both architectures.

Abstractness A measures the ratio of abstract classes and interfaces to the total number of classes in a package, as presented in Equation 2.4. Therefore, it can take values between 0 and 1. Its visualization in Figure 7.27b shows that the typical package in both applications has an Abstractness of 0. The IQR of the CRUD architecture reaches from 0 to 0.2; the IQR of the ES-CQRS architecture reaches from 0 to 0.4.

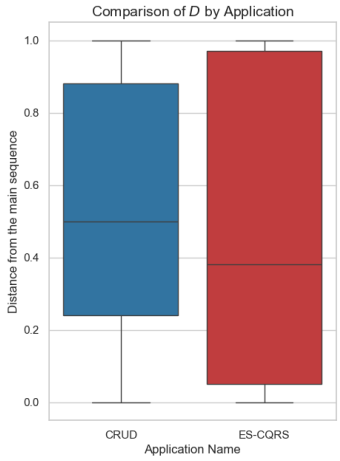
Figure 7.27d shows a scatter plot which visualizes A and I, as well as the "Main Sequence" (gray diagonal line). The concept of the "Main Sequence" was explained in subsection 2.11.2. The distribution shows two prominent large clusters of the ES-CQRS architecture located at the corners of (0, 0) and (1, 0), while the remaining smaller points are scattered primarily in the lower half of the graph below the diagonal line. Generally, it can be seen that ES-CQRS has more packages than CRUD and is more abstract. The ES-CQRS architecture exhibits a slightly lower median Distance from the Main Sequence D (Figure 7.27c) with a value of around 0.4, opposed to around 0.5 for the CRUD architecture. However, both IQRs for D have a wide spread, with the ES-CQRS architecture spanning from 0.1 to almost 1.



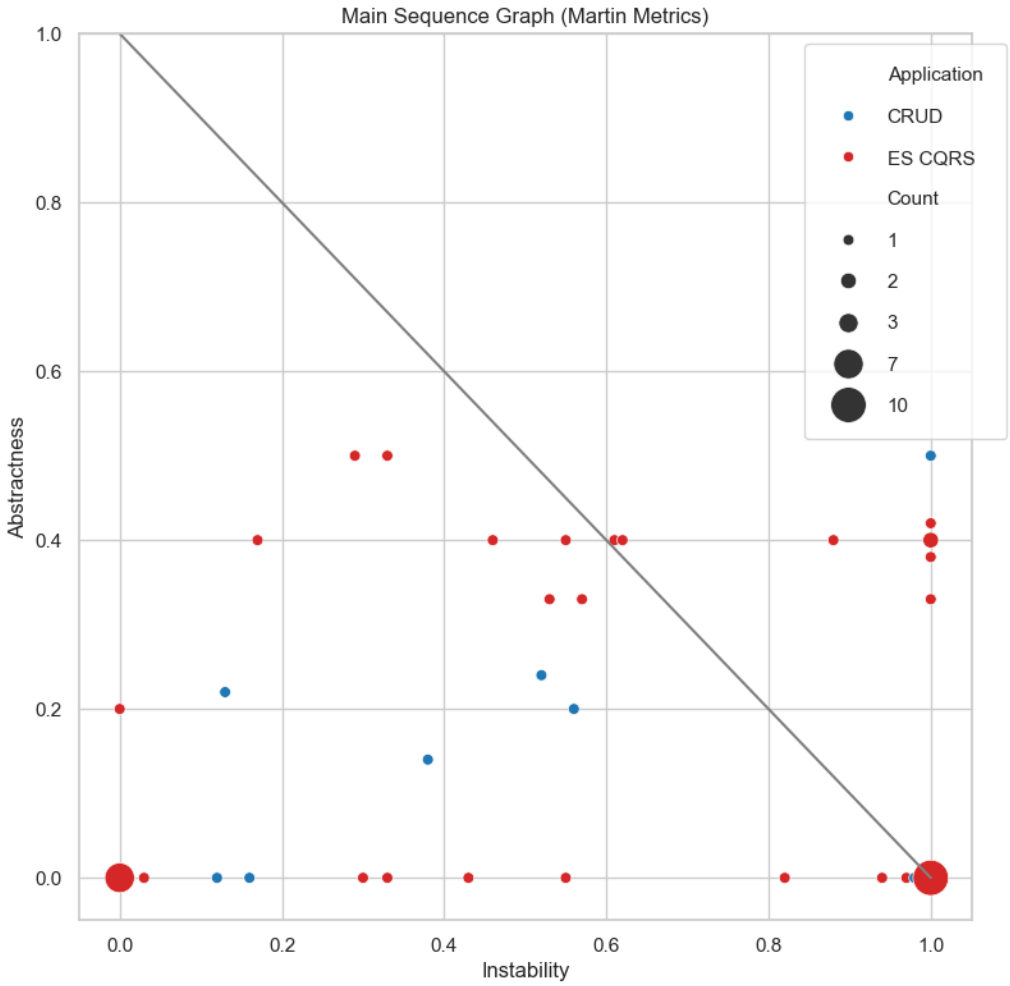
(a) Instability per package. Descriptive statistics in Table C.4.



(b) Abstractness per package. Descriptive statistics in Table C.5.



(c) Distance from the main sequence per package. Descriptive statistics in Table C.6.



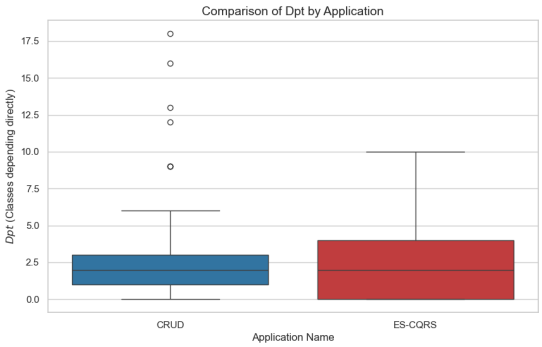
(d) Visualization of the Main Sequence

Figure 7.27: Comparison of Instability I and Abstractness A. Detailed results in section C.2.

7.2.4 Dependency Metrics

Table 6.4 outlined all dependency metrics recorded in the applications. In this section, these metrics are visualized using boxplots and descriptive statistics. Per-class results for the described dependency metrics are available in section C.3.

Figure 7.28 illustrates the distribution of *Dpt*, the number of classes depending directly on a class (*dependents*), for both architectures. Both architectures have an equal median at 2. This indicates that in both cases, a typical class has two dependents. The range between the 25th to 75th percentile, also called IQR is indicated by the shaded areas. It differs for the two architectures. The CRUD architecture shows that 50% of classes have 1 to 3 dependents, while the ES-CQRS architecture exhibits higher variability. Its IQR spans from 0 to 4. Notably, since the 25th percentile aligns with the minimum value of 0, at least 75% of the classes in this architecture have 4 or fewer dependents.



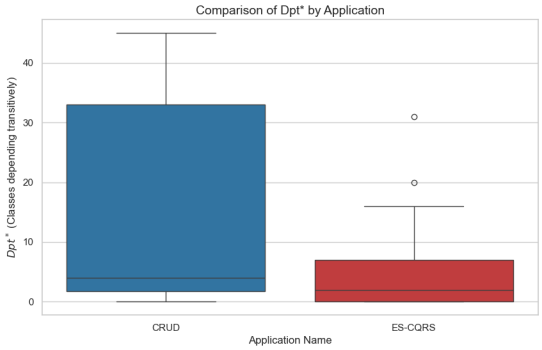
(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	1.0	2.0	3.0	18.0	7
ES-CQRS	0.0	0.0	2.0	4.0	10.0	0

(b) Descriptive statistics

Figure 7.28: Comparison of *Dpt* (direct dependants) by Application.

*Dpt**, the transitive dependent count, is presented in Figure 7.29. Compared to *Dpt*, a larger difference between the architectures is visible. While the median values of both applications are still similar, with values between 2 and 4, the CRUD architecture's IQR has a much wider range than the ES-CQRS architecture. 50% of classes in the CRUD architecture have between 2 and 33 transitive dependents, while at least 75% of the ES-CQRS architecture's classes have between 0 and 7 transitive dependents.



(a) Boxplot distribution

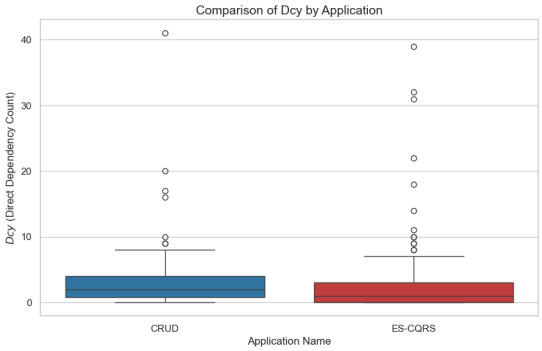
Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	1.8	4.0	33.0	45.0	0
ES-CQRS	0.0	0.0	2.0	7.0	31.0	2

(b) Descriptive statistics

Figure 7.29: Comparison of *Dpt** (transitive dependants) by Application.

Figure 7.30 illustrates the distribution of *Dcy*, a metric representing the number of classes a

given class directly depends on (*dependencies*). Similar to the results for *Dpt*, the direct dependencies across both architectures exhibit similar values. CRUD and ES-CQRS architecture exhibit a median *Dcy* of 2, respectively 1, with both IQRs situated between 0 and 4. Both architectures feature several outliers, with individual classes reaching direct dependency counts of approximately 40.



(a) Boxplot distribution

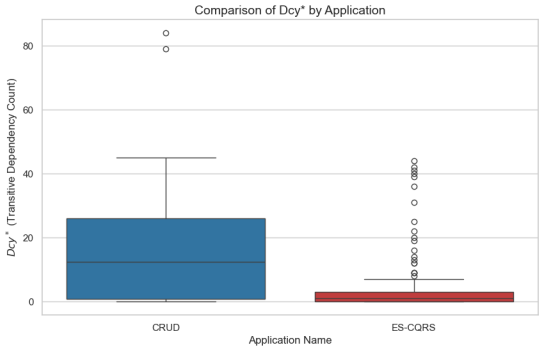
Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.8	2.0	4.0	41.0	7
ES-CQRS	0.0	0.0	1.0	3.0	39.0	13

(b) Descriptive statistics

Figure 7.30: Comparison of *Dcy* (direct dependencies) by Application.

A more pronounced divergence is visible in the transitive dependencies, *Dcy**, presented in Figure 7.31. In the CRUD architecture, this distribution shows a much wider range, with the IQR sitting between 1 and 26 transitive dependencies.

The ES-CQRS architecture exhibits a more concentrated distribution. Its IQR is narrower, with at least 75% of classes having between 0 and 3 transitive dependencies. While there are numerous outliers reaching more than 40 transitive dependencies, the primary distribution remains lower than that of the CRUD application.



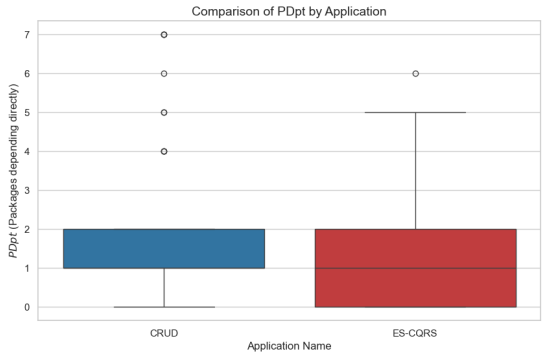
(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.8	12.5	26.0	84.0	2
ES-CQRS	0.0	0.0	1.0	3.0	44.0	20

(b) Descriptive statistics

Figure 7.31: Comparison of *Dcy** (transitive dependencies) by Application.

PDpt, presented in Figure 7.32, is a metric describing the number of packages transitively depending on a class. Both architectures have a median value of 1 and a P75 of 2, indicating that at least 75% of classes have less than 2 transitive package dependents.



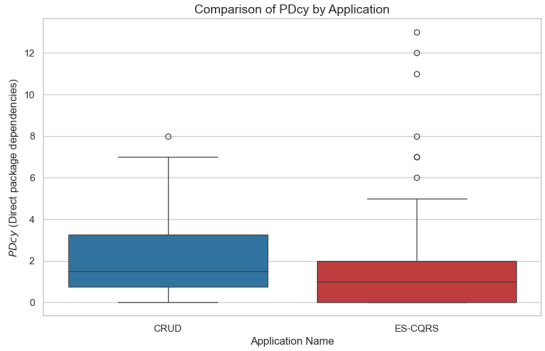
(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	1.0	1.0	2.0	7.0	9
ES-CQRS	0.0	0.0	1.0	2.0	6.0	1

(b) Descriptive statistics

Figure 7.32: Comparison of *PDpt* (transitive package dependents) by Application.

The *PDcy* metric describes the number of packages a class transitively depends on. It is presented in Figure 7.33. Again, both architectures show similar values with a median of 1.5 for the CRUD architecture and 1 for ES-CQRS. The IQR ranges from ≈ 1 to ≈ 3 for CRUD, and from 0 to 2 for ES-CQRS. The ES-CQRS architecture exhibits more outliers at 7 with a maximum value of 13, while the CRUD application shows one outlier depending on 8 packages.



(a) Boxplot distribution

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.8	1.5	3.2	8.0	1
ES-CQRS	0.0	0.0	1.0	2.0	13.0	7

(b) Descriptive statistics

Figure 7.33: Comparison of *PDcy* (transitive package dependencies) by Application.

7.2.5 MOOD Metrics

Results of the Metrics for Object-Oriented Design (MOOD) suite, outlined in subsection 2.11.3, are presented in Figure 7.34. AHF is the highest value for both architectures. Both architectures sit at around 95%.

The CRUD architecture exhibits a MIF of around 16%, higher than the ES-CQRS architecture at 1%. With a value of 45%, the ES-CQRS architecture's MHF is higher than the 29% for the CRUD architecture.

The CRUD architecture's CF sits at around 12%, while the ES-CQRS's architecture's CF has a value of around 3%.

The AIF of the CRUD architecture is at around 28%, the ES-CQRS application reaches 8%.

Generally, the CRUD Architecture covers a larger total surface area on the diagram, specifically showing higher values on the AIF and CF axes compared to the ES-CQRS Architecture.

It is worth noting that the PF of both architectures is not present in the diagram. This is due to the fact that the values exceed 100%, with the CRUD architecture having a PF of 360%,

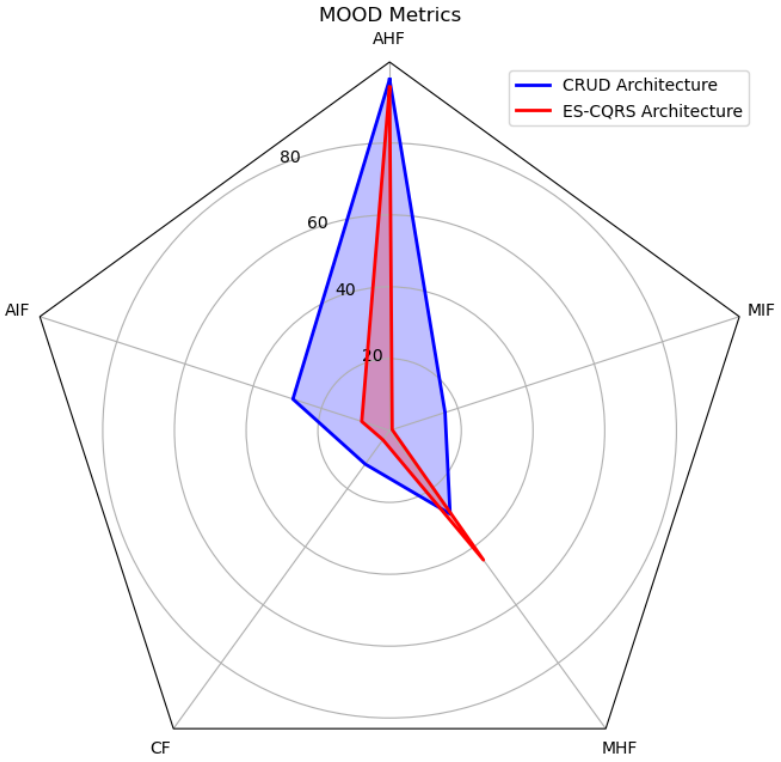


Figure 7.34: MOOD metrics presented in a spider diagram. Results in C.11

and the ES-CQRS architecture having a PF of 185%. The PF calculates the ratio of polymorphic situations to the maximum possible number of polymorphic situations, but the static analysis tool used does not take classes from libraries or external modules into consideration when computing the total possible number, which is why the values exceed 100%.

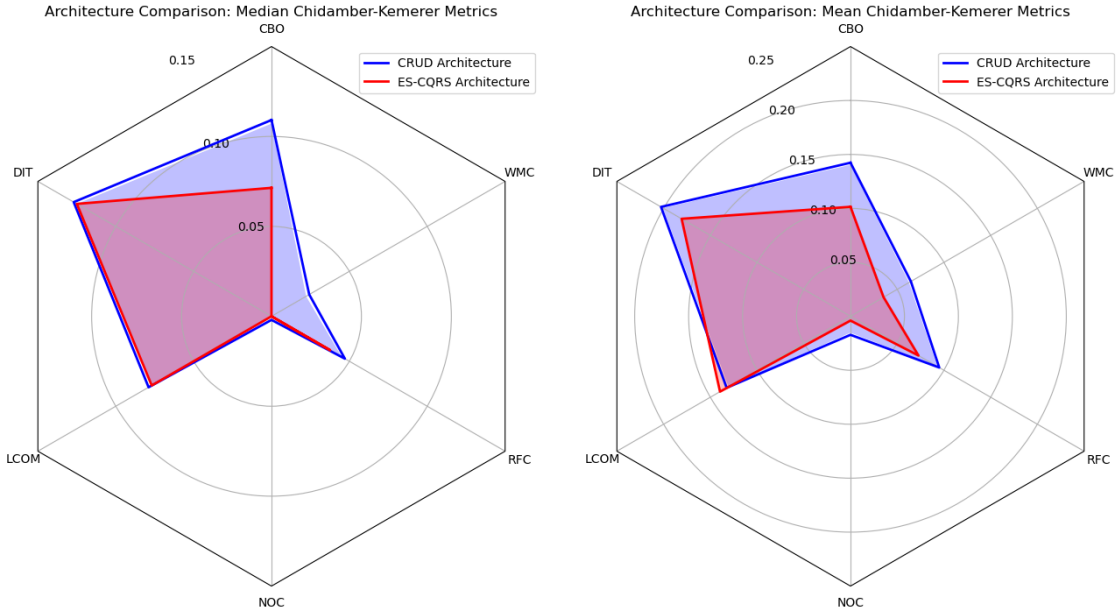
7.2.6 Chidamber Kemerer Metrics

In subsection 2.11.4, the Chidamber and Kemerer Metrics suite was described. Its results are presented in Figure 7.35 using a spider diagram. As the metrics are calculated on a per-class basis, the values were normalized and aggregated using a median and a mean for visualization purposes.

Figure 7.35a shows the median values of the metrics. The CRUD architecture's plot (blue) forms the larger shape. It reaches the highest point on the CBO axis, with a value around 0.1, and the DIT axis with a value around 0.13. It also shows a distinct outward point on the RFC axis.

The ES-CQRS architecture's plot forms a smaller shape nested mostly inside the blue area. It shows lower values than the CRUD architecture on the CBO (0.07), WMC (0), and RFC ($<$ 0.05) axes. It sits close to the CRUD architecture on the LCOM and DIT axes. While the median WMC of ES-CQRS sits at a value of 0, the mean (Figure 7.35b) shows a value of about 0.02.

Both architectures exhibit a median NOC of 0. However, Figure 7.35b reveals that some polymorphism exists in the CRUD application.



(a) Median CK-Metrics

(b) Mean CK Metrics

Figure 7.35: CK-Metrics presented in spider diagrams. Detailed results in C.5

Chapter 8

Discussion

8.1 Interpretation of Results

TODO: explain this section. First, bottlenecks and anomalies are described, then interpret results per research question. Finally give "recommendations" — when may which architecture better suited?

8.1.1 Identifying Bottlenecks

In general, the bottlenecks of either application lie in threadpool saturation or database contention. These two metrics are intrinsically linked, with database connection contention acting as a back-pressure mechanism that blocks threads from being freed up. Each request is processed in a thread ($tomcat\_threads$), and most requests make at least one database round-trip. When the database is heavily used ($hikari\_connections$ close to 10), each thread has to wait longer to get access to a connection. This in turn fills up the threadpool. As soon as the threadpool is saturated ($tomcat\_threads = 200$), any further requests are queued before they can be processed. This can be seen in most load tests, as the two metrics often reach their ceiling at the same load (L1, L2, L4, L5, L6, L8). As soon as either of these resources is *saturated*, meaning no headroom is left, the latencies typically increase sharply, due to the queueing which occurs before a request can be processed. Before that point, enough resources are available to concurrently handle most incoming requests.

A clear correlation exists between the Scalability Metric (ψ) and resource headroom. For example, in L1 and L2, ψ for ES-CQRS approaches 0.01 as the threadpool reaches saturation (zero headroom), indicating that the system can no longer effectively convert additional resource expenditure into performance gains.

During enrollment (L3), the bottleneck is not visible immediately from the resource consumption metrics. The ES-CQRS application shows a sharp increase in latency at 100 RPS. At this point, neither the threadpool nor the database connections are saturated. The issue here most likely stems from the subscription query which is used in the implementation. It blocks until the read-side projector updated the student's enrollment status. The point at which the latencies start increasing sharply can be assumed to be the moment in which the projector can not handle incoming events quickly enough. While enough worker threads would be available, the existing requests need to wait for a longer time until the request finishes processing.

8.1.2 Anomalies in Test Results

section 7.1.11 revealed inconsistencies in the collected resource consumption metrics for the CRUD application. Beyond 250 RPS, Prometheus could not detect any CPU usage, and the threadpool metrics exhibited a sharp decline in usage at 200 RPS and 250 RPS. As previously mentioned, these results were confirmed through repeated test executions, suggesting a consistent failure at this load level. A definitive root cause is difficult to pinpoint. One possible cause is resource contention within the shared infrastructure, as both the Spring Boot application and the PostgreSQL instance compete for CPU cycles on the same VM. The computational intensity of the complex joins required for this endpoint may saturate the database, consequently starving the SpringBoot server of CPU time. Because a system call is necessary to determine the CPU usage of a process, metrics may not have been recorded accurately (or at all) due to high resource contention. However, this does not explain why the threadpool usage increases again at even higher loads. Ultimately, this shared-resource setup limits both performance of the applications and the effectiveness of the testing setup, making it difficult to identify the reason for such anomalies.

Furthermore, it is worth noting that in some cases, the metrics for data store size showed an unexpected confidence interval (CI) in the ES-CQRS application. Several read-only tests show a non-zero CI, which is surprising as the seed data is identical for each test and no further data is written to the system during load generation. This observation can be made in L4, L5, L6, L8. Identifying a reason for this effect would require further investigation.

8.1.3 Projection Lags

Projection lag is the delay between the moment an event is published by the write-side and the moment that change is reflected in the read-side (projections). In a CQRS architecture, the read-side is updated asynchronously by event processors (projectors). When the system is under high load, the event processor may become a bottleneck, when it is unable to keep up with the new events being published. This results in eventual consistency, where a user might successfully save data but receive an "old" or "not found" response when querying the read-side immediately afterward, as the background synchronization has not yet caught up.

Several tests showed a decrease in storage size for the ES-CQRS application under increased load, which was attributed to projection lag. The projection lag is a very likely explanation, however no metrics were taken during the load tests which can definitely confirm this as the root cause. Furthermore, even projection lag does not fully explain why the storage taken at higher load is *lower* than at a previous load. One explanation for this could be that the higher load on the system takes up more system resources to handle the incoming requests, taking away CPU cycles from the event processor. However, this is only a theoretical conclusion that would require detailed profiling to confirm.

8.1.4 Performance and Scalability

The results obtained during load testing reveal substantial tradeoffs between the two architectural patterns. The CRUD application shows better write performance in both throughput and resource efficiency, while the ES-CQRS application offers specialized advantages for complex read queries at the cost of higher overhead. In most write-heavy scenarios like L1, L2, and L3, the CRUD system maintains significantly lower latency and consumes fewer resources. In contrast, the ES-CQRS system frequently hits resource saturation — reaching 200 threads and 10 database connections — and violates latency thresholds as load increases. This divergence is especially visible in the

enrollment test (L3), where the CRUD application manages much higher request volumes before performance degrades. Across nearly all tests, the ES-CQRS architecture requires more CPU and storage, often using double the disk space due to the overhead of storing both events and multiple denormalized projections.

These differences are reflected in the calculated scalability metric ψ. While the CRUD application often exhibits values above 1.0, indicating efficient scaling where throughput increases outpace cost, the ES-CQRS application frequently shows values near zero under high load (e.g., $\psi \approx 0.01$ in L1). This metric serves as a useful tool for evaluating "productivity" by balancing throughput against resource costs like CPU and thread saturation. However, its usability is sensitive to how the "cost" function is weighted. For instance, the inclusion of a high penalty for projection lag (weighted at 3.0 in the model) heavily penalizes the ES-CQRS application's scalability score when consistency thresholds are missed.

However, the ES-CQRS application demonstrates its strength in specific read-heavy scenarios such as retrieving student credits (L6), where it provides a significant speedup compared to the CRUD application. While the CRUD application struggles with complex aggregations at high loads, reaching a scalability of $\psi \approx 0.00$ as its threadpool saturates, ES-CQRS benefits from pre-calculated read models that keep latencies low even at 3000 RPS. A similar advantage can be observed in L5 (reading a list of all lectures), because CQRS is specifically designed to optimize these types of data retrievals. In these read-intensive contexts, the ES-CQRS application maintains a higher scalability factor (e.g., $\psi \approx 1.1$ at 2000 RPS) because the low latency and lack of complex DB joins compensate for its higher idle resource usage.

A critical drawback of Event Sourcing and CQRS is the projection lag observed in the freshness test (L7), where the application fails to reflect new writes on the read side within the required 100ms window (SLO 2) under heavy load. This eventual consistency means that even if reads have low latencies, they are not guaranteed to be up-to-date. Represented by the *read_visible_rate* metric, this *freshness* drops as low as 0.02 for ES-CQRS at 400 IPS. This highlights a tradeoff for systems that require immediate data freshness: the mechanism that enables fast and scalable reads in ES-CQRS can also lead to stale data being served when the system is under pressure.

It should be noted that the results of L8 — the historic reconstruction load test presented in section 7.1.13 — yielded a rather surprising result in terms of active database connections. Despite only using the database for two indexed ID lookups before streaming events from Axon's Event Store, the ES-CQRS application saturated the connection pool with a median of 10 active connections. This behavior is counter-intuitive compared to the CRUD implementation, which consistently maintains fewer active connections even though it performs more complex data fetching directly from the relational database through Envers.

Beyond the specific resource metrics, the ES-CQRS implementation adds an inherent architectural overhead due to the abstraction layers provided by Axon Framework and Axon Server. While these tools offer benefits like location transparency and automated message routing, they introduce a more complex data flow compared to the direct function calls inside a standard Layered Architecture. Theoretically, the process of wrapping, routing, and handling command messages and events through the framework adds a latency penalty and increases CPU demand. However, as no profiling was conducted to isolate these specific costs during program execution, this point remains theoretical.

8.1.5 Architectural Flexibility

The static analysis results reveal distinct characteristics of the CRUD and ES-CQRS implementations. While metric suites like MOOD and Chidamber and Kemerer Metrics provide a broad overview of code quality, the most insightful data for evaluating architectural flexibility results from the coupling and dependency metrics. These metrics provide a direct representation of how components interact and the extent to which a change in one area of the system propagates through others.

Evaluation of Metric Utility

The analysis indicates that transitive dependency metrics (Dpt^* and Dcy^*), described in subsection 7.2.4, are more useful than their direct counterparts (Dpt and Dcy) for assessing the quality and coupling of an architecture. While the median values for direct dependencies are similar across both architectures, the ES-CQRS approach shows a significantly narrower IQR for transitive dependencies. In the CRUD application, 50% of classes have between 2 and 33 transitive dependents, whereas at least 75% of ES-CQRS classes have 7 or fewer. This suggests that the ES-CQRS architecture more effectively isolates components, preventing a "rippling effect" of changes common in highly coupled architectures.

In contrast, the Distance from the Main Sequence (D), composed of Abstractness (A) and Instability (I), appears to be less accurate in this specific context. It is especially worth noting that a high Abstractness score does not inherently equate to a high level of *functional abstraction*. For instance, the ES-CQRS architecture utilizes many JPA repositories on its Query side. These components are technically interfaces. This inflates the Abstractness score without necessarily having an architectural impact. Furthermore, these metrics are highly sensitive to the chosen package layout rather than the logic itself. A prime example is the `api` package in the ES-CQRS application. It serves as a vital decoupling point between the Command and Query side, but because the classes are not abstract and highly depended upon, they appear "poor" according to traditional instability metrics despite their arguably high architectural value. Similarly, colocating Controller, Service and Repository classes in the same package in the CRUD application results in improved Instability values, even though the dependencies between classes are still present in the same way.

Architectural Impact on Evolution and Scalability

The structural differences between the two approaches have an impact regarding long-term flexibility and evolution. The CRUD architecture displays higher CBO, higher coupling (C_a, C_e) and more dependencies (Dcy^*, Dpt^*), and a larger "surface area" in the MOOD metrics. This increased coupling implies that as the codebase grows, the complexity of making changes increases non-linearly, as each class is transitively linked to a larger portion of the system.

In contrast, the ES-CQRS architecture demonstrates a structural advantage for scalability and evolution. The primary difference lies in the separation of concerns between writes and reads. By decoupling the Command and Query sides, the architecture maintains lower transitive coupling across the majority of the system.

Consequently, the ES-CQRS approach likely provides a more seamless transition to horizontal scaling. Because the dependencies are already logically and physically partitioned (shown by the lower Dcy^* and $PDcy$ values), the effort required to split the monolith into independent

microservices is significantly reduced. This allows for specific parts of the system, such as high-traffic read models, to be scaled horizontally on separate infrastructure without requiring the entire application to be replicated. Additionally, Axon Framework provides location transparency, reducing the need to write additional boilerplate code when attempting to split the system into separate services. Therefore, while the ES-CQRS architecture may have more outliers in package coupling, its fundamental structure provides the necessary isolation for the long-term scalability and independent evolution of system components.

It can be noted that the ES-CQRS pattern introduces unique challenges regarding developer experience and code clarity. The indirection introduced by Axon's message buses leads to transparency and decoupling, which is reflected by the dependency and coupling metrics. However, this makes the execution flow less explicit than the direct method calls found in a standard Layered Architecture. The application logic becomes more difficult to follow through static analysis alone. This fragmented data flow may increase the cognitive load on developers and make debugging harder as stack traces become less explicit.

8.1.6 Traceability

The findings presented in this section are primarily derived from a synthesis of existing literature regarding system architecture and data integrity previously presented in section 3.3, alongside empirical performance data collected in L8 (section 7.1.13).

The comparison between CRUD and ES-CQRS systems reveals a trade-off between the reliability of historical data and the speed of accessing it. In traditional CRUD systems, the audit log acts as a secondary observer. This architecture is susceptible to the "dual-write" problem, where a database update succeeds but the log entry fails. As noted by Gantz [43, p. 155], the reliability of an audit depends on the system's ability to produce accurate evidence. If the audit trail diverges from the application state, it fails to meet the criteria for strict legal compliance. In contrast, Event-sourced systems address these accuracy concerns by adhering to the principle that the event log *is* the truth [7]. Because every state change is recorded as an immutable event, the system preserves the exact domain context and "user intent" [6, p. 531]. This makes the reconstruction process deterministic and eliminates the risk of silent data divergence. Even if read-side projections become desynchronized, they can be corrected and rebuilt from the log. Kleppmann [6] suggests that this process additionally makes diagnosing unexpected behaviors significantly easier through event replay.

Furthermore, CRUD logs often capture *what* changed without preserving the *why*. As Kleppmann [6, p. 531] argues, the application logic in CRUD is transient, meaning business context is often lost over time. Capturing and reconstructing intent is possible in CRUD systems. It was also implemented in this study, though it requires additional effort to attach the context to changes. In contrast, events in ES carry inherent metadata which contain intent.

However, the efficiency of this reconstruction remains a challenge when using Event Sourcing. While Monagari [38] cites a dramatic reduction in incident resolution time for financial institutions using ES, this study's load testing results (L8) provide contrasting empirical results. The results show that CRUD architectures are significantly faster in reconstructing historic state, at least for simple history queries. Reconstructing a student's grade history, for example, proved more efficient in the CRUD model.

Ultimately, while ES-CQRS provides the "superior evidence" required for forensic or legal auditing [23, p. 17], the CRUD approach remains more performant for frequent, high-volume historical lookups. As historical state lookups are likely uncommon in typical applications, the

choice of architecture should be guided by whether the priority lies in the speed of retrieval or the absolute integrity of the audit trail.

It should also be mentioned that frequent lookups of historic state are likely uncommon in typical applications. If a high volume of time-travel queries was a real use-case in an application, the architects may want to adopt an entirely different approach in storing current and historic state, such as time-series databases.

8.1.7 Architectural Trade-offs and Recommendations (TODO)

Abcdefg.

8.2 Limitations of the Study

This research provides an empirical, quantifiable comparison between CRUD and ES-CQRS architectures. However, several technical and methodological limitations must be acknowledged when interpreting the results. One of these limitations is the infrastructure environment used for testing. Because both the load generation and the server-side components were hosted on VMs running on the same physical machine, the observed network latency does not accurately reflect a distributed production environment. In a real-world scenario, services typically communicate across a physical network rather than on a single host. Particularly the ES-CQRS architecture involves a more complex chain of communication, including the command and query bus, aggregate handling, projections and event storage. In a distributed environment, the cumulative network round trips required for these steps would likely result in higher latencies than those recorded in this study's test setup.

Except for Axon Server's storage size, the server-side metrics were only collected from the SpringBoot application. Because the VM also hosted PostgreSQL and Axon Server, the collected CPU metric ($process\_cpu\_usage$) does not capture the total system load. It is likely that the overall system CPU was fully saturated even when the server's CPU usage suggested remaining overhead. This is a threat to the validity of the performance data.

The accuracy of the benchmarking results was further affected by the testing configuration. In some load tests, a significant number of iterations was dropped at high RPS because the maximum number of VUs in the k6 configuration was insufficient for the load. This means the results do not reflect the real performance under the respective load, but instead the load of $RPS - dropped\_iterations\_rate$.

Additionally, a more granular performance analysis would require measuring garbage collection pauses, memory allocation rates, database time, and other system metrics. Furthermore, without profiling, the "root causes" of specific bottlenecks remain theoretical, as the collected data shows the *results* of system stress rather than a detailed breakdown of internal execution delays. The performance analysis is also limited because the tests only varied the number of requests per second. Other factors, such as the size of the data being sent in POST requests or the volume of data fetched in queries, remained constant per test and were not evaluated.

Next, the server-side metrics collected through Actuator and Prometheus may have a reduced accuracy under very high load because the metric collection in itself is bound to system resources and may start slowing down under high load. Also, the actuator endpoints themselves experience the same queueing delay as other requests.

Furthermore, although both applications were developed following industry standards and

common best practices, it cannot be formally guaranteed that the implementations are entirely free of defects or suboptimal coding practices. Consequently, the observed performance metrics may be influenced by unidentified implementation errors rather than being representative of the underlying architecture.

One such implementation error which likely influenced the load testing measurements is a specific oversight in the ES-CQRS implementation regarding resource contention. In the current setup, both the synchronous lookup projectors — which belong solely to the write-side and are used to ensure immediate consistency — and the asynchronous query-side projections share the same underlying database instance and connection pool. This creates a bottleneck where the lookup projectors, which should ideally be very low-latency, must compete for database connections and I/O cycles with the query-side event processors. This shared resource dependency undermines the theoretical decoupling of the command and query sides, as heavy query-side projection activity directly introduces latency into the command-handling pipeline.

Beyond the technical measurements, the given architectural evaluation is limited as it is simply a synthesis of static analysis. This study did not investigate the topic of schema evolution, which is the process of managing how data structures, such as events or database tables, change over time. In a long-running production system, the difficulty of evolving an event store's schema is an additional factor in the total cost of ownership and flexibility of an ES-CQRS system. Similarly, managing complex database migrations in coupled CRUD architectures without data-loss may introduce additional challenges.

Because the use case and the load patterns used in this study were artificially generated, they may not capture the unpredictable nature of real-world user behavior or specific business requirements.

While a load test was developed and executed for a simple history query (L8), the potential use-cases for time-travel queries are far wider. More complex historical queries involving several entities would have been a different challenge, and could have shown different results for the two architectures, both in implementation complexity and latencies.

Finally, the emphasis placed on certain architectural benefits, such as traceability or scalability, is inherently subjective. Different organizations or developers might prioritize SLOs or compliance requirements differently, which would alter the perceived value of one architecture over the other.

8.3 Answering the Research Questions

This section will provide a conclusive answer to the three sub-research questions, before providing a holistic answer to the main research question.

8.3.1 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?*

The CRUD architecture generally outperforms ES-CQRS during writes, maintaining lower latency and higher resource efficiency. The CRUD system has minimal architectural overhead, which could be one reason for this observation. Further, while writes in ES-CQRS *technically* run on an append-only event stream, which should be very fast in theory, realistic business requirements, such as the ones chosen for this study, still require synchronous writing to lookup tables, resulting in longer blocking times during command handling. While CRUD systems scale efficiently until

reaching threadpool or database connection limits, ES-CQRS frequently hits these saturation points sooner.

However, ES-CQRS demonstrates improved scalability for intensive read operations, as its pre-calculated, denormalized projections allow it to handle high request volumes that cause the CRUD implementation to fail. Generally, it can be observed that the more intensive a read-scenario is, the higher are the performance benefits of the ES-CQRS implementation.

In summary, ES-CQRS offers a specialized performance trade-off: it provides high-speed data retrieval at the cost of significantly higher CPU, storage, and a risk of stale data due to projection lag. It should be noted that the business requirement of immediately consistent writes most likely substantially decreased the ES-CQRS implementation's performance, as the synchronous lookup tables added substantial overhead.

In conclusion, this study demonstrates that architectural choices dictates a system's performance and scalability limits. CRUD remains resource efficient and performant under write-heavy scenarios. Conversely, ES-CQRS effectively "pre-pays" for performance through higher storage taken up by projections, enabling a substantially higher scalability ceiling for intensive read operations. These findings suggest that the value of ES-CQRS is not universal, but rather highly dependent on a domain's specific ratio of reads to writes and its tolerance for eventual consistency.

8.3.2 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?*

The fundamental structural difference between the two approaches lies in the degree of component isolation, with the ES-CQRS architecture maintaining lower transitive coupling than the CRUD model. The CRUD application suffers from high dependency counts, which could cause "ripple effects" where changes propagate across the system. This confirms the concerns raised by Singh et al. [39] regarding CBO as a primary detractor of quality, as the CRUD model's higher coupling creates a non-linear increase in change complexity. In contrast, ES-CQRS effectively separates read and write concerns, leading to a reduced dependency count and high separation of concerns. This structural decoupling suggests enhanced long-term flexibility by allowing for independent evolution of features and modules, and a more seamless transition towards horizontal scaling or microservices. Conversely, the CRUD application's higher coupling suggests that as the codebase grows, the complexity of making changes will increase non-linearly. Ultimately, the architectural isolation in ES-CQRS provides a more robust foundation for system scalability and maintenance.

Schema evolution is a topic that was not evaluated in the thesis, but it plays a substantial role in how a system can evolve over time. Literature, such as [42], suggest that schema evolution is challenging, particularly in ES-CQRS systems.

8.3.3 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?*

CRUD systems offer higher performance for high-volume historical lookups. However, they are susceptible to the "dual-write" problem, which can lead to a divergence between the application

state and the audit trail. This divergence undermines the reliability of the system's "evidence," which Gantz [43, p. 155] identifies as the critical factor for successful IT auditing and compliance.

In contrast, Event Sourcing guarantees data integrity by treating the immutable event log as the singular "truth" of the system [7], capturing both system state and business intent. While this event-driven approach facilitates deterministic reconstruction for forensic auditing, empirical results from this study (L8) indicate significantly lower performance in historic reconstruction compared to traditional audit logs stored in SQL tables.

Ultimately, the CRUD application provides higher efficiency for simple historic state retrieval, whereas ES-CQRS offers the "superior evidence" [23, p. 17] required for rigorous legal requirements. Furthermore, as suggested by [6, p. 531], [38], the event log simplifies bug reproduction and incident diagnosis by allowing developers to replay exact system states. Therefore, the choice between these architectures requires developers to make a trade-off between the required speed of access and the guaranteed accuracy of the historical record.

8.3.4 Conclusion

> *How does an Event Sourcing architecture compare to CRUD systems with an independent audit log regarding performance, scalability, flexibility and traceability?*

While the CRUD implementation demonstrates superior write performance and lower CPU and storage overhead, it suffers from higher transitive coupling that may hinder the system's architectural evolution. In contrast, the combination of ES and CQRS provides improved latencies for read-heavy scenarios and appears to be more flexible in terms of architectural evolution. However, it introduces additional challenges such as eventual consistency, added infrastructure complexity and increased storage requirements.

Regarding traceability, the CRUD application offers faster retrieval of historical states but remains vulnerable to data divergence, whereas Event Sourcing ensures data integrity by treating the event log as the immutable source of truth.

Ultimately, the choice between these architectures should be driven by the specific domain requirements: CRUD is better suited for write-heavy applications with simple data relations, while ES-CQRS excels in complex, read-intensive environments, and domains requiring valuing high traceability. Implementing Event Sourcing and CQRS can therefore be seen as a strategic investment in flexibility and data reliability at the potential expense of raw write throughput.

8.4 Optimizations and Further Work

This section outlines potential technical refinements for the existing implementations and identifies opportunities for future work. While the current results establish a baseline for both architectures, several optimization strategies could be applied to reduce the performance bottlenecks in both applications.

Firstly, as described in section 8.2, several issues can be identified in the testing setup. To collect more meaningful results during load testing, these should be eliminated.

Next, the performance of both applications could be optimized by investigating their source code and accurately identifying bottlenecks through profiling. As previously mentioned in section 8.2, several implementation details likely influenced measurements. Improving on these aspects would be the first step towards an increased performance.

The underlying infrastructure and framework configurations offer room for improvement. While no tuning was done for this thesis on purpose to make "out-of-the-box" performance comparable, Table 8.1 summarizes parameters that could be tuned to resolve the resource saturation and latency bottlenecks observed in the ES-CQRS and CRUD applications. Before doing any tuning, further profiling would be necessary to identify bottlenecks in function calls or queries.

Keyword	Action	Goal
Pool Tuning	Adjust Tomcat thread counts and database connection limits.	Reduce queueing delays.
Event Store	Tune Axon page sizes and set different snapshot thresholds.	Reduce the number of expensive I/O operations when reading long event streams.
Reactivity	Use reactive programming paradigms by implementing Spring WebFlux[1] and reactive JPA repositories[2].	Free up worker threads while waiting for database or network responses.
Query Refinement	Replace auto-generated Hibernate queries with custom JPQL queries, optimize further using `@BatchSize` and EAGER fetching options.	Eliminate inefficient queries and reduce the total number of database round-trips to minimize database time
Serialization	Tune Jackson's serialization options	Minimize CPU overhead.
Projections	Transition from JSON-based projections to flat, denormalized SQL-native tables, or attempt to replace JSON by binary formats like Protobuf[3] or Kryo[4]	Eliminate serialization overhead, reduce projection storage size and enable high-performance JDBC mapping.

Table 8.1: Proposed System Tuning and Optimization Strategies

Additional performance gains could be achieved through the implementation of caching. Introducing a cache for frequently accessed student data could provide the CRUD application with read speeds comparable to CQRS without the structural complexity of Event Sourcing, though this introduces its own challenges regarding cache invalidation.

The need for synchronous lookup tables to enable consistent writes in ES-CQRS likely added substantial overhead, drastically reducing write performance. This need emerged from the non-functional requirements defined for this study. However, in high-traffic environments where immediate consistency during writes may not be valued as highly, the system could also implement eventual consistency on the write-side, avoiding the overhead of global set-based validation. Instead, the read-side could be queried as a heuristic before issuing a Command, and process managers (Sagas) could be applied to later correct any illegal states. This would likely increase write performance, while taking the risk of temporary inconsistencies. Therefore, future research should explore how different domain requirements influence technical performance, specifically evaluat-

[1] `https://docs.spring.io/spring-framework/reference/web/webflux.html`

[2] `https://docs.spring.io/spring-data/jpa/reference/data-commons/api/java/org/springframework/data/repository/reactive/ReactiveCrudRepository.html`

[3] `https://protobuf.dev/`

[4] `https://github.com/EsotericSoftware/kryo`

ing how the prioritization of immediate versus eventual consistency changes the scalability and resource efficiency of each architectural style.

Furthermore, as this study evaluated both applications on a single machine, the assessment of scalability and architectural flexibility remains primarily theoretical. To provide empirical evidence for these claims, a follow-up study would be appropriate, which physically transitions the systems into microservices and performs horizontal scaling. By implementing these structural changes, the practical performance limits and the true decoupling of the ES-CQRS pattern could be validated further. Additionally, the effort required to transform the monolithic structure into microservices would serve as a practical benchmark to validate the accuracy of the static analysis results.

Another point that could be evaluated in future work is schema evolution. This topic was only briefly mentioned in this study, but an empirical comparison of schema evolution methods in both CRUD and ES-CQRS systems could be worth researching.

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] 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`

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

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

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

[8] 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/`

[9] 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`

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

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

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

[13] 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

[14] 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

[15] 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

[16] 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

[17] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, & B. B. Welch, "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

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

[19] 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

[20] 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

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

[22] R. Malyi & P. Serdyuk, "Developing a Performance Evaluation Benchmark for Event Sourcing Databases," en, *Vìsnik Nacìonal'nogo unìversitetu "L'vìvs'ka politehnìka". 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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.

[29] 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

[30] 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.

[31] 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.

[32] F. Brito e Abreu & R. Carapuça, "Object-Oriented Software Engineering: Measuring and Controlling the Development Process," en, *Proceedings of "4th Int. Conf. on Software Quality"*, no. Revised Version, 1994.

[33] 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

[34] 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/

[35] 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.

[36] 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/

[37] 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

[38] 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

[39] 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

[40] 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/

[41] 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/

[42] 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

[43] 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/

[44] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," *Doctor Dobbs Journal*, Jan. 2008.

[45] G. Young, *Versioning in an Event Sourced System*, en. Leanpub, May 2017. Accessed: Feb. 22, 2026. [Online]. Available: https://leanpub.com/read/esversioning/

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

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

[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, *Aggregates*, 2025. Accessed: Jan. 23, 2026. [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.12/axon-framework-commands/modeling/aggregate/

[67] 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/

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

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

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

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

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

[77] 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

[78] Broadcom, Inc., *Endpoints :: Spring Boot*, 2026. Accessed: Feb. 28, 2026. [Online]. Available: https://docs.spring.io/spring-boot/reference/actuator/endpoints.html

[79] Broadcom, Inc., *Enabling Production-ready Features :: Spring Boot*, 2026. Accessed: Feb. 28, 2026. [Online]. Available: https://docs.spring.io/spring-boot/reference/actuator/enabling.html

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

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

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

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

[84] 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/

[85] 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/

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

[87] 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/

[88] 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/

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

[90] M. F. Triola, *Elementary statistics*, eng, 11. ed. Boston, Mass: Pearson Education, 2012, ISBN: 978-0-321-69450-8.

[91] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling*, en. John Wiley & Sons, 1991, ISBN: 978-0-471-50336-1. Accessed: Feb. 22, 2026. [Online]. Available: https://www.researchgate.net/publication/259310412_The_Art_of_Computer_Systems_Performance_Analysis_Techniques_For_Experimental_Design_Measurement_Simulation_and_Modeling_NY_Wiley

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`
- `https://github.com/lukas-karsch/thesis`

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

Load Testing Results

This chapter of the appendix presents tables generated from the results during load testing. The *speedup* represents the factor by which the ES-CQRS application differed from the CRUD application. The statistical significance of the performance differences was evaluated using the *Mann-Whitney U* test. If its result is significant ($p \leq 0.05$), that means that the difference in values are unlikely to be due to noise or randomness.

The results are reported using the probability thresholds defined in Table B.1, where a lower p-value indicates a higher level of statistical significance.

Significance	p**-value**	**Interpretation**
***	$p \leq 0.001$	Highly significant
**	$p \leq 0.01$	Very significant
*	$p \leq 0.05$	Significant
n.s.	$p > 0.05$	Not significant

Table B.1: Significance thresholds

B.1 L1: Create Course Simple

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	25	4.47	0.0	7.84	0.0	1.8x Slower	***
latency_p50	25	4.18	0.0	7.03	0.0	1.7x Slower	***
latency_p95	25	5.99	0.0	12.21	0.0	2.0x Slower	***
latency_p99	25	14.73	0.0	20.49	0.0	1.4x Slower	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	50	3.42	0.0	6.3	0.0	1.8x Slower	***
latency_p50	50	3.14	0.0	5.48	0.0	1.7x Slower	***
latency_p95	50	5.25	0.0	10.69	0.0	2.0x Slower	***
latency_p99	50	6.82	0.0	14.73	0.0	2.2x Slower	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	100	2.65	0.0	4.87	0.0	1.8x Slower	***
latency_p50	100	2.39	0.0	4.18	0.0	1.7x Slower	***
latency_p95	100	4.26	0.0	9.15	0.0	2.1x Slower	***
latency_p99	100	6.1	0.0	13.77	0.0	2.3x Slower	***
dropped_iterations_rate	200	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	200	2.12	0.0	4.1	0.0	1.9x Slower	***
latency_p50	200	1.9	0.0	3.32	0.0	1.7x Slower	***
latency_p95	200	3.37	0.0	8.65	0.0	2.6x Slower	***
latency_p99	200	5.28	0.0	14.43	0.0	2.7x Slower	***
dropped_iterations_rate	500	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	500	1.96	0.0	7.35	0.0	3.7x Slower	***
latency_p50	500	1.82	0.0	3.87	0.0	2.1x Slower	***
latency_p95	500	2.67	0.0	25.33	0.0	9.5x Slower	***
latency_p99	500	5.61	0.0	43.76	0.0	7.8x Slower	***
dropped_iterations_rate	1000	0.0	0.0	51.79	1.9	*N/A*	***
latency_avg	1000	2.19	0.0	318.47	0.01	145.7x Slower	***
latency_p50	1000	1.99	0.0	6.51	0.0	3.3x Slower	***
latency_p95	1000	2.99	0.0	1595.75	0.03	534.2x Slower	***
latency_p99	1000	6.94	0.0	1894.46	0.04	272.8x Slower	***

Table B.2: Statistical comparison of latencies for POST /courses (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	3.46	0.0	6.69	0.0	1.9x Slower	***
latency_p50	25	3.26	0.0	6.1	0.0	1.9x Slower	***
latency_p95	25	5.28	0.0	11.01	0.0	2.1x Slower	***
latency_p99	25	12.95	0.0	17.63	0.0	1.4x Slower	***
latency_avg	50	2.65	0.0	5.34	0.0	2.0x Slower	***
latency_p50	50	2.45	0.0	4.87	0.0	2.0x Slower	***
latency_p95	50	4.24	0.0	9.62	0.0	2.3x Slower	***
latency_p99	50	5.58	0.0	13.25	0.0	2.4x Slower	***
latency_avg	100	2.05	0.0	4.09	0.0	2.0x Slower	***
latency_p50	100	1.88	0.0	3.47	0.0	1.8x Slower	***
latency_p95	100	3.46	0.0	8.2	0.0	2.4x Slower	***
latency_p99	100	5.15	0.0	12.33	0.0	2.4x Slower	***
latency_avg	200	1.74	0.0	3.56	0.0	2.1x Slower	***
latency_p50	200	1.62	0.0	2.9	0.0	1.8x Slower	***
latency_p95	200	2.77	0.0	7.93	0.0	2.9x Slower	***
latency_p99	200	4.67	0.0	13.07	0.0	2.8x Slower	***
latency_avg	500	1.66	0.0	6.73	0.0	4.0x Slower	***
latency_p50	500	1.57	0.0	3.52	0.0	2.2x Slower	***
latency_p95	500	2.28	0.0	23.89	0.0	10.5x Slower	***
latency_p99	500	5.14	0.0	41.59	0.0	8.1x Slower	***
latency_avg	1000	1.94	0.0	76.92	0.0	39.6x Slower	***
latency_p50	1000	1.82	0.0	6.1	0.0	3.3x Slower	***
latency_p95	1000	2.67	0.0	343.39	0.0	128.5x Slower	***
latency_p99	1000	6.3	0.0	434.82	0.0	69.0x Slower	***

Table B.3: Statistical comparison of latencies for POST /courses (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0163	0.0012	0.0404	0.0019	2.5x Higher	***
50	0.0182	0.0019	0.0506	0.0033	2.8x Higher	***
100	0.0252	0.0013	0.0818	0.0062	3.2x Higher	***
200	0.0322	0.0039	0.1368	0.0062	4.3x Higher	***
500	0.0533	0.0007	0.2894	0.0028	5.4x Higher	***
1000	0.1066	0.0013	0.4563	0.0025	4.3x Higher	***

Table B.4: Statistical comparison of *cpu_usage* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
50	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
100	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
200	10.0	0.0	10.0	0.0	1.0x Lower	***
500	11.0	0.0	38.0	3.5	3.5x Higher	***
1000	23.0	0.5	200.0	0.0	8.7x Higher	***

Table B.5: Statistical comparison of *tomcat_threads* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0	0.0	0.0	*N/A*	***
50	0.0	0.0	0.0	0.0	*N/A*	***
100	0.0	0.0	0.0	0.0	*N/A*	***
200	0.0	0.0	1.0	0.0	*N/A*	***
500	1.0	0.0	4.0	0.5	4.0x Higher	***
1000	2.0	0.0	9.0	0.5	4.5x Higher	***

Table B.6: Statistical comparison of *hikari_connections* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	9.51	0.0	18.95	0.01	2.0x Higher	***
50	11.05	0.0	21.28	0.0	1.9x Higher	***
100	14.12	0.0	25.87	0.01	1.8x Higher	***
200	20.26	0.0	35.14	0.01	1.7x Higher	***
500	38.69	0.01	60.57	0.19	1.6x Higher	***
1000	69.33	0.02	52.91	0.32	1.3x Lower	***

Table B.7: Statistical comparison of Data Store Size (MB) for run-create-course-simple, aggregated over at least 25 runs

B.2 L2: Create Course Prerequisites

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	25	5.11	0.0	8.0	0.0	1.6x Slower	***
latency_p50	25	4.71	0.0	7.09	0.0	1.5x Slower	***
latency_p95	25	7.36	0.0	12.87	0.0	1.7x Slower	***
latency_p99	25	16.32	0.0	20.78	0.0	1.3x Slower	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	50	3.96	0.0	6.44	0.0	1.6x Slower	***
latency_p50	50	3.59	0.0	5.54	0.0	1.5x Slower	***
latency_p95	50	6.35	0.0	11.47	0.0	1.8x Slower	***
latency_p99	50	8.49	0.0	15.93	0.0	1.9x Slower	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	100	3.1	0.0	5.02	0.0	1.6x Slower	***
latency_p50	100	2.8	0.0	4.26	0.0	1.5x Slower	***
latency_p95	100	5.29	0.0	9.89	0.0	1.9x Slower	***
latency_p99	100	7.48	0.0	14.98	0.0	2.0x Slower	***
dropped_iterations_rate	200	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	200	2.54	0.0	4.33	0.0	1.7x Slower	***
latency_p50	200	2.3	0.0	3.41	0.0	1.5x Slower	***
latency_p95	200	4.15	0.0	9.83	0.0	2.4x Slower	***
latency_p99	200	6.45	0.0	16.18	0.0	2.5x Slower	***
dropped_iterations_rate	500	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	500	2.38	0.0	10.47	0.0	4.4x Slower	***
latency_p50	500	2.12	0.0	4.16	0.0	2.0x Slower	***
latency_p95	500	3.71	0.0	38.46	0.0	10.4x Slower	***
latency_p99	500	6.64	0.0	80.1	0.01	12.1x Slower	***
dropped_iterations_rate	1000	0.0	0.0	60.39	1.79	*N/A*	***
latency_avg	1000	2.73	0.0	375.11	0.01	137.3x Slower	***
latency_p50	1000	2.38	0.0	9.52	0.0	4.0x Slower	***
latency_p95	1000	5.12	0.0	1637.58	0.03	320.1x Slower	***
latency_p99	1000	8.44	0.0	1995.49	0.03	236.5x Slower	***

Table B.8: Statistical comparison of latencies for POST /courses with prerequisites (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	4.11	0.0	6.88	0.0	1.7x Slower	***
latency_p50	25	3.78	0.0	6.19	0.0	1.6x Slower	***
latency_p95	25	6.49	0.0	11.96	0.0	1.8x Slower	***
latency_p99	25	14.58	0.0	19.88	0.0	1.4x Slower	***
latency_avg	50	3.17	0.0	5.52	0.0	1.7x Slower	***
latency_p50	50	2.9	0.0	4.95	0.0	1.7x Slower	***
latency_p95	50	5.46	0.0	10.53	0.0	1.9x Slower	***
latency_p99	50	7.31	0.0	15.01	0.0	2.1x Slower	***
latency_avg	100	2.5	0.0	4.26	0.0	1.7x Slower	***
latency_p50	100	2.27	0.0	3.55	0.0	1.6x Slower	***
latency_p95	100	4.54	0.0	8.99	0.0	2.0x Slower	***
latency_p99	100	6.48	0.0	13.76	0.0	2.1x Slower	***
latency_avg	200	2.15	0.0	3.8	0.0	1.8x Slower	***
latency_p50	200	1.97	0.0	2.98	0.0	1.5x Slower	***
latency_p95	200	3.59	0.0	9.01	0.0	2.5x Slower	***
latency_p99	200	5.52	0.0	14.85	0.0	2.7x Slower	***
latency_avg	500	2.09	0.0	9.71	0.0	4.6x Slower	***
latency_p50	500	1.92	0.0	3.81	0.0	2.0x Slower	***
latency_p95	500	3.3	0.0	36.37	0.0	11.0x Slower	***
latency_p99	500	6.04	0.0	80.66	0.01	13.3x Slower	***
latency_avg	1000	2.45	0.0	89.45	0.0	36.5x Slower	***
latency_p50	1000	2.12	0.0	8.85	0.0	4.2x Slower	***
latency_p95	1000	4.65	0.0	362.02	0.01	77.8x Slower	***
latency_p99	1000	8.04	0.0	455.05	0.01	56.6x Slower	***

Table B.9: Statistical comparison of latencies for POST /courses with prerequisites (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.02	0.0019	0.0391	0.0024	2.0x Higher	***
50	0.0237	0.0019	0.0544	0.0032	2.3x Higher	***
100	0.0328	0.0031	0.0913	0.0046	2.8x Higher	***
200	0.0404	0.0051	0.1531	0.0061	3.8x Higher	***
500	0.0677	0.0006	0.3424	0.0036	5.1x Higher	***
1000	0.1423	0.0012	0.4594	0.0025	3.2x Higher	***

Table B.10: Statistical comparison of *cpu_usage* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
50	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
100	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
200	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
500	12.0	0.0	61.0	2.5	5.1x Higher	***
1000	25.0	0.5	200.0	0.0	8.0x Higher	***

Table B.11: Statistical comparison of *tomcat_threads* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0	0.0	0.0	*N/A*	***
50	0.0	0.0	0.0	0.0	*N/A*	***
100	0.0	0.0	0.0	0.0	*N/A*	***
200	0.0	0.0	1.0	0.5	*N/A*	***
500	1.0	0.0	6.0	0.5	6.0x Higher	***
1000	2.0	0.0	10.0	0.0	5.0x Higher	***

Table B.12: Statistical comparison of *hikari_connections* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.94	0.01	20.3	0.01	1.9x Higher	***
50	13.41	0.01	23.86	0.01	1.8x Higher	***
100	18.39	0.02	30.91	0.01	1.7x Higher	***
200	28.33	0.02	45.01	0.01	1.6x Higher	***
500	58.13	0.03	76.99	0.37	1.3x Higher	***
1000	107.64	0.05	64.26	0.36	1.7x Lower	***

Table B.13: Statistical comparison of Data Store Size (MB) for POST /courses with prerequisites, aggregated over at least 25 runs

B.3 L3: Enroll to Lecture

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	25	7.04	0.0	17.2	0.0	2.4x Slower	***
latency_p50	25	6.46	0.0	15.64	0.0	2.4x Slower	***
latency_p95	25	10.0	0.0	26.08	0.0	2.6x Slower	***
latency_p99	25	19.36	0.0	35.23	0.0	1.8x Slower	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	50	5.97	0.0	15.5	0.0	2.6x Slower	***
latency_p50	50	5.58	0.0	13.91	0.0	2.5x Slower	***
latency_p95	50	8.57	0.0	24.64	0.0	2.9x Slower	***
latency_p99	50	11.33	0.0	36.39	0.0	3.2x Slower	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	100	5.37	0.0	42.88	0.0	8.0x Slower	***
latency_p50	100	5.13	0.0	15.14	0.0	3.0x Slower	***
latency_p95	100	7.24	0.0	154.19	0.03	21.3x Slower	***
latency_p99	100	10.13	0.0	559.48	0.09	55.2x Slower	***
dropped_iterations_rate	200	0.0	0.0	36.33	0.37	N/A	***
failure_rate	200	0.0	0.0	0.04	0.0	N/A	***
latency_avg	200	5.69	0.0	1264.09	0.01	222.1x Slower	***
latency_p50	200	5.57	0.0	682.82	0.01	122.6x Slower	***
latency_p95	200	7.73	0.0	2616.77	0.06	338.6x Slower	***
latency_p99	200	9.5	0.0	10014.08	0.0	1054.0x Slower	***
dropped_iterations_rate	500	35.96	0.3	238.71	0.58	6.6x Slower	***
failure_rate	500	0.0	0.0	0.06	0.0	N/A	***
latency_avg	500	556.24	0.0	3349.81	0.01	6.0x Slower	***
latency_p50	500	14.84	0.0	2563.18	0.03	172.7x Slower	***
latency_p95	500	1991.0	0.01	11631.77	0.71	5.8x Slower	***
latency_p99	500	2593.67	0.01	11931.41	0.01	4.6x Slower	***

Table B.14: Statistical comparison of latencies for POST /lectures/{lectureId}/enroll (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	6.07	0.0	16.18	0.0	2.7x Slower	***
latency_p50	25	5.49	0.0	14.81	0.0	2.7x Slower	***
latency_p95	25	8.96	0.0	25.54	0.0	2.8x Slower	***
latency_p99	25	18.27	0.0	33.55	0.0	1.8x Slower	***
latency_avg	50	5.21	0.0	14.56	0.0	2.8x Slower	***
latency_p50	50	5.02	0.0	13.21	0.0	2.6x Slower	***
latency_p95	50	7.69	0.0	23.89	0.0	3.1x Slower	***
latency_p99	50	9.85	0.0	35.31	0.0	3.6x Slower	***
latency_avg	100	4.75	0.0	41.94	0.0	8.8x Slower	***
latency_p50	100	4.7	0.0	14.41	0.0	3.1x Slower	***
latency_p95	100	6.76	0.0	151.88	0.04	22.5x Slower	***
latency_p99	100	8.91	0.0	573.93	0.09	64.4x Slower	***
latency_avg	200	5.2	0.0	1247.57	0.02	240.1x Slower	***
latency_p50	200	5.13	0.0	689.99	0.01	134.5x Slower	***
latency_p95	200	7.06	0.0	2647.03	0.05	375.1x Slower	***
latency_p99	200	8.75	0.0	9698.48	0.01	1108.4x Slower	***
latency_avg	500	244.63	0.0	1283.39	0.01	5.2x Slower	***
latency_p50	500	13.82	0.0	633.12	0.03	45.8x Slower	***
latency_p95	500	1289.29	0.0	8969.55	0.06	7.0x Slower	***
latency_p99	500	1776.48	0.01	9823.63	0.01	5.5x Slower	***

Table B.15: Statistical comparison of latencies for POST /lectures/{lectureId}/enroll (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0338	0.0017	0.0914	0.0035	2.5x Higher	***
50	0.0416	0.002	0.1481	0.005	3.2x Higher	***
100	0.0639	0.0022	0.284	0.0087	4.2x Higher	***
200	0.1146	0.0032	0.4198	0.0113	3.5x Higher	***
500	0.3566	0.0066	0.4033	0.0124	1.1x Higher	***

Table B.16: Statistical comparison of *cpu_usage* for POST /lectures/{lectureId}/enroll, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x	n.s.
50	10.0	0.0	10.0	0.0	1.0x	n.s.
100	10.0	0.0	65.0	2.1559	5.8x Higher	***
200	10.0	0.0	200.0	4.2754	18.2x Higher	***
500	17.0	7.9176	200.0	0.0	2.1x Higher	***

Table B.17: Statistical comparison of *tomcat_threads* for POST /lectures/{lectureId}/enroll, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0312	0.0	0.081	3.3x Higher	***
50	0.0	0.0352	1.0	0.0898	3.6x Higher	***
100	0.0	0.0385	2.0	0.1316	5.3x Higher	***
200	1.0	0.0414	4.0	0.1711	4.7x Higher	***
500	4.0	0.3303	4.0	0.1692	1.4x Lower	***

Table B.18: Statistical comparison of *hikari_connections* for POST /lectures/{lectureId}/enroll, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.54	0.0	24.11	0.02	2.3x Higher	***
50	12.32	0.0	31.64	0.02	2.6x Higher	***
100	15.86	0.0	55.94	0.05	3.5x Higher	***
200	22.97	0.01	227.27	11.74	9.7x Higher	***
500	40.77	0.03	232.4	0.55	5.7x Higher	***

Table B.19: Statistical comparison of Data Store Size (MB) for POST /lectures/{lectureId}/enroll, aggregated over at least 25 runs

B.4 L4: Read Lectures for Student

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	25	2.52	0.0	3.15	0.0	1.3x Slower	***
latency_p50	25	2.37	0.0	3.03	0.0	1.3x Slower	***
latency_p95	25	3.74	0.0	4.38	0.0	1.2x Slower	***
latency_p99	25	5.16	0.0	5.39	0.0	1.0x Slower	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	50	1.88	0.0	2.49	0.0	1.3x Slower	***
latency_p50	50	1.71	0.0	2.32	0.0	1.4x Slower	***
latency_p95	50	3.13	0.0	3.79	0.0	1.2x Slower	***
latency_p99	50	4.69	0.0	5.03	0.0	1.1x Slower	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	100	1.4	0.0	1.94	0.0	1.4x Slower	***
latency_p50	100	1.23	0.0	1.74	0.0	1.4x Slower	***
latency_p95	100	2.42	0.0	3.17	0.0	1.3x Slower	***
latency_p99	100	4.16	0.0	4.67	0.0	1.1x Slower	***
dropped_iterations_rate	200	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	200	1.02	0.0	1.46	0.0	1.4x Slower	***
latency_p50	200	0.88	0.0	1.27	0.0	1.4x Slower	***
latency_p95	200	1.83	0.0	2.59	0.0	1.4x Slower	***
latency_p99	200	3.35	0.0	4.21	0.0	1.3x Slower	***
dropped_iterations_rate	500	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	500	0.88	0.0	1.2	0.0	1.4x Slower	***
latency_p50	500	0.82	0.0	1.06	0.0	1.3x Slower	***
latency_p95	500	1.26	0.0	1.92	0.0	1.5x Slower	***
latency_p99	500	2.27	0.0	3.56	0.0	1.6x Slower	***
dropped_iterations_rate	1000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	1000	0.83	0.0	1.14	0.0	1.4x Slower	***
latency_p50	1000	0.78	0.0	0.99	0.0	1.3x Slower	***
latency_p95	1000	1.12	0.0	1.71	0.0	1.5x Slower	***
latency_p99	1000	2.14	0.0	3.86	0.0	1.8x Slower	***
dropped_iterations_rate	2000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	2000	0.82	0.0	1.51	0.0	1.8x Slower	***
latency_p50	2000	0.77	0.0	1.18	0.0	1.5x Slower	***
latency_p95	2000	1.09	0.0	3.24	0.0	3.0x Slower	***
latency_p99	2000	2.05	0.0	6.71	0.0	3.3x Slower	***
dropped_iterations_rate	3000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	3000	1.09	0.0	4.24	0.0	3.9x Slower	***
latency_p50	3000	0.92	0.0	2.58	0.0	2.8x Slower	***
latency_p95	3000	1.94	0.0	11.61	0.0	6.0x Slower	***
latency_p99	3000	4.28	0.0	26.63	0.0	6.2x Slower	***
dropped_iterations_rate	4000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	4000	141.42	0.06	53.75	0.03	2.6x Faster	**
latency_p50	4000	4.13	0.02	12.36	0.01	3.0x Slower	*
latency_p95	4000	574.89	0.14	201.1	0.13	2.9x Faster	**
latency_p99	4000	682.42	0.14	262.09	0.2	2.6x Faster	**

Table B.20: Statistical comparison of latencies for GET /lectures (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	1.6	0.0	2.22	0.0	1.4x Slower	***
latency_p50	25	1.49	0.0	2.11	0.0	1.4x Slower	***
latency_p95	25	2.7	0.0	3.27	0.0	1.2x Slower	***
latency_p99	25	3.62	0.0	3.91	0.0	1.1x Slower	***
latency_avg	50	1.14	0.0	1.7	0.0	1.5x Slower	***
latency_p50	50	0.99	0.0	1.58	0.0	1.6x Slower	***
latency_p95	50	2.19	0.0	2.79	0.0	1.3x Slower	***
latency_p99	50	3.21	0.0	3.62	0.0	1.1x Slower	***
latency_avg	100	0.8	0.0	1.26	0.0	1.6x Slower	***
latency_p50	100	0.61	0.0	1.11	0.0	1.8x Slower	***
latency_p95	100	1.65	0.0	2.33	0.0	1.4x Slower	***
latency_p99	100	2.74	0.0	3.28	0.0	1.2x Slower	***
latency_avg	200	0.64	0.0	1.03	0.0	1.6x Slower	***
latency_p50	200	0.55	0.0	0.75	0.0	1.4x Slower	***
latency_p95	200	1.28	0.0	1.95	0.0	1.5x Slower	***
latency_p99	200	2.19	0.0	2.91	0.0	1.3x Slower	***
latency_avg	500	0.57	0.0	0.87	0.0	1.5x Slower	***
latency_p50	500	0.52	0.0	0.59	0.0	1.1x Slower	***
latency_p95	500	0.99	0.0	1.51	0.0	1.5x Slower	***
latency_p99	500	1.68	0.0	2.71	0.0	1.6x Slower	***
latency_avg	1000	0.57	0.0	0.86	0.0	1.5x Slower	***
latency_p50	1000	0.51	0.0	0.56	0.0	1.1x Slower	***
latency_p95	1000	0.97	0.0	1.37	0.0	1.4x Slower	***
latency_p99	1000	1.52	0.0	2.98	0.0	2.0x Slower	***
latency_avg	2000	0.57	0.0	1.2	0.0	2.1x Slower	***
latency_p50	2000	0.51	0.0	0.8	0.0	1.6x Slower	***
latency_p95	2000	0.97	0.0	2.66	0.0	2.7x Slower	***
latency_p99	2000	1.4	0.0	5.53	0.0	3.9x Slower	***
latency_avg	3000	0.79	0.0	3.64	0.0	4.6x Slower	***
latency_p50	3000	0.56	0.0	2.21	0.0	4.0x Slower	***
latency_p95	3000	1.44	0.0	10.19	0.0	7.1x Slower	***
latency_p99	3000	3.26	0.0	24.01	0.0	7.4x Slower	***
latency_avg	4000	23.18	0.0	19.79	0.01	1.2x Faster	n.s.
latency_p50	4000	2.33	0.0	11.08	0.01	4.8x Slower	***
latency_p95	4000	136.23	0.01	58.66	0.0	2.3x Faster	***
latency_p99	4000	192.52	0.0	76.94	0.01	2.5x Faster	***

Table B.21: Statistical comparison of latencies for GET /lectures (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0138	0.0012	0.0201	0.0012	1.5x Higher	***
50	0.0163	0.0013	0.0277	0.0013	1.7x Higher	***
100	0.0214	0.0013	0.0416	0.002	1.9x Higher	***
200	0.0251	0.0008	0.0446	0.002	1.8x Higher	***
500	0.0536	0.0006	0.0941	0.0015	1.8x Higher	***
1000	0.1121	0.0011	0.1835	0.0013	1.6x Higher	***
2000	0.2324	0.0016	0.3869	0.0025	1.7x Higher	***
3000	0.4149	0.0051	0.5437	0.0024	1.3x Higher	***
4000	0.6112	0.0219	0.6015	0.0062	1.0x Lower	n.s.

Table B.22: Statistical comparison of *cpu_usage* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
50	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
100	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
200	10.0	0.0	10.0	0.0	1.0x Lower	n.s.
500	10.0	0.0	10.0	0.0	1.0x Lower	***
1000	15.0	0.5	20.0	0.5	1.3x Higher	***
2000	24.0	0.0	47.0	0.5	2.0x Higher	***
3000	31.0	0.5	158.0	14.0	5.1x Higher	***
4000	200.0	0.0	200.0	0.0	1.0x Lower	***

Table B.23: Statistical comparison of *tomcat_threads* for GET /lectures, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0	0.0	0.0	*N/A*	***
50	0.0	0.0	0.0	0.0	*N/A*	n.s.
100	0.0	0.0	0.0	0.0	*N/A*	n.s.
200	0.0	0.0	0.0	0.0	*N/A*	n.s.
500	0.0	0.0	0.0	0.0	*N/A*	***
1000	0.0	0.0	0.0	0.0	*N/A*	***
2000	1.0	0.0	0.0	0.5	*N/A*	***
3000	2.0	0.0	1.0	0.5	2.0x Lower	**
4000	8.0	0.5	6.0	1.0	1.3x Lower	***

Table B.24: Statistical comparison of *hikari_connections* for GET /lectures, aggregated over at least 25 runs

B.5 L5: Read All Lectures

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	25	26.25	0.0	4.43	0.0	5.9x Faster	***
latency_p50	25	25.75	0.0	4.03	0.0	6.4x Faster	***
latency_p95	25	29.31	0.0	6.96	0.0	4.2x Faster	***
latency_p99	25	33.76	0.0	9.73	0.0	3.5x Faster	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	50	27.13	0.0	3.56	0.0	7.6x Faster	***
latency_p50	50	26.68	0.0	3.21	0.0	8.3x Faster	***
latency_p95	50	30.22	0.0	6.0	0.0	5.0x Faster	***
latency_p99	50	33.76	0.0	8.67	0.0	3.9x Faster	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	N/A	n.s.
latency_avg	100	36.75	0.0	2.75	0.0	13.4x Faster	***
latency_p50	100	32.73	0.0	2.44	0.0	13.4x Faster	***
latency_p95	100	65.72	0.03	4.87	0.0	13.5x Faster	***
latency_p99	100	124.88	0.03	7.77	0.0	16.1x Faster	***
dropped_iterations_rate	150	25.16	0.14	0.0	0.0	N/A	***
latency_avg	150	2256.08	0.01	2.16	0.0	1042.7x Faster	***
latency_p50	150	2733.07	0.01	1.83	0.0	1495.0x Faster	***
latency_p95	150	4778.46	0.02	4.24	0.0	1127.1x Faster	***
latency_p99	150	6548.91	0.02	7.33	0.0	893.2x Faster	***
dropped_iterations_rate	200	59.19	0.21	0.0	0.0	N/A	***
latency_avg	200	3210.72	0.01	1.98	0.0	1618.7x Faster	***
latency_p50	200	3697.29	0.01	1.63	0.0	2273.8x Faster	***
latency_p95	200	5815.3	0.04	3.79	0.0	1534.3x Faster	***
latency_p99	200	7568.92	0.04	7.13	0.0	1061.2x Faster	***
dropped_iterations_rate	250	94.27	0.17	0.0	0.0	N/A	***
latency_avg	250	4113.37	0.01	1.89	0.0	2180.6x Faster	***
latency_p50	250	4649.66	0.01	1.55	0.0	3003.4x Faster	***
latency_p95	250	6864.04	0.03	3.53	0.0	1943.9x Faster	***
latency_p99	250	8594.06	0.04	6.98	0.0	1232.0x Faster	***
dropped_iterations_rate	500	260.24	0.21	0.0	0.0	N/A	***
latency_avg	500	8357.45	0.02	1.69	0.0	4935.9x Faster	***
latency_p50	500	9253.31	0.02	1.38	0.0	6701.6x Faster	***
latency_p95	500	11820.67	0.04	3.21	0.0	3688.1x Faster	***
latency_p99	500	13540.36	0.07	7.09	0.0	1909.2x Faster	***
dropped_iterations_rate	1000	566.0	0.14	0.0	0.0	N/A	***
latency_avg	1000	16657.54	0.02	1.83	0.0	9115.5x Faster	***
latency_p50	1000	18600.29	0.04	1.4	0.0	13272.3x Faster	***
latency_p95	1000	21746.72	0.08	3.98	0.0	5464.2x Faster	***
latency_p99	1000	23531.71	0.1	9.67	0.0	2433.1x Faster	***

Table B.25: Statistical comparison of latencies for GET /lectures/all (client), aggregated over at least 20 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	25.07	0.0	3.11	0.0	8.1x Faster	***
latency_p50	25	25.31	0.0	2.75	0.0	9.2x Faster	***
latency_p95	25	27.96	0.0	5.47	0.0	5.1x Faster	***
latency_p99	25	33.16	0.0	8.06	0.0	4.1x Faster	***
latency_avg	50	25.87	0.0	2.44	0.0	10.6x Faster	***
latency_p50	50	25.43	0.0	2.11	0.0	12.0x Faster	***
latency_p95	50	30.44	0.0	4.65	0.0	6.5x Faster	***
latency_p99	50	33.27	0.0	6.89	0.0	4.8x Faster	***
latency_avg	100	35.54	0.0	1.83	0.0	19.4x Faster	***
latency_p50	100	31.3	0.0	1.58	0.0	19.9x Faster	***
latency_p95	100	63.78	0.03	3.69	0.0	17.3x Faster	***
latency_p99	100	125.91	0.03	6.18	0.0	20.4x Faster	***
latency_avg	150	1565.62	0.0	1.62	0.0	963.9x Faster	***
latency_p50	150	1838.09	0.0	1.36	0.0	1354.3x Faster	***
latency_p95	150	3905.5	0.0	3.31	0.0	1178.3x Faster	***
latency_p99	150	5609.12	0.03	5.7	0.0	984.4x Faster	***
latency_avg	200	1670.39	0.01	1.51	0.0	1108.4x Faster	***
latency_p50	200	1869.72	0.0	1.27	0.0	1473.8x Faster	***
latency_p95	200	3921.77	0.0	3.07	0.0	1276.6x Faster	***
latency_p99	200	5654.95	0.01	5.59	0.0	1012.1x Faster	***
latency_avg	250	1719.74	0.0	1.43	0.0	1198.8x Faster	***
latency_p50	250	1884.6	0.0	1.21	0.0	1551.2x Faster	***
latency_p95	250	3931.56	0.0	2.94	0.0	1338.6x Faster	***
latency_p99	250	5688.65	0.02	5.54	0.0	1025.9x Faster	***
latency_avg	500	2014.02	NaN	1.33	0.0	1515.5x Faster	n.s.
latency_p50	500	1946.13	NaN	1.07	0.0	1823.6x Faster	n.s.
latency_p95	500	3543.35	NaN	2.69	0.0	1317.9x Faster	n.s.
latency_p99	500	3571.98	NaN	6.09	0.0	586.6x Faster	n.s.
latency_avg	1000	2504.97	0.12	1.47	0.0	1701.2x Faster	***
latency_p50	1000	2039.8	0.04	1.19	0.0	1720.2x Faster	***
latency_p95	1000	4359.39	0.58	3.35	0.0	1302.7x Faster	***
latency_p99	1000	5471.07	0.59	8.31	0.0	658.6x Faster	***

Table B.26: Statistical comparison of latencies for GET /lectures/all (server), aggregated over at least 20 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0188	0.0012	0.0302	0.0013	1.6x Higher	***
50	0.0288	0.0008	0.0416	0.0014	1.4x Higher	***
100	0.0567	0.001	0.0566	0.0033	1.0x	n.s.
200	0.0706	0.0059	0.0712	0.0043	1.0x	n.s.
250	—	—	0.0827	0.0019	N/A	n.s.
500	—	—	0.1536	0.0008	N/A	n.s.
1000	—	—	0.3043	0.0014	N/A	n.s.

Table B.27: Statistical comparison of *cpu_usage* for GET /lectures/all, aggregated over at least 20 runs. No data could be collected for CRUD at 250, 500, 1000 RPS

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x	n.s.
50	10.0	0.0	10.0	0.0	1.0x	n.s.
100	15.0	0.5	10.0	0.0	1.5x Lower	***
200	96.0	87.0	10.0	0.0	9.6x Lower	***
250	10.0	6.5	10.0	0.0	1.0x	***
500	200.0	31.0	10.0	0.0	20.0x Lower	***
1000	200.0	0.0	20.0	1.0	10.0x Lower	***

Table B.28: Statistical comparison of *tomcat_threads* for GET /lectures/all, aggregated over at least 20 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	1.0	0.0	0.0	0.0	NaN	***
50	1.0	0.0	0.0	0.0	NaN	***
100	3.0	0.0	0.0	0.0	NaN	***
200	10.0	0.5	0.0	0.0	NaN	***
250	3.0	1.0	0.0	0.0	NaN	***
500	10.0	0.5	0.0	0.0	NaN	***
1000	10.0	5.0	0.0	0.0	NaN	***

Table B.29: Statistical comparison of *hikari_connections* for GET /lectures/all, aggregated over at least 20 runs

B.6 L6: Get Credits

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	25	2.52	0.0	2.35	0.0	1.1x Faster	***
latency_p50	25	2.42	0.0	2.28	0.0	1.1x Faster	***
latency_p95	25	3.34	0.0	3.08	0.0	1.1x Faster	***
latency_p99	25	4.44	0.0	3.75	0.0	1.2x Faster	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	50	2.09	0.0	1.94	0.0	1.1x Faster	***
latency_p50	50	1.98	0.0	1.85	0.0	1.1x Faster	***
latency_p95	50	2.85	0.0	2.68	0.0	1.1x Faster	***
latency_p99	50	3.94	0.0	3.42	0.0	1.2x Faster	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	100	1.76	0.0	1.59	0.0	1.1x Faster	***
latency_p50	100	1.67	0.0	1.51	0.0	1.1x Faster	***
latency_p95	100	2.42	0.0	2.23	0.0	1.1x Faster	***
latency_p99	100	3.47	0.0	3.15	0.0	1.1x Faster	***
dropped_iterations_rate	200	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	200	1.46	0.0	1.24	0.0	1.2x Faster	***
latency_p50	200	1.37	0.0	1.15	0.0	1.2x Faster	***
latency_p95	200	2.03	0.0	1.84	0.0	1.1x Faster	***
latency_p99	200	3.02	0.0	2.76	0.0	1.1x Faster	***
dropped_iterations_rate	500	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	500	1.37	0.0	1.07	0.0	1.3x Faster	***
latency_p50	500	1.3	0.0	1.0	0.0	1.3x Faster	***
latency_p95	500	1.86	0.0	1.44	0.0	1.3x Faster	***
latency_p99	500	2.49	0.0	2.28	0.0	1.1x Faster	***
dropped_iterations_rate	1000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	1000	1.4	0.0	1.04	0.0	1.3x Faster	***
latency_p50	1000	1.3	0.0	0.98	0.0	1.3x Faster	***
latency_p95	1000	1.95	0.0	1.32	0.0	1.5x Faster	***
latency_p99	1000	3.03	0.0	2.33	0.0	1.3x Faster	***
dropped_iterations_rate	2000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	2000	2.51	0.0	1.22	0.0	2.1x Faster	***
latency_p50	2000	1.84	0.0	1.05	0.0	1.7x Faster	***
latency_p95	2000	4.52	0.0	1.98	0.0	2.3x Faster	***
latency_p99	2000	10.71	0.0	4.44	0.0	2.4x Faster	***
dropped_iterations_rate	3000	359.02	4.88	0.0	0.0	*N/A*	***
latency_avg	3000	963.06	0.01	2.77	0.0	347.7x Faster	***
latency_p50	3000	1252.0	0.0	1.85	0.0	677.8x Faster	***
latency_p95	3000	1510.21	0.01	7.28	0.0	207.5x Faster	***
latency_p99	3000	1607.3	0.01	15.38	0.0	104.5x Faster	***
dropped_iterations_rate	4000	979.2	5.85	0.0	0.0	*N/A*	***
latency_avg	4000	1481.69	0.01	11.02	0.0	134.5x Faster	***
latency_p50	4000	1732.48	0.01	4.37	0.0	396.3x Faster	***
latency_p95	4000	2031.7	0.01	46.93	0.01	43.3x Faster	***
latency_p99	4000	2170.61	0.03	95.55	0.01	22.7x Faster	***
dropped_iterations_rate	5000	1581.01	5.85	38.39	12.42	41.2x Lower	***
latency_avg	5000	1857.55	0.01	667.2	0.1	2.8x Faster	***
latency_p50	5000	2150.63	0.01	773.75	0.15	2.8x Faster	***
latency_p95	5000	2470.19	0.01	1078.16	0.02	2.3x Faster	***
latency_p99	5000	2643.37	0.03	1137.65	0.02	2.3x Faster	***

Table B.30: Statistical comparison of latencies for GET /stats/credits (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	1.64	0.0	1.49	0.0	1.1x Faster	***
latency_p50	25	1.57	0.0	1.44	0.0	1.1x Faster	***
latency_p95	25	2.37	0.0	2.09	0.0	1.1x Faster	***
latency_p99	25	3.01	0.0	2.5	0.0	1.2x Faster	***
latency_avg	50	1.37	0.0	1.19	0.0	1.1x Faster	***
latency_p50	50	1.29	0.0	1.14	0.0	1.1x Faster	***
latency_p95	50	2.03	0.0	1.79	0.0	1.1x Faster	***
latency_p99	50	2.66	0.0	2.27	0.0	1.2x Faster	***
latency_avg	100	1.16	0.0	0.94	0.0	1.2x Faster	***
latency_p50	100	1.1	0.0	0.69	0.0	1.6x Faster	***
latency_p95	100	1.72	0.0	1.47	0.0	1.2x Faster	***
latency_p99	100	2.32	0.0	1.98	0.0	1.2x Faster	***
latency_avg	200	1.08	0.0	0.82	0.0	1.3x Faster	***
latency_p50	200	1.0	0.0	0.58	0.0	1.7x Faster	***
latency_p95	200	1.6	0.0	1.29	0.0	1.2x Faster	***
latency_p99	200	2.06	0.0	1.71	0.0	1.2x Faster	***
latency_avg	500	1.08	0.0	0.74	0.0	1.5x Faster	***
latency_p50	500	1.02	0.0	0.53	0.0	1.9x Faster	***
latency_p95	500	1.58	0.0	1.02	0.0	1.5x Faster	***
latency_p99	500	1.99	0.0	1.58	0.0	1.3x Faster	***
latency_avg	1000	1.13	0.0	0.74	0.0	1.5x Faster	***
latency_p50	1000	1.05	0.0	0.52	0.0	2.0x Faster	***
latency_p95	1000	1.67	0.0	0.99	0.0	1.7x Faster	***
latency_p99	1000	2.38	0.0	1.66	0.0	1.4x Faster	***
latency_avg	2000	2.15	0.0	0.92	0.0	2.3x Faster	***
latency_p50	2000	1.54	0.0	0.61	0.0	2.5x Faster	***
latency_p95	2000	3.98	0.0	1.6	0.0	2.5x Faster	***
latency_p99	2000	9.58	0.0	3.38	0.0	2.8x Faster	***
latency_avg	3000	72.22	0.0	2.29	0.0	31.6x Faster	***
latency_p50	3000	6.12	0.0	1.53	0.0	4.0x Faster	***
latency_p95	3000	261.79	0.0	6.22	0.0	42.1x Faster	***
latency_p99	3000	344.01	0.0	13.62	0.0	25.3x Faster	***
latency_avg	4000	79.02	0.0	8.81	0.0	9.0x Faster	***
latency_p50	4000	6.81	0.0	3.78	0.0	1.8x Faster	***
latency_p95	4000	263.43	0.0	39.23	0.0	6.7x Faster	***
latency_p99	4000	350.1	0.0	55.89	0.0	6.3x Faster	***
latency_avg	5000	78.74	0.0	34.61	0.0	2.3x Faster	***
latency_p50	5000	6.81	0.0	38.01	0.0	5.6x Slower	***
latency_p95	5000	262.91	0.0	53.99	0.0	4.9x Faster	***
latency_p99	5000	349.45	0.0	63.7	0.0	5.5x Faster	***

Table B.31: Statistical comparison of latencies for GET /stats/credits (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.01	0.0006	0.0113	0.0	1.1x Higher	***
50	0.0163	0.0006	0.0189	0.0012	1.2x Higher	**
100	0.0264	0.0013	0.0266	0.0019	1.0x	*
200	0.0418	0.0025	0.0382	0.0007	1.1x Lower	n.s.
500	0.0918	0.0022	0.0857	0.0008	1.1x Lower	***
1000	0.2069	0.0023	0.1681	0.0015	1.2x Lower	***
2000	0.4454	0.0023	0.3535	0.0035	1.3x Lower	***
3000	0.5763	0.0006	0.5338	0.0015	1.1x Lower	***
4000	0.5608	0.0007	0.572	0.0067	1.0x	***
5000	0.5703	0.24	0.6996	0.0018	1.2x Higher	***

Table B.32: Statistical comparison of *cpu_usage* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x	n.s.
50	10.0	0.0	10.0	0.0	1.0x	n.s.
100	10.0	0.0	10.0	0.0	1.0x	n.s.
200	10.0	0.0	10.0	0.0	1.0x	n.s.
500	10.0	0.0	10.0	0.0	1.0x	***
1000	11.0	0.5	16.0	0.0	1.5x Higher	***
2000	36.0	3.0	31.0	0.5	1.2x Lower	***
3000	200.0	0.0	123.0	10.5	1.6x Lower	***
4000	200.0	0.0	200.0	0.0	1.0x	n.s.
5000	200.0	53.5	200.0	0.0	1.0x	***

Table B.33: Statistical comparison of *tomcat_threads* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0	0.0	0.0	*N/A*	n.s.
50	0.0	0.0	0.0	0.0	*N/A*	n.s.
100	0.0	0.0	0.0	0.0	*N/A*	***
200	0.0	0.0	0.0	0.0	*N/A*	***
500	0.0	0.5	0.0	0.0	*N/A*	***
1000	1.0	0.0	0.0	0.0	*N/A*	***
2000	3.0	0.0	0.0	0.0	*N/A*	***
3000	10.0	0.0	1.0	0.0	10.0x Lower	***
4000	10.0	0.0	2.0	0.5	5.0x Lower	***
5000	9.0	1.0	7.0	0.5	1.3x Lower	**

Table B.34: Statistical comparison of *hikari_connections* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

B.7 L7: Time to Consistency

This load test consists of 2 requests per iteration. Therefore, the client-side latencies for both requests are listed separately. In a third table, the dropped iterations are listed.

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	10	6.81	0.3	9.31	0.06	1.4x Slower	***
latency_p50	10	5.69	0.04	8.3	0.03	1.5x Slower	***
latency_p95	10	15.44	0.1	18.56	0.68	1.2x Slower	***
latency_p99	10	17.02	0.09	21.91	0.21	1.3x Slower	***
latency_avg	20	5.08	0.02	7.6	0.02	1.5x Slower	***
latency_p50	20	4.62	0.04	6.82	0.02	1.5x Slower	***
latency_p95	20	7.28	0.06	11.82	0.09	1.6x Slower	***
latency_p99	20	16.05	0.13	20.44	0.25	1.3x Slower	***
latency_avg	50	3.67	0.02	5.95	0.02	1.6x Slower	***
latency_p50	50	3.35	0.02	5.31	0.01	1.6x Slower	***
latency_p95	50	5.82	0.04	10.24	0.08	1.8x Slower	***
latency_p99	50	7.8	0.06	13.81	0.2	1.8x Slower	***
latency_avg	100	2.96	0.02	4.65	0.01	1.6x Slower	***
latency_p50	100	2.72	0.02	4.02	0.01	1.5x Slower	***
latency_p95	100	4.74	0.07	8.75	0.06	1.8x Slower	***
latency_p99	100	6.89	0.06	13.31	0.1	1.9x Slower	***
latency_avg	200	2.59	0.01	4.44	0.04	1.7x Slower	***
latency_p50	200	2.37	0.01	3.5	0.02	1.5x Slower	***
latency_p95	200	3.82	0.05	9.77	0.22	2.6x Slower	***
latency_p99	200	5.98	0.03	16.8	0.77	2.8x Slower	***
latency_avg	300	2.57	0.01	6.8	0.17	2.6x Slower	***
latency_p50	300	2.38	0.01	3.62	0.02	1.5x Slower	***
latency_p95	300	3.69	0.06	22.33	0.96	6.0x Slower	***
latency_p99	300	6.43	0.11	40.38	1.46	6.3x Slower	***
latency_avg	400	2.56	0.01	15.67	1.32	6.1x Slower	***
latency_p50	400	2.36	0.01	4.76	0.06	2.0x Slower	***
latency_p95	400	3.68	0.05	67.3	12.07	18.3x Slower	***
latency_p99	400	6.62	0.09	146.14	26.15	22.1x Slower	***
latency_avg	500	2.63	0.01	182.81	6.77	69.4x Slower	***
latency_p50	500	2.43	0.01	6.85	0.33	2.8x Slower	***
latency_p95	500	3.73	0.05	985.08	38.15	263.8x Slower	***
latency_p99	500	6.9	0.08	1193.07	27.85	172.8x Slower	***

Table B.35: Statistical comparison of latencies for POST /lectures/create (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	10	3.02	0.01	3.69	0.02	1.2x Slower	***
latency_p50	10	2.75	0.02	3.24	0.02	1.2x Slower	***
latency_p95	10	4.8	0.03	6.43	0.06	1.3x Slower	***
latency_p99	10	6.45	0.09	7.78	0.06	1.2x Slower	***
latency_avg	20	2.51	0.01	3.03	0.01	1.2x Slower	***
latency_p50	20	2.28	0.02	2.58	0.01	1.1x Slower	***
latency_p95	20	4.03	0.03	5.66	0.04	1.4x Slower	***
latency_p99	20	5.52	0.11	7.58	0.07	1.4x Slower	***
latency_avg	50	1.87	0.01	2.34	0.01	1.2x Slower	***
latency_p50	50	1.67	0.01	1.98	0.01	1.2x Slower	***
latency_p95	50	3.13	0.01	4.84	0.04	1.5x Slower	***
latency_p99	50	4.55	0.05	7.04	0.08	1.5x Slower	***
latency_avg	100	1.67	0.05	1.84	0.01	1.1x Slower	***
latency_p50	100	1.52	0.08	1.49	0.01	1.0x Faster	n.s.
latency_p95	100	2.7	0.03	4.1	0.04	1.5x Slower	***
latency_p99	100	3.93	0.05	7.07	0.11	1.8x Slower	***
latency_avg	200	1.56	0.03	1.82	0.02	1.2x Slower	***
latency_p50	200	1.36	0.02	1.31	0.01	1.0x Faster	***
latency_p95	200	2.4	0.02	4.69	0.1	2.0x Slower	***
latency_p99	200	3.5	0.03	9.02	0.36	2.6x Slower	***
latency_avg	300	1.67	0.01	3.1	0.1	1.9x Slower	***
latency_p50	300	1.52	0.01	1.38	0.01	1.1x Faster	***
latency_p95	300	2.56	0.02	11.46	0.5	4.5x Slower	***
latency_p99	300	3.7	0.05	23.15	0.81	6.3x Slower	***
latency_avg	400	1.86	0.03	7.14	0.33	3.8x Slower	***
latency_p50	400	1.74	0.03	1.71	0.03	1.0x Faster	n.s.
latency_p95	400	2.89	0.02	32.75	2.11	11.3x Slower	***
latency_p99	400	4.02	0.06	59.97	2.87	14.9x Slower	***
latency_avg	500	2.16	0.03	94.17	3.68	43.6x Slower	***
latency_p50	500	2.12	0.04	2.71	0.19	1.3x Slower	***
latency_p95	500	3.32	0.04	595.97	28.16	179.7x Slower	***
latency_p99	500	4.6	0.14	737.46	18.52	160.3x Slower	***

Table B.36: Statistical comparison of latencies for GET /lectures/{lectureId} (client), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	0	0	0	0	*N/A*	n.s.
25	0	0	0	0	*N/A*	n.s.
50	0	0	0	0	*N/A*	n.s.
100	0	0	0	0	*N/A*	n.s.
200	0	0	0	0	*N/A*	n.s.
300	0	0	0	0	*N/A*	n.s.
400	0	0	0	0	*N/A*	n.s.
500	0	0	8.5	1.12	*N/A*	n.s.

Table B.37: Statistical comparison of *dropped_iterations_rate* for POST /lectures/create, then GET /lectures/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	0.0201	0.0013	0.0391	0.0024	1.9x Higher	***
20	0.0288	0.0021	0.048	0.0031	1.7x Higher	***
50	0.0328	0.0019	0.0658	0.0044	2.0x Higher	***
100	0.0456	0.0059	0.1101	0.0049	2.4x Higher	***
200	0.0682	0.002	0.1779	0.0025	2.6x Higher	***
300	0.0951	0.0018	0.2602	0.0018	2.7x Higher	***
400	0.1249	0.0009	0.3566	0.0029	2.9x Higher	***
500	0.1601	0.001	0.426	0.0035	2.7x Higher	***

Table B.38: Statistical comparison of *cpu_usage* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	10.0	0.0	10.0	0.0	1.0x	n.s.
20	10.0	0.0	10.0	0.0	1.0x	n.s.
50	10.0	0.0	10.0	0.0	1.0x	n.s.
100	10.0	0.0	10.0	0.0	1.0x	n.s.
200	10.0	0.0	11.0	0.0	1.1x Higher	***
300	11.0	0.5	39.0	1.0	3.5x Higher	***
400	13.0	0.0	100.0	1.5	7.7x Higher	***
500	17.0	0.0	200.0	0.0	11.8x Higher	***

Table B.39: Statistical comparison of *tomcat_threads* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	0.0	0.0	0.0	0.0	*N/A*	***
20	0.0	0.0	0.0	0.0	*N/A*	***
50	0.0	0.0	0.0	0.0	*N/A*	***
100	0.0	0.0	0.0	0.0	*N/A*	n.s.
200	1.0	0.0	1.0	0.5	1.0x	***
300	1.0	0.0	3.0	0.0	3.0x Higher	***
400	1.0	0.0	4.0	0.0	4.0x Higher	***
500	2.0	0.0	5.0	0.0	2.5x Higher	***

Table B.40: Statistical comparison of *hikari_connections* for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	9.27	0.01	17.97	0.01	1.9x Higher	***
20	10.02	0.02	19.08	0.01	1.9x Higher	***
50	12.33	0.02	22.51	0.01	1.8x Higher	***
100	16.11	0.01	28.15	0.01	1.7x Higher	***
200	23.69	0.01	39.5	0.01	1.7x Higher	***
300	31.26	0.01	50.89	0.02	1.6x Higher	***
400	38.78	0.01	60.11	0.23	1.6x Higher	***
500	46.29	0.02	57.78	0.57	1.2x Higher	***

Table B.41: Statistical comparison of Data Store Size (MB) for POST /lectures/create; then /GET/{lectureId}, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
10	1.0	0.0	1.0	0.0	1.0x	n.s.
20	1.0	0.0	1.0	0.0	1.0x	n.s.
50	1.0	0.0	1.0	0.0	1.0x	n.s.
100	1.0	0.0	1.0	0.0	1.0x	n.s.
200	1.0	0.0	1.0	0.0	1.0x	n.s.
300	1.0	0.0	0.75	0.01	1.3x Lower	***
400	1.0	0.0	0.02	0.0	53.3x Lower	***
500	1.0	0.0	0.01	0.0	126.8x Lower	***

Table B.42: Statistical comparison of *read_visible_rate* for GET /lectures/{lectureId}, aggregated over at least 25 runs

B.8 L8: Reconstruct Grade History

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
dropped_iterations_rate	25	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	25	2.05	0.0	3.47	0.0	1.7x Slower	***
latency_p50	25	1.97	0.0	3.34	0.0	1.7x Slower	***
latency_p95	25	2.79	0.0	4.54	0.0	1.6x Slower	***
latency_p99	25	3.83	0.0	6.03	0.0	1.6x Slower	***
dropped_iterations_rate	50	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	50	1.67	0.0	2.87	0.0	1.7x Slower	***
latency_p50	50	1.58	0.0	2.71	0.0	1.7x Slower	***
latency_p95	50	2.4	0.0	3.98	0.0	1.7x Slower	***
latency_p99	50	3.36	0.0	5.57	0.0	1.7x Slower	***
dropped_iterations_rate	100	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	100	1.36	0.0	2.43	0.0	1.8x Slower	***
latency_p50	100	1.28	0.0	2.27	0.0	1.8x Slower	***
latency_p95	100	1.94	0.0	3.47	0.0	1.8x Slower	***
latency_p99	100	2.95	0.0	5.12	0.0	1.7x Slower	***
dropped_iterations_rate	200	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	200	1.1	0.0	2.02	0.0	1.8x Slower	***
latency_p50	200	1.02	0.0	1.84	0.0	1.8x Slower	***
latency_p95	200	1.61	0.0	3.02	0.0	1.9x Slower	***
latency_p99	200	2.47	0.0	4.76	0.0	1.9x Slower	***
dropped_iterations_rate	500	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	500	0.95	0.0	1.88	0.0	2.0x Slower	***
latency_p50	500	0.9	0.0	1.73	0.0	1.9x Slower	***
latency_p95	500	1.28	0.0	2.62	0.0	2.0x Slower	***
latency_p99	500	1.86	0.0	4.92	0.0	2.6x Slower	***
dropped_iterations_rate	1000	0.0	0.0	0.0	0.0	*N/A*	n.s.
latency_avg	1000	0.96	0.0	2.19	0.0	2.3x Slower	***
latency_p50	1000	0.92	0.0	1.88	0.0	2.0x Slower	***
latency_p95	1000	1.23	0.0	3.92	0.0	3.2x Slower	***
latency_p99	1000	1.8	0.0	7.6	0.0	4.2x Slower	***
dropped_iterations_rate	2000	0.0	0.0	19.22	10.17	*N/A*	***
latency_avg	2000	1.04	0.0	1177.93	0.25	1131.0x Slower	***
latency_p50	2000	0.98	0.0	1360.65	0.22	1395.0x Slower	***
latency_p95	2000	1.37	0.0	2022.47	0.6	1478.6x Slower	***
latency_p99	2000	2.57	0.0	2167.05	0.61	844.6x Slower	***

Table B.43: Statistical comparison of latencies for GET /stats/grades/history (client), aggregated over at least 25 runs

		CRUD (ms)		ES-CQRS (ms)			
Metric	**RPS**	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
latency_avg	25	1.15	0.0	2.56	0.0	2.2x Slower	***
latency_p50	25	1.08	0.0	2.43	0.0	2.2x Slower	***
latency_p95	25	1.78	0.0	3.49	0.0	2.0x Slower	***
latency_p99	25	2.42	0.0	5.16	0.0	2.1x Slower	***
latency_avg	50	0.94	0.0	2.08	0.0	2.2x Slower	***
latency_p50	50	0.71	0.0	1.95	0.0	2.7x Slower	***
latency_p95	50	1.55	0.0	3.04	0.0	2.0x Slower	***
latency_p99	50	2.07	0.0	4.42	0.0	2.1x Slower	***
latency_avg	100	0.75	0.0	1.74	0.0	2.3x Slower	***
latency_p50	100	0.57	0.0	1.63	0.0	2.9x Slower	***
latency_p95	100	1.27	0.0	2.64	0.0	2.1x Slower	***
latency_p99	100	1.74	0.0	3.99	0.0	2.3x Slower	***
latency_avg	200	0.71	0.0	1.57	0.0	2.2x Slower	***
latency_p50	200	0.54	0.0	1.44	0.0	2.7x Slower	***
latency_p95	200	1.05	0.0	2.34	0.0	2.2x Slower	***
latency_p99	200	1.46	0.0	3.73	0.0	2.6x Slower	***
latency_avg	500	0.63	0.0	1.53	0.0	2.4x Slower	***
latency_p50	500	0.52	0.0	1.39	0.0	2.7x Slower	***
latency_p95	500	0.98	0.0	2.13	0.0	2.2x Slower	***
latency_p99	500	1.32	0.0	4.13	0.0	3.1x Slower	***
latency_avg	1000	0.67	0.0	1.86	0.0	2.8x Slower	***
latency_p50	1000	0.52	0.0	1.61	0.0	3.1x Slower	***
latency_p95	1000	0.98	0.0	3.43	0.0	3.5x Slower	***
latency_p99	1000	1.31	0.0	6.76	0.0	5.2x Slower	***
latency_avg	2000	0.75	0.0	87.01	0.0	115.5x Slower	***
latency_p50	2000	0.54	0.0	100.67	0.0	187.5x Slower	***
latency_p95	2000	1.05	0.0	130.56	0.0	124.6x Slower	***
latency_p99	2000	1.71	0.0	148.77	0.0	87.0x Slower	***

Table B.44: Statistical comparison of latencies for GET /stats/grades/history (server), aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0087	0.0006	0.0164	0.0006	1.9x Higher	***
50	0.0125	0.0006	0.0253	0.0013	2.0x Higher	***
100	0.0177	0.0012	0.0355	0.0018	2.0x Higher	***
200	0.0239	0.0007	0.0584	0.0023	2.4x Higher	***
500	0.0546	0.0009	0.1414	0.0018	2.6x Higher	***
1000	0.1118	0.0009	0.2892	0.0022	2.6x Higher	***
2000	0.2345	0.0017	0.5086	0.0028	2.2x Higher	***

Table B.45: Statistical comparison of *cpu_usage* for GET /stats/grades/history, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	10.0	0.0	10.0	0.0	1.0x	n.s.
50	10.0	0.0	10.0	0.0	1.0x	n.s.
100	10.0	0.0	10.0	0.0	1.0x	n.s.
200	10.0	0.0	10.0	0.0	1.0x	n.s.
500	10.0	0.0	10.0	0.0	1.0x	***
1000	15.0	0.0	19.0	0.5	1.3x Higher	***
2000	25.0	0.0	200.0	0.0	8.0x Higher	***

Table B.46: Statistical comparison of *tomcat_threads* for GET /stats/grades/history, aggregated over at least 25 runs

	CRUD (ms)		ES-CQRS (ms)			
RPS	**Median**	**CI ±**	**Median**	**CI ±**	**Ratio**	**Significance**
25	0.0	0.0	0.0	0.0	*N/A*	***
50	0.0	0.0	0.0	0.0	*N/A*	**
100	0.0	0.0	0.0	0.0	*N/A*	***
200	0.0	0.0	0.0	0.0	*N/A*	***
500	0.0	0.0	0.0	0.5	*N/A*	***
1000	1.0	0.5	1.0	0.0	1.0x	***
2000	1.0	0.5	10.0	0.5	10.0x Higher	***

Table B.47: Statistical comparison of *hikari_connections* for GET /stats/grades/history, aggregated over at least 25 runs

Appendix C

Static Analysis Results

This chapter lists tables containing the full results of static analysis metrics. To make the tables more readable, some package names are abbreviated according to Table C.1.

Package Name	**Abbreviation**
karsch.lukas	k.l
features	f
lectures	le
enrollment	e
command	c

Table C.1: Package name abbreviations used when presenting results.

C.1 Coupling Metrics

Package	C_a	C_e
karsch.lukas	0	0
karsch.lukas.audit	18	23
karsch.lukas.auth	16	3
karsch.lukas.config	6	0
karsch.lukas.courses	58	36
karsch.lukas.dev	0	92
karsch.lukas.exceptions	0	16
karsch.lukas.featureflags	7	0
karsch.lukas.lectures	251	271
karsch.lukas.stats	3	144
karsch.lukas.time	29	4
karsch.lukas.users	130	19
karsch.lukas.uuid	0	0

Table C.2: Coupling Metrics for CRUD (Packages)

Package	C_a	C_e
k.l	0	0
k.l.config	0	0
k.l.config.aspect	0	0
k.l.config.commandInterceptors	0	2
k.l.core.auth	7	3
k.l.core.exceptions	74	0
k.l.core.json	6	0
k.l.core.lookup	23	0
k.l.core.queries	10	5
k.l.core.uuid	5	2
k.l.dev	0	6
k.l.f.course.api	65	0
k.l.f.course.commands	13	17
k.l.f.course.exceptions	4	3
k.l.f.course.queries	0	21
k.l.f.course.web	0	33
k.l.f.enrollment.api	113	0
k.l.f.enrollment.command	7	104
k.l.f.enrollment.command.lookup	5	6
k.l.f.enrollment.command.lookup.credits	15	13
k.l.f.enrollment.exception	5	6
k.l.f.lectures.api	390	12
k.l.f.lectures.command	6	177
k.l.f.lectures.command.lookup.assessment	10	16
k.l.f.lectures.command.lookup.lecture	9	14
k.l.f.lectures.command.lookup.timeSlot	6	43
k.l.f.lectures.exceptions	4	2
k.l.f.lectures.queries	0	277
k.l.f.lectures.web	0	148
k.l.f.professor.api	18	0
k.l.f.professor.command	8	9
k.l.f.stats.api	23	0
k.l.f.stats.queries.credits	0	16
k.l.f.stats.queries.gradeHistory	0	43
k.l.f.stats.queries.grades	0	48
k.l.f.stats.web	0	32
k.l.f.student.api	21	0
k.l.f.student.command	2	9
k.l.f.student.command.lookup	25	5
k.l.f.users.web	0	19
k.l.infra.web	0	38

Table C.3: Coupling Metrics for CRUD (Packages)

C.2 Instability and Abstractness Metrics

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.1	0.5	1.0	1.0	0
ES-CQRS	0.0	0.3	0.6	1.0	1.0	0

Table C.4: Descriptive Statistics for Instability per package I

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.0	0.0	0.2	0.5	0
ES-CQRS	0.0	0.0	0.0	0.4	0.5	0

Table C.5: Descriptive Statistics for Abstractness per package A

Application	Min	P25	Median	P75	Max	Outliers
CRUD	0.0	0.2	0.5	0.9	1.0	0
ES-CQRS	0.0	0.1	0.4	1.0	1.0	0

Table C.6: Descriptive Statistics for Distance from the main sequence per package D

Package	A	I	D
karsch.lukas	0.0	1.0	1.0
karsch.lukas.audit	0.2	0.24	0.56
karsch.lukas.auth	0.0	0.84	0.16
karsch.lukas.config	0.0	1.0	0.0
karsch.lukas.courses	0.14	0.47	0.38
karsch.lukas.dev	0.0	0.0	1.0
karsch.lukas.exceptions	0.0	0.0	1.0
karsch.lukas.featureflags	0.0	1.0	0.0
karsch.lukas.lectures	0.24	0.24	0.52
karsch.lukas.stats	0.0	0.02	0.98
karsch.lukas.time	0.0	0.88	0.12
karsch.lukas.users	0.22	0.65	0.13
karsch.lukas.uuid	0.5	0.5	1.0

Table C.7: Abstractness A, Instability I and Distance from the Main Sequence D metrics for CRUD (Packages)

Package	A	I	D
k.l	0.0	1.0	1.0
k.l.config	0.0	1.0	1.0
k.l.config.aspect	0.0	1.0	1.0
k.l.config.commandInterceptors	0.0	0.0	1.0
k.l.core.auth	0.0	0.7	0.3
k.l.core.exceptions	0.2	0.8	0.0
k.l.core.json	0.0	1.0	0.0
k.l.core.lookup	0.0	1.0	0.0
k.l.core.queries	0.5	0.17	0.33
k.l.core.uuid	0.5	0.21	0.29
k.l.dev	0.0	0.0	1.0
k.l.f.course.api	0.0	1.0	0.0
k.l.f.course.commands	0.33	0.1	0.57
k.l.f.course.exceptions	0.0	0.57	0.43
k.l.f.course.queries	0.33	0.33	1.0
k.l.f.course.web	0.0	0.0	1.0
k.l.f.enrollment.api	0.0	1.0	0.0
k.l.f.enrollment.command	0.0	0.06	0.94
k.l.f.enrollment.command.lookup	0.4	0.05	0.55
k.l.f.enrollment.command.lookup.credits	0.4	0.14	0.46
k.l.f.enrollment.exception	0.0	0.45	0.55
k.l.f.lectures.api	0.0	0.97	0.03
k.l.f.lectures.command	0.0	0.03	0.97
k.l.f.lectures.command.lookup.assessment	0.4	0.02	0.62
k.l.f.lectures.command.lookup.lecture	0.4	0.01	0.61
k.l.f.lectures.command.lookup.timeSlot	0.4	0.28	0.88
k.l.f.lectures.exceptions	0.0	0.67	0.33
k.l.f.lectures.queries	0.42	0.42	1.0
k.l.f.lectures.web	0.0	0.0	1.0
k.l.f.professor.api	0.0	1.0	0.0
k.l.f.professor.command	0.33	0.14	0.53
k.l.f.stats.api	0.0	1.0	0.0
k.l.f.stats.queries.credits	0.4	0.4	1.0
k.l.f.stats.queries.gradeHistory	0.4	0.4	1.0
k.l.f.stats.queries.grades	0.38	0.38	1.0
k.l.f.stats.web	0.0	0.0	1.0
k.l.f.student.api	0.0	1.0	0.0
k.l.f.student.command	0.0	0.18	0.82
k.l.f.student.command.lookup	0.4	0.43	0.17
k.l.f.users.web	0.0	0.0	1.0
k.l.infra.web	0.0	0.0	1.0

Table C.8: Abstractness A, Instability I and Distance from the Main Sequence D metrics for ES-CQRS (Packages)

C.3 Dependency Metrics

Class	Dpt	Dpt*	Dcy	Dcy*	PDpt	PDcy
k.l.CrudApplication	1	1	0	0	1	0
k.l.audit.AuditContext	2	43	0	0	1	0
k.l.audit.AuditEntityListener	1	41	9	12	1	4
k.l.audit.AuditHelper	3	44	0	0	1	0
k.l.audit.AuditLogEntry	5	45	0	0	2	0
k.l.audit.AuditService	2	6	4	4	2	1
k.l.audit.AuditableEntity	9	41	1	12	4	1
k.l.audit.CollectionWithIdSerializer	1	44	1	1	1	1
k.l.audit.DynamicIdSerializer	2	45	0	0	1	0
k.l.audit.IdPropertySerializerModifier	1	43	2	2	1	1
k.l.audit.IdSerializationModule	1	42	1	3	1	1
k.l.audit.JpaAuditingConfiguration	0	0	0	0	0	0
k.l.auth.CustomAuthFilter	0	0	2	3	0	1
k.l.auth.NotAuthenticatedException	3	3	0	0	2	0
k.l.config.SpringContext	1	42	0	0	1	0
k.l.courses.CourseDtoMapper	2	5	4	18	2	3
k.l.courses.CourseEntity	13	39	1	13	7	1
k.l.courses.CoursesController	0	0	8	28	0	5
k.l.courses.CoursesNotFoundException	2	4	0	0	2	0
k.l.courses.CoursesService	1	1	8	23	1	4
k.l.courses.SimpleCourseDtoMapper	2	7	3	16	2	3
k.l.dev.SeedDataRunner	0	0	16	30	0	6
k.l.exceptions.GlobalExceptionHandler	0	0	1	1	0	1
k.l.featureflags.Feature	3	3	0	0	2	0
k.l.featureflags.FeatureFlagService	2	2	1	1	2	1
k.l.featureflags.FeaturesEndpoint	0	0	2	2	0	1
k.l.le.AlreadyEnrolledException	1	2	0	0	1	0
k.l.le.AssessmentGradeEntity	6	8	3	24	4	3
k.l.le.EnrollmentEntity	6	33	3	23	4	3
k.l.le.LectureAssessmentEntity	12	33	4	23	5	4
k.l.le.LectureAssessmentMapper	1	3	4	28	1	4
k.l.le.LectureDetailDtoMapper	1	2	9	40	1	6
k.l.le.LectureDtoMapper	1	2	6	32	1	6
k.l.le.LectureEntity	18	33	10	23	7	6
k.l.le.LectureNotFoundException	1	2	0	0	1	0
k.l.le.LectureNotOpenForEnrollmentException	1	2	1	1	1	1
k.l.le.LectureWaitlistEntryComparator	2	3	2	24	1	2
k.l.le.LectureWaitlistEntryEntity	6	33	3	23	1	3
k.l.le.LecturesController	0	0	17	84	0	5
k.l.le.LecturesService	1	1	41	79	1	8
k.l.le.SimpleLectureDtoMapper	2	4	4	26	2	4
k.l.le.WaitlistedStudentMapper	2	3	4	28	1	4
k.l.stats.GradedAssessmentDtoMapper	1	4	4	27	1	3
k.l.stats.StatsController	0	0	8	45	0	3
k.l.stats.StatsService	2	3	20	40	2	7
k.l.time.TimeSlotMapper	4	5	3	3	1	3

k.l.time.TimeSlotValueObject	9	37	0	0	5	0
k.l.time.TimeSlotValueObjectComparator	2	35	1	1	2	1
k.l.users.ProfessorDtoMapper	2	4	3	26	1	3
k.l.users.ProfessorEntity	9	33	2	23	6	2
k.l.users.StudentDtoMapper	2	4	3	26	1	3
k.l.users.StudentEntity	16	33	2	23	7	2
k.l.users.StudentNotFoundException	1	4	0	0	1	0
k.l.users.UsersController	0	0	5	31	0	2
k.l.users.UsersService	1	1	6	28	1	1
k.l.uuid.UuidV7Generator	1	1	1	1	1	1

Table C.9: Dependency Metrics for CRUD (Classes)

Class	Dpt	Dpt*	Dcy	Dcy*	PDpt	PDcy
k.l.EsCqrsApplication	1	1	0	0	1	0
k.l.config.JacksonConfiguration	0	0	0	0	0	0
k.l.config.RetryConfig	0	0	0	0	0	0
k.l.config.aspect.LoggingAspect	0	0	0	0	0	0
k.l.config.aspect.RetryLoggingAspect	0	0	0	0	0	0
k.l.config.commandInterceptors.AggregateNotFoundInterceptor	0	0	1	1	0	1
k.l.core.auth.CustomAuthFilter	0	0	2	3	0	1
k.l.core.auth.NotAuthenticatedException	2	2	0	0	2	0
k.l.core.exceptions.DomainException	8	16	1	1	3	1
k.l.core.exceptions.ErrorDetails	7	31	0	0	5	0
k.l.core.exceptions.MissingResourceException	4	20	1	1	3	1
k.l.core.exceptions.NotAllowedException	6	13	1	1	3	1
k.l.core.exceptions.QueryException	3	3	0	0	3	0
k.l.core.json.Defaults	2	4	0	0	1	0
k.l.core.lookup.TimeSlotEmbeddable	7	13	0	0	4	0
k.l.core.lookup.TimeSlotEmbeddableComparator	2	4	1	1	1	1
k.l.core.queries.CourseMapper	2	3	3	3	2	2
k.l.core.uuid.UuidProviderImpl	0	0	2	2	0	2
k.l.dev.SeedDataRunner	0	0	5	6	0	5
k.l.f.course.api.CourseCreatedEvent	7	7	0	0	4	0
k.l.f.course.api.CreateCourseCommand	6	7	0	0	5	0
k.l.f.course.api.FindAllCoursesQuery	3	3	0	0	2	0
k.l.f.course.api.FindCourseByIdQuery	3	3	0	0	2	0
k.l.f.course.commands.CourseAggregate	2	2	4	6	1	3
k.l.f.course.commands.CourseLookupProjector	0	0	4	9	0	2
k.l.f.course.commands.CourseValidator	0	0	3	3	0	1
k.l.f.course.commands.CoursesLookupJpaEntity	3	3	0	0	1	0
k.l.f.course.exceptions.MissingCoursesException	4	14	1	2	2	1
k.l.f.course.queries.CourseProjectionEntity	3	3	1	1	1	1
k.l.f.course.queries.CourseProjector	1	1	8	9	1	4
k.l.f.course.web.CoursesController	0	0	10	12	0	7
k.l.f.e.api.AssignGradeCommand	5	7	0	0	4	0
k.l.f.e.api.AwardCreditsCommand	3	6	0	0	2	0
k.l.f.e.api.CreditsAwardedEvent	5	7	0	0	4	0
k.l.f.e.api.EnrollmentCreatedEvent	7	9	0	0	5	0

k.l.f.e.api.GradeAssignedEvent	3	6	0	0	3	0
k.l.f.e.api.GradeUpdatedEvent	3	6	0	0	3	0
k.l.f.e.api.UpdateGradeCommand	3	5	0	0	2	0
k.l.f.e.c.AwardCreditsSaga	1	1	7	9	1	4
k.l.f.e.c.EnrollmentAggregate	3	3	10	16	3	4
k.l.f.e.c.EnrollmentCommandHandler	0	0	18	31	0	12
k.l.f.e.c.lookup.EnrollmentLookupEntity	5	6	0	0	2	0
k.l.f.e.c.lookup.EnrollmentLookupProjector	0	0	4	19	0	3
k.l.f.e.c.lookup.EnrollmentValidator	0	0	3	3	0	1
k.l.f.e.c.lookup.credits.StudentCreditsLookupProjectionEntity	3	3	0	0	1	0
k.l.f.e.c.lookup.credits.StudentCreditsLookupProjector	0	0	5	20	0	4
k.l.f.e.c.lookup.credits.StudentCreditsValidator	0	0	4	4	0	2
k.l.f.e.exception.AssessmentNotFoundException	1	1	1	2	1	1
k.l.f.e.exception.MissingGradeException	1	4	1	2	1	1
k.l.f.e.exception.StudentNotEnrolledException	1	1	1	2	1	1
k.l.f.le.api.AddAssessmentCommand	5	14	2	2	4	2
k.l.f.le.api.AdvanceLectureLifecycleCommand	6	14	1	1	4	1
k.l.f.le.api.AssessmentAddedEvent	7	14	2	2	6	2
k.l.f.le.api.AssignTimeSlotsToLectureCommand	3	12	1	1	2	1
k.l.f.le.api.ConfirmStudentEnrollmentCommand	4	13	0	0	1	0
k.l.f.le.api.CreateEnrollmentCommand	3	6	0	0	2	0
k.l.f.le.api.CreateLectureCommand	6	15	1	1	5	1
k.l.f.le.api.DisenrollStudentCommand	3	12	0	0	2	0
k.l.f.le.api.EnrollStudentCommand	5	14	0	0	4	0
k.l.f.le.api.EnrollmentStatusQuery	2	2	0	0	2	0
k.l.f.le.api.EnrollmentStatusUpdate	2	2	1	1	2	1
k.l.f.le.api.FindLectureByIdQuery	3	3	0	0	2	0
k.l.f.le.api.GetLectureWaitlistQuery	2	2	0	0	2	0
k.l.f.le.api.GetLecturesForStudentQuery	2	2	0	0	2	0
k.l.f.le.api.LectureCreatedEvent	10	13	2	2	5	1
k.l.f.le.api.LectureLifecycleAdvancedEvent	8	14	1	1	5	1
k.l.f.le.api.StudentDisenrolledEvent	4	13	0	0	2	0
k.l.f.le.api.StudentEnrolledEvent	5	13	0	0	2	0
k.l.f.le.api.StudentEnrollmentApprovedEvent	5	13	0	0	1	0
k.l.f.le.api.StudentRemovedFromWaitlistEvent	4	13	0	0	2	0
k.l.f.le.api.StudentWaitlistedEvent	5	13	0	0	2	0
k.l.f.le.api.TimeSlotsAssignedEvent	4	12	1	1	3	1
k.l.f.le.api.WaitlistClearedEvent	4	13	0	0	2	0
k.l.f.le.c.AssessmentValueObject	1	11	2	2	1	2
k.l.f.le.c.EnrollmentApprovalSaga	1	1	5	5	1	3
k.l.f.le.c.LectureAggregate	4	10	32	36	4	11
k.l.f.le.c.lookup.assessment.AssessmentLookupEntity	6	7	2	2	2	2
k.l.f.le.c.lookup.assessment.AssessmentLookupProjector	0	0	6	40	0	5
k.l.f.le.c.lookup.assessment.AssessmentValidator	0	0	3	5	0	1
k.l.f.le.c.lookup.lecture.LectureLookupEntity	5	5	1	1	2	1
k.l.f.le.c.lookup.lecture.LectureLookupProjector	0	0	6	39	0	3
k.l.f.le.c.lookup.lecture.LectureValidator	0	0	3	4	0	1
k.l.f.le.c.lookup.timeSlot.LectureTimeSlotProjector	0	0	8	41	0	5
k.l.f.le.c.lookup.timeSlot.LectureTimeslotLookupEntity	3	3	2	2	1	1
k.l.f.le.c.lookup.timeSlot.TimeSlotValidator	0	0	9	12	0	5

k.l.f.le.exceptions.LectureNotFoundException	2	2	1	2	2	1
k.l.f.le.queries.CourseProjectionEntity	2	2	1	1	1	1
k.l.f.le.queries.LectureDetailProjectionEntity	2	2	1	1	1	1
k.l.f.le.queries.LectureProjector	0	0	39	44	0	13
k.l.f.le.queries.ProfessorProjectionEntity	2	2	0	0	1	0
k.l.f.le.queries.StudentLecturesProjectionEntity	2	2	1	1	1	1
k.l.f.le.queries.StudentLecturesProjector	0	0	14	22	0	4
k.l.f.le.queries.StudentProjectionEntity	2	2	0	0	1	0
k.l.f.le.web.LecturesController	0	0	31	42	0	8
k.l.f.professor.api.CreateProfessorCommand	5	6	0	0	5	0
k.l.f.professor.api.ProfessorCreatedEvent	3	3	0	0	2	0
k.l.f.professor.c.ProfessorAggregate	1	1	2	2	1	1
k.l.f.professor.c.ProfessorLookupEntity	2	3	0	0	1	0
k.l.f.professor.c.ProfessorLookupTable	0	0	4	5	0	2
k.l.f.professor.c.ProfessorValidator	0	0	2	3	0	1
k.l.f.stats.api.GetCreditsForStudentQuery	2	2	0	0	2	0
k.l.f.stats.api.GetGradeHistoryQuery	2	2	0	0	2	0
k.l.f.stats.api.GetGradesForStudentQuery	2	2	0	0	2	0
k.l.f.stats.queries.credits.StudentCreditsProjectionEntity	2	2	0	0	1	0
k.l.f.stats.queries.credits.StudentCreditsProjector	0	0	7	8	0	6
k.l.f.stats.queries.gradeHistory.AssessmentProjectionEntity	2	2	0	0	1	0
k.l.f.stats.queries.gradeHistory.EnrollmentProjectionEntity	2	2	0	0	1	0
k.l.f.stats.queries.gradeHistory.GradeHistoryProjector	0	0	11	13	0	5
k.l.f.stats.queries.grades.AssessmentProjectionEntity	2	2	1	1	1	1
k.l.f.stats.queries.grades.GradedAssessmentId	3	3	0	0	1	0
k.l.f.stats.queries.grades.GradedAssessmentProjectionEntity	2	2	2	2	1	2
k.l.f.stats.queries.grades.SimpleCourseProjectionEntity	2	2	0	0	1	0
k.l.f.stats.queries.grades.SimpleLectureProjectionEntity	2	2	0	0	1	0
k.l.f.stats.queries.grades.StudentGradesProjectionEntity	2	2	1	1	1	1
k.l.f.stats.queries.grades.StudentGradesProjectionEntityId	3	3	0	0	1	0
k.l.f.stats.queries.grades.StudentGradesProjector	0	0	22	25	0	7
k.l.f.stats.web.StatsController	0	0	9	14	0	4
k.l.f.student.api.CreateStudentCommand	5	6	0	0	5	0
k.l.f.student.api.StudentCreatedEvent	3	3	0	0	3	0
k.l.f.student.c.StudentAggregate	1	1	2	2	1	1
k.l.f.student.c.lookup.StudentLookupEntity	6	15	0	0	2	0
k.l.f.student.c.lookup.StudentLookupProjector	0	0	4	5	0	3
k.l.f.student.c.lookup.StudentValidator	0	0	3	3	0	1
k.l.f.users.web.UsersController	0	0	7	7	0	5
k.l.infra.web.GlobalExceptionHandler	0	0	3	3	0	2

Table C.10: Dependency Metrics for ES-CQRS (Classes)

C.4 MOOD

Metric	CRUD	ES-CQRS
AHF	97.71%	95.55%
AIF	28.35%	8.17%
CF	11.56%	3.18%
MHF	28.71%	44.53%
MIF	16.26%	0.76%
PF	362.50%	185.71%

Table C.11: MOOD metrics

C.5 Chidamber Kemerer Metrics

Class	CBO	DIT	LCOM	NOC	RFC	WMC
k.l.CrudApplication	1	1	1	0	2	1
k.l.audit.AuditContext	2	1	1	0	12	7
k.l.audit.AuditEntityListener	9	1	1	0	41	13
k.l.audit.AuditHelper	3	1	1	0	2	1
k.l.audit.AuditLogEntry	5	1	9	0	19	0
k.l.audit.AuditLogRepository	0	0	0	0	0	0
k.l.audit.AuditService	6	1	3	0	20	10
k.l.audit.AuditableEntity	9	1	1	8	2	0
k.l.audit.CollectionWithIdSerializer	2	2	2	0	8	6
k.l.audit.DynamicIdSerializer	2	2	1	0	12	3
k.l.audit.IdPropertySerializerModifier	3	2	1	0	11	8
k.l.audit.IdSerializationModule	2	3	0	0	2	1
k.l.audit.JpaAuditingConfiguration	0	1	0	0	0	0
k.l.auth.CustomAuthFilter	2	1	1	0	3	1
k.l.auth.NotAuthenticatedException	3	7	0	0	3	2
k.l.config.SpringContext	1	1	1	0	3	2
k.l.courses.CourseDtoMapper	6	1	1	0	10	1
k.l.courses.CourseEntity	14	2	9	0	20	0
k.l.courses.CoursesController	8	1	1	0	14	3
k.l.courses.CoursesNotFoundException	2	7	0	0	3	1
k.l.courses.CoursesRepository	0	0	0	0	1	0
k.l.courses.CoursesService	9	1	1	0	26	3
k.l.courses.SimpleCourseDtoMapper	5	1	1	0	6	1
k.l.dev.SeedDataRunner	16	1	1	0	55	10
k.l.exceptions.GlobalExceptionHandler	1	1	1	0	12	5
k.l.featureflags.Feature	3	0	2	0	2	0
k.l.featureflags.FeatureFlagService	3	1	1	0	9	4
k.l.featureflags.FeaturesEndpoint	2	1	1	0	8	3
k.l.le.AlreadyEnrolledException	1	7	0	0	3	1
k.l.le.AssessmentGradeEntity	9	2	8	0	18	0
k.l.le.AssessmentGradeRepository	0	0	0	0	3	0
k.l.le.EnrollmentEntity	7	2	4	0	11	1
k.l.le.EnrollmentRepository	0	0	0	0	6	0
k.l.le.LectureAssessmentEntity	15	2	8	0	18	0
k.l.le.LectureAssessmentMapper	5	1	1	0	7	1
k.l.le.LectureAssessmentRepository	0	0	0	0	2	0
k.l.le.LectureDetailDtoMapper	10	1	1	0	17	1
k.l.le.LectureDtoMapper	7	1	1	0	12	1
k.l.le.LectureEntity	24	2	13	0	38	2
k.l.le.LectureNotFoundException	1	7	0	0	3	1
k.l.le.LectureNotOpenForEnrollmentException	2	7	0	0	3	1
k.l.le.LectureWaitlistEntryComparator	4	1	1	0	6	2
k.l.le.LectureWaitlistEntryEntity	8	2	4	0	11	1
k.l.le.LectureWaitlistEntryRepository	0	0	0	0	2	0
k.l.le.LecturesController	17	1	1	0	35	19
k.l.le.LecturesRepository	0	0	0	0	4	0
k.l.le.LecturesService	42	1	1	0	126	33
k.l.le.SimpleLectureDtoMapper	6	1	1	0	6	1
k.l.le.WaitlistedStudentMapper	6	1	1	0	6	1
k.l.stats.GradedAssessmentDtoMapper	5	1	1	0	8	1
k.l.stats.StatsController	8	1	1	0	12	5
k.l.stats.StatsService	22	1	1	0	75	16
k.l.time.TimeSlotMapper	7	1	1	0	5	1
k.l.time.TimeSlotValueObject	9	2	3	0	4	0

k.l.time.TimeSlotValueObjectComparator	3	1	1	0	6	3
k.l.users.ProfessorDtoMapper	5	1	1	0	5	1
k.l.users.ProfessorEntity	10	2	5	0	12	0
k.l.users.ProfessorRepository	0	0	0	0	0	0
k.l.users.StudentDtoMapper	5	1	1	0	5	1
k.l.users.StudentEntity	17	2	6	0	15	0
k.l.users.StudentNotFoundException	1	7	0	0	3	1
k.l.users.StudentRepository	0	0	0	0	0	0
k.l.users.UsersController	5	1	1	0	7	2
k.l.users.UsersService	7	1	2	0	14	2
k.l.uuid.GeneratedUuidV7	0	0	0	0	0	0
k.l.uuid.UuidV7Generator	2	2	1	0	2	1

Table C.12: CK Metrics for CRUD architecture.

Class	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
k.l.EsCqrsApplication	1	1	1	0	2	1
k.l.config.JacksonConfiguration	0	1	1	0	9	1
k.l.config.RetryConfig	0	1	0	0	0	0
k.l.config.aspect.LoggingAspect	0	1	2	0	18	3
k.l.config.aspect.RetryLoggingAspect	0	1	2	0	11	4
k.l.config.commandInterceptors.AggregateNotFoundInterceptor	1	1	1	0	3	1
k.l.core.auth.CustomAuthFilter	2	1	1	0	3	1
k.l.core.auth.NotAuthenticatedException	2	7	0	0	3	2
k.l.core.exceptions.DomainException	9	7	0	1	2	1
k.l.core.exceptions.ErrorDetails	7	0	2	0	2	0
k.l.core.exceptions.MissingResourceException	5	7	0	4	2	1
k.l.core.exceptions.NotAllowedException	7	7	0	0	2	1
k.l.core.exceptions.QueryException	3	4	0	0	2	1
k.l.core.json.Defaults	2	1	0	0	0	0
k.l.core.lookup.TimeSlotEmbeddable	7	2	3	0	4	0
k.l.core.lookup.TimeSlotEmbeddableComparator	3	1	1	0	6	3
k.l.core.queries.CourseMapper	5	1	1	0	13	1
k.l.core.queries.ICourseProjectionEntity	0	0	0	0	6	0
k.l.core.uuid.UuidProvider	0	0	0	0	1	0
k.l.core.uuid.UuidProviderImpl	2	1	1	0	2	1
k.l.dev.SeedDataRunner	5	1	1	0	16	1
k.l.f.course.api.CourseCreatedEvent	7	2	6	0	7	0
k.l.f.course.api.CreateCourseCommand	6	2	6	0	7	0
k.l.f.course.api.FindAllCoursesQuery	3	1	0	0	0	0
k.l.f.course.api.FindCourseByIdQuery	3	2	1	0	2	0
k.l.f.course.commands.CourseAggregate	6	1	1	0	15	4
k.l.f.course.commands.CourseLookupProjector	4	1	1	0	9	1
k.l.f.course.commands.CourseLookupRepository	0	0	0	0	2	0
k.l.f.course.commands.CourseValidator	3	1	1	0	16	5
k.l.f.course.commands.CoursesLookupJpaEntity	3	1	5	0	10	0
k.l.f.course.commands.ICourseValidator	0	0	0	0	5	0
k.l.f.course.exceptions.MissingCoursesException	5	8	0	0	4	2
k.l.f.course.queries.CourseProjectionEntity	4	1	6	0	12	0
k.l.f.course.queries.CourseProjector	9	1	1	0	34	4
k.l.f.course.queries.CourseRepository	0	0	0	0	0	0
k.l.f.course.web.CoursesController	10	1	1	0	28	3
k.l.f.e.api.AssignGradeCommand	5	2	5	0	7	2
k.l.f.e.api.AwardCreditsCommand	3	2	2	0	3	0
k.l.f.e.api.CreditsAwardedEvent	5	2	4	0	5	0
k.l.f.e.api.EnrollmentCreatedEvent	7	2	4	0	5	0

k.l.f.e.api.GradeAssignedEvent	3	2	6	0	7	0
k.l.f.e.api.GradeUpdatedEvent	3	2	6	0	7	0
k.l.f.e.api.UpdateGradeCommand	3	2	5	0	7	2
k.l.f.e.c.AwardCreditsSaga	8	1	1	0	20	4
k.l.f.e.c.EnrollmentAggregate	13	1	1	0	46	12
k.l.f.e.c.EnrollmentCommandHandler	18	1	1	0	38	8
k.l.f.e.c.lookup.EnrollmentLookupEntity	5	1	3	0	8	0
k.l.f.e.c.lookup.EnrollmentLookupProjector	4	1	1	0	7	1
k.l.f.e.c.lookup.EnrollmentLookupRepository	0	0	0	0	2	0
k.l.f.e.c.lookup.EnrollmentValidator	3	1	1	0	6	2
k.l.f.e.c.lookup.IEnrollmentValidator	0	0	0	0	2	0
k.l.f.e.c.lookup.credits.IStudentCreditsValidator	0	0	0	0	2	0
k.l.f.e.c.lookup.credits.StudentCreditsLookupProjectionEntity	3	1	3	0	9	0
k.l.f.e.c.lookup.credits.StudentCreditsLookupProjector	5	1	1	0	15	2
k.l.f.e.c.lookup.credits.StudentCreditsLookupRepository	0	0	0	0	0	0
k.l.f.e.c.lookup.credits.StudentCreditsValidator	4	1	1	0	11	2
k.l.f.e.exception.AssessmentNotFoundException	2	8	0	0	3	1
k.l.f.e.exception.MissingGradeException	2	8	0	0	2	1
k.l.f.e.exception.StudentNotEnrolledException	2	8	0	0	3	1
k.l.f.le.api.AddAssessmentCommand	7	2	6	0	7	0
k.l.f.le.api.AdvanceLectureLifecycleCommand	7	2	3	0	4	0
k.l.f.le.api.AssessmentAddedEvent	9	2	6	0	7	0
k.l.f.le.api.AssignTimeSlotsToLectureCommand	4	2	3	0	4	0
k.l.f.le.api.ConfirmStudentEnrollmentCommand	4	2	2	0	3	0
k.l.f.le.api.CreateEnrollmentCommand	3	2	4	0	5	0
k.l.f.le.api.CreateLectureCommand	7	2	5	0	6	0
k.l.f.le.api.DisenrollStudentCommand	3	2	2	0	3	0
k.l.f.le.api.EnrollStudentCommand	5	2	2	0	3	0
k.l.f.le.api.EnrollmentStatusQuery	2	2	2	0	3	0
k.l.f.le.api.EnrollmentStatusUpdate	3	2	3	0	4	0
k.l.f.le.api.FindLectureByIdQuery	3	2	1	0	2	0
k.l.f.le.api.GetLectureWaitlistQuery	2	2	1	0	2	0
k.l.f.le.api.GetLecturesForStudentQuery	2	2	1	0	2	0
k.l.f.le.api.LectureCreatedEvent	12	2	6	0	7	0
k.l.f.le.api.LectureLifecycleAdvancedEvent	9	2	3	0	4	0
k.l.f.le.api.StudentDisenrolledEvent	4	2	2	0	3	0
k.l.f.le.api.StudentEnrolledEvent	5	2	2	0	3	0
k.l.f.le.api.StudentEnrollmentApprovedEvent	5	2	3	0	4	0
k.l.f.le.api.StudentRemovedFromWaitlistEvent	4	2	2	0	3	0
k.l.f.le.api.StudentWaitlistedEvent	5	2	3	0	4	0
k.l.f.le.api.TimeSlotsAssignedEvent	5	2	3	0	4	0
k.l.f.le.api.WaitlistClearedEvent	4	2	2	0	3	0
k.l.f.le.c.AssessmentValueObject	3	1	1	0	4	0
k.l.f.le.c.EnrollmentApprovalSaga	6	1	1	0	12	2
k.l.f.le.c.LectureAggregate	36	1	2	0	114	45
k.l.f.le.c.lookup.assessment.AssessmentLookupEntity	8	1	5	0	12	0
k.l.f.le.c.lookup.assessment.AssessmentLookupProjector	6	1	1	0	13	1
k.l.f.le.c.lookup.assessment.AssessmentLookupRepository	0	0	0	0	1	0
k.l.f.le.c.lookup.assessment.AssessmentValidator	3	1	1	0	5	2
k.l.f.le.c.lookup.assessment.IAssessmentValidator	0	0	0	0	2	0
k.l.f.le.c.lookup.lecture.ILectureValidator	0	0	0	0	1	0
k.l.f.le.c.lookup.lecture.LectureLookupEntity	6	1	5	0	13	0
k.l.f.le.c.lookup.lecture.LectureLookupProjector	6	1	1	0	21	3
k.l.f.le.c.lookup.lecture.LectureLookupRepository	0	0	0	0	0	0
k.l.f.le.c.lookup.lecture.LectureValidator	3	1	1	0	3	1
k.l.f.le.c.lookup.timeSlot.ITimeSlotValidator	0	0	0	0	1	0
k.l.f.le.c.lookup.timeSlot.LectureTimeSlotProjector	8	1	1	0	23	3
k.l.f.le.c.lookup.timeSlot.LectureTimeslotLookupEntity	5	1	2	0	7	0

k.l.f.le.c.lookup.timeSlot.LectureTimeslotLookupRepository	0	0	0	0	0	0
k.l.f.le.c.lookup.timeSlot.TimeSlotValidator	9	1	1	0	18	2
k.l.f.le.exceptions.LectureNotFoundException	3	8	0	0	2	1
k.l.f.le.queries.CourseProjectionEntity	3	1	6	0	12	0
k.l.f.le.queries.CourseRepository	0	0	0	0	0	0
k.l.f.le.queries.LectureDetailProjectionEntity	3	1	12	0	25	0
k.l.f.le.queries.LectureDetailRepository	0	0	0	0	0	0
k.l.f.le.queries.LectureProjector	39	1	2	0	133	21
k.l.f.le.queries.ProfessorProjectionEntity	2	1	4	0	9	0
k.l.f.le.queries.ProfessorRepository	0	0	0	0	0	0
k.l.f.le.queries.StudentLecturesProjectionEntity	3	1	5	0	13	0
k.l.f.le.queries.StudentLecturesProjector	14	1	1	0	64	10
k.l.f.le.queries.StudentLecturesRepository	0	0	0	0	1	0
k.l.f.le.queries.StudentProjectionEntity	2	1	4	0	10	0
k.l.f.le.queries.StudentRepository	0	0	0	0	0	0
k.l.f.le.web.LecturesController	31	1	1	0	66	23
k.l.f.professor.api.CreateProfessorCommand	5	2	3	0	4	0
k.l.f.professor.api.ProfessorCreatedEvent	3	2	3	0	4	0
k.l.f.professor.c.IProfessorValidator	0	0	0	0	1	0
k.l.f.professor.c.ProfessorAggregate	3	1	1	0	9	2
k.l.f.professor.c.ProfessorLookupEntity	2	1	2	0	6	0
k.l.f.professor.c.ProfessorLookupRepository	0	0	0	0	0	0
k.l.f.professor.c.ProfessorLookupTable	4	1	1	0	5	1
k.l.f.professor.c.ProfessorValidator	2	1	1	0	3	1
k.l.f.stats.api.GetCreditsForStudentQuery	2	2	1	0	2	0
k.l.f.stats.api.GetGradeHistoryQuery	2	2	4	0	5	0
k.l.f.stats.api.GetGradesForStudentQuery	2	2	1	0	2	0
k.l.f.stats.queries.credits.StudentCreditsProjectionEntity	2	1	2	0	6	0
k.l.f.stats.queries.credits.StudentCreditsProjectionRepository	0	0	0	0	0	0
k.l.f.stats.queries.credits.StudentCreditsProjector	7	1	1	0	23	4
k.l.f.stats.queries.gradeHistory.AssessmentProjectionEntity	2	1	2	0	6	0
k.l.f.stats.queries.gradeHistory.AssessmentProjectionRepository	0	0	0	0	0	0
k.l.f.stats.queries.gradeHistory.EnrollmentProjectionEntity	2	1	3	0	8	0
k.l.f.stats.queries.gradeHistory.EnrollmentProjectionRepository	0	0	0	0	1	0
k.l.f.stats.queries.gradeHistory.GradeHistoryProjector	11	1	1	0	48	5
k.l.f.stats.queries.grades.AssessmentProjectionEntity	3	1	4	0	10	0
k.l.f.stats.queries.grades.AssessmentProjectionRepository	0	0	0	0	1	0
k.l.f.stats.queries.grades.GradedAssessmentId	3	1	3	0	9	0
k.l.f.stats.queries.grades.GradedAssessmentProjectionEntity	4	1	7	0	16	0
k.l.f.stats.queries.grades.GradedAssessmentProjectionRepository	0	0	0	0	1	0
k.l.f.stats.queries.grades.SimpleCourseProjectionEntity	2	1	3	0	8	0
k.l.f.stats.queries.grades.SimpleCourseProjectionRepository	0	0	0	0	0	0
k.l.f.stats.queries.grades.SimpleLectureProjectionEntity	2	1	3	0	8	0
k.l.f.stats.queries.grades.SimpleLectureProjectionRepository	0	0	0	0	0	0
k.l.f.stats.queries.grades.StudentGradesProjectionEntity	3	1	3	0	8	0
k.l.f.stats.queries.grades.StudentGradesProjectionEntityId	3	1	3	0	9	0
k.l.f.stats.queries.grades.StudentGradesProjector	22	1	1	0	78	16
k.l.f.stats.queries.grades.StudentGradesRepository	0	0	0	0	1	0
k.l.f.stats.web.StatsController	9	1	1	0	13	5
k.l.f.student.api.CreateStudentCommand	5	2	4	0	5	0
k.l.f.student.api.StudentCreatedEvent	3	2	4	0	5	0
k.l.f.student.c.StudentAggregate	3	1	1	0	11	3
k.l.f.student.c.lookup.IStudentValidator	0	0	0	0	3	0
k.l.f.student.c.lookup.StudentLookupEntity	6	1	2	0	6	0
k.l.f.student.c.lookup.StudentLookupProjector	4	1	1	0	6	1
k.l.f.student.c.lookup.StudentLookupRepository	0	0	0	0	0	0
k.l.f.student.c.lookup.StudentValidator	3	1	1	0	7	3
k.l.f.users.web.UsersController	7	1	1	0	13	2

k.l.infra.web.GlobalExceptionHandler	3	1	2	0	20	13

Table C.13: CK Metrics for ES-CQRS architecture.

Appendix D

Nutzung von KI-Tools

Kapitel / Codemodul	**Tool(s)**	**Beschreibung und Begründung Einsatzzweck**
Alle Kapitel	Gemini	LaTeX Syntax-Unterstützung bei Tabellen, figures, listings
Alle Kapitel	DeepL Write	Grammatikalische und stilistische Überarbeitung von Textsegmenten
Alle Kapitel	DeepL Translate	Übersetzung von Wörtern und Formulierungen
Kapitel 7 (Results)	Gemini	Hilfe bei der Visualisierung & statistischen Auswertung der Ergebnisse (matplotlib, pandas, seaborn)

Table D.1: AI tools used throughout the thesis