Bachelor's Thesis in Computer Science and Media

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

Lukas Karsch

45259

Hochschule der Medien Stuttgart

Submitted on 2026/03/02

to obtain the degree of Bachelor of Science

Main Supervisor: Prof. Dr. Tobias Jordine

Secondary Supervisor: Felix Messner

Ehrenwörtliche Erklärung

Contents

1 Introduction

1.1 Motivation

1.2 Research question(s)

1.3 Goals and non goals

1.4 Structure of the paper

2 Basics

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 (Jacobs & Walsh 2004).

Web APIs are interfaces that enable applications to communicate. They use HTTP as a network-based API (Fielding 2000, p. 138). Modern APIs typically follow REST principles. REST stands for "Representational State Transfer" and describes an architectural style for distributed hypermedia systems (Fielding 2000, p. 76).

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

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

- Book; ID=1; URI=`http://example.com/books/1`
- Book; ID=2; URI=`http://example.com/books/2`
- Author; ID=100; URI=`http://example.com/authors/100`

The "Hypermedia as the engine of application state (HATEOAS)" principle states that resources should be linked to each other. Clients should be able to control the application by following a series of links provided by the server (Tilkov 2007).

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

Resources are decoupled from their representations. Clients can request different representations of a resource, depending on their needs (Tilkov 2007): a web browser might request HTML, while another server or application might request XML or JSON.

2.2 Layered Architecture Foundations

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

- Presentation: Handles requests and displays data in a user interface or by turning it into representations (e.g. JSON)
- Business: Encapsulates business logic
- Persistence: Persists data by interacting with the underlying persistence technologies (e.g. SQL databases)
- Database

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

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

The database responds with raw data, which is turned into a Data Access Object 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. (Richards 2015)

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 (2003) describes this as an object-oriented *antipattern*.

2.3 Domain Driven Design

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

The core idea of DDD is that the primary focus of a software project should not be the underlying technologies, but the domain. The domain is the topic with which a software concerns itself. The software design should be based on a model that closely matches the domain and reflects a deep understanding of business requirements. (Evans 2004, 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. (Evans 2004, 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 artificial and forms a selective abstraction which should be chosen for its utility. (Evans 2004, pp. 12, 13)

While Layered Architecture organizes code into technical tiers and is typically built on Anemic Domain Models, often resulting in the *big ball of mud* antipattern (Richards 2015, p. V), DDD demands a Rich Domain Model where objects incorporate both data and the behavior or rules that govern that data. The code is structured semantically into bounded context and modules which are chosen to tell the "story" of a system rather than its technicalities. (Evans 2004, p. 80)

Entities (also known as reference objects) are domain elements fundamentally defined by a thread of continuity and identity rather than their specific attributes. Entities must be distinguishable from other entities, even if they share the same characteristics. To ensure consistency and identity, a unique identifier is assigned to entities. This identifier is immutable throughout the object's life. (Evans 2004, pp. 65–69)

Value Objects are elements that describe the nature or state of something and have no conceptual identity of their own. They are interesting only for their characteristics. While two entities with the same characteristics are considered as different from each other, the system does not care about "identity" of a value object, since only its characteristics are relevant. Value objects should be used to encapsulate concepts, such as using an "Address" object instead of distinct "Street" and "City" attributes. Value objects should be immutable. They are never modified, instead they are replaced entirely when a new value is required. (Evans 2004, pp. 70–72)

Using a Rich Domain Model does not mean that there should be no layers, the opposite is the case. Evans (2004) advocates for using layers in domain driven designs. He proposes the following layers: (Evans 2004, p. 53)

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

Evans (2004, 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 Anemic Domain Model, it is important to use services only when necessary. Services are not allowed to strip the entities and value objects in the domain of behavior. According to Evans, a good domain service has the following characteristics:

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

2.4 CRUD architecture

Layered architectures are the standard for data-oriented enterprise applications. These applications mostly follow a CRUD architecture. CRUD is an acronym coined by Martin (1983) that stands for "Create, Read, Update, Delete". These four actions can be applied to any record of data.

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

ACID (Atomicity, Consistency, Isolation, Durability) are an important feature of CRUD applications. They can be guaranteed using transactions, ensuring that data stays consistent and operations are atomic. (Bernstein & Newcomer 2009, pp. 10, 11)

2.5 CQRS Architecture

Command Query Responsibility Segregation (CQRS) is an architectural pattern based on the fundamental idea that the models used to update information should be separate from the models used to read information. This approach originated as an extension of Bertrand Meyer's Command And Query Separation (CQS) principle, which states that a method should either perform an action (a command) or return data (a query), but never both. (Meyer 2006, 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. (Young 2010, p. 17)

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

- Command Side: Deals with data changes and captures user intent. Commands tell the system what needs to be done rather than overwriting previous state. Commands are validated by the system before execution and can be rejected. (Young 2010, 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. (Young 2010, p. 20)

In a CQRS architecture, the read side typically updates its data asynchronously by consuming notifications or events generated by the write side.

Because the models for updating and reading information are strictly separated, a synchronization mechanism is required to ensure the read store eventually reflects the changes made by commands. This usually leads to stale data on the read side.

Each read service independently updates its model by consuming notifications or events published by the write side, allowing the read model to store optimized, denormalized views on the data. (Young 2010, p. 23)

2.6 (Eventual) Consistency

Gray et al. (1996) explain that large-scale systems become unstable if they are held consistent at all times according to ACID principles. This is mostly due to the large amount of communication necessary to handle atomic transactions in distributed systems. To address these issues, modern distributed systems often adopt the BASE (Basically Available, Soft State, Eventual Consistency) model which explicitly trades off isolation and strong consistency for availability. Eventually consistent systems are allowed to exist in a so-called "soft state" which eventually converges through the use of synchronization mechanisms over time rather than being strongly consistent at all times. (Braun et al. 2021; Vogels 2009) This creates an inconsistency window in which data is not consistent across the system. During this window, stale data may be read. (Vogels 2009)

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. (Michelson 2006) This approach allows for a high degree of loose coupling, as the system publishing an event does not need to know about the recipient(s) or how they will react. These architectures offer excellent horizontal scalability and resilience, as individual system components can fail or be updated without bringing down the entire network. (Fowler 2005)

Event Sourcing is an architectural pattern within the landscape of event driven architectures. Event-sourced systems ensure that *all* changes to a system's state are captured and stored as an ordered sequence of domain events. (Fowler 2005) 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. (Fowler 2017; Lima et al. 2021)

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. (Fowler 2005) 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. (Malyi & Serdyuk 2024) This separation of concerns is frequently managed by pairing event sourcing with the Command Query Responsibility Segregation (CQRS) pattern, which physically divides the data structures used for reading from those used for writing state changes. (Young 2010, p. 50)

Because every action taken on the system is stored, a number of facilities can be built on top of the event log: Temporal queries can be made, which determine the exact state of the application at any point in time. The event log acts as an immutable audit trail, making Event Sourcing architectures highly valuable for systems like accounting applications. (Fowler 2005)

2.8 Traceability and auditing in IT systems

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. (Committee on National Security Systems 2010) In information security, the audit log stores a record of system activities, enabling the reconstruction of events. (ATIS Committee 2013) 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. (Joint Task Force Interagency Working Group 2020, p. 266)

Traceability and auditing are legal requirements across various sectors, as they are derived from federal laws and regulations intended to protect the integrity and confidentiality of sensitive data. Organizations implement these mechanisms to stay compliant with mandates that require a verifiable, time-sequenced history of system activities to support oversight and forensic reviews. In the 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. (U.S. Securities and Exchange Commission 2012)

Fowler (2004) 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.

2.8.2 Event Streams

2.8.3 Rebuilding state from an audit log and an event stream

2.9 Scalability of systems

3 Related Work

4 Proposed Method

This thesis aims to provide a fair, quantitative comparison of CRUD and CQRS / ES architectures. To achieve this, the architectures should be applied not only to the same domain, but to the exact same requirements. The implementations can then be tested against the same contract tests.

This chapter will first present the requirements for the actual application, then outline metrics and comparison methods.

4.1 Project requirements

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

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

4.1.1 Entities

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

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

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

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

4.1.2 Business rules

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

- Existence checks: any requests including references to entities will fail if the references entities do not exist.
- Requests leading to conflicts, for example creating a lecture with overlapping time slots, will fail
- When a student tries enrolling to a lecture which is already full, they will be put on a waitlist
- When a student disenrolls from a lecture, the next eligible student (higher semesters are preferred) will be enrolled
- Actions on a lecture can only be done during the appropriate lifecycle state (enrolling only when the lifecycle is "open for enrollment", grades can only be assigned when the lecture is "finished")

4.1.3 Contract Tests

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

These contract tests were implemented in a separate maven module called `test-suite`. The test classes use `JUnit 5` and `restassured` to send and assert HTTP requests. A pseudocode example might look like this:

4.2 Performance

4.3 Scalability or flexibility (TODO)

4.4 Traceability

4.5 Tech Stack

5 Implementation

5.1 CRUD implementation

5.2 ES/CQRS implementation

5.3 Infrastructure

6 Results

7 Discussion

7.1 Analysis of results

7.2 Conclusion & Further work

Finally, I'm done!

Glossary

ACID Atomicity, Consistency, Isolation, Durability. 8, 9

Anemic Domain Model The objects describing the domain only hold data, no logic. 6, 7

API API stands for *Application Programming Interface*. It describes the public interface of a module or service, often exposed over a network. 13

Atomicity Atomicity means that an action is either fully executed or not at all. Atomic operations make sure the application is not left in an invalid state (Bernstein & Newcomer 2009, p. 10). 8

BASE Basically Available, Soft State, Eventual Consistency. 9

Contract Test A contract test verifies that services implement a shared interface by testing their interactions against an explicitly defined contract. 11

CQRS Command Query Responsibility Segregation. 8, 10, 11, 13

CQS Command And Query Separation. 8

CRUD Create Read Update Delete. 8, 11, 13

DAO Data Access Object. 5

DDD Domain Driven Design. 6

DTO Data Transfer Object. 8

ES Event Sourcing. 11, 13

HATEOAS Hypermedia as the engine of application state. 4

HTML HyperText Markup Language. 5

HTTP HTTP stands for *Hypertext Transfer Protocol*. It is a protocol used in internet communication and was defined in RFC 2616 (*RFC 2616: HTTP/1.1* 2025). 4, 13

JSON JavaScript Object Notation. 5

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

Rich Domain Model Objects incorporate both data and the behavior or rules that govern that data. 6, 7

URI Uniform Resource Identifier. 4

WWW World Wide Web. 4

XML Extensible Markup Language. 5, 11

References

ATIS Committee (Mar. 2013). *ATIS Telecom Glossary - audit trail*. en. URL: `https://web.archive.org/web/20130313232104/http://www.atis.org/glossary/definition.aspx?id=5572` (visited on 01/07/2026).

Bernstein, Philip A. & Eric Newcomer (2009). *Principles of transaction processing*. eng. 2nd ed. The Morgan Kaufmann series in data management systems. Burlington, MA: Morgan Kaufmann Publishers. ISBN: 978-1-55860-623-4.

Braun, Susanne et al. (Oct. 2021). "Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study". In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. arXiv:2108.03758 [cs], pp. 1–11. DOI: `10.1145/3475716.3475771`. URL: `http://arxiv.org/abs/2108.03758` (visited on 12/27/2025).

Committee on National Security Systems (Apr. 2010). *National Information Assurance Glossary*. en. URL: `https://web.archive.org/web/20120227163121/http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf` (visited on 01/07/2026).

Evans, Eric (2004). *Domain-driven design: tackling complexity in the heart of software*. eng. Boston: Addison-Wesley. ISBN: 978-0-321-12521-7.

Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". en. PhD thesis. University of California. URL: `https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf` (visited on 01/08/2026).

Fowler, Martin (Nov. 2003). *Anemic Domain Model*. URL: `https://martinfowler.com/bliki/AnemicDomainModel.html` (visited on 12/27/2025).

— (Apr. 2004). *Audit Log*. URL: `https://martinfowler.com/eaaDev/AuditLog.html` (visited on 11/13/2025).

Fowler, Martin (Dec. 2005). *Event Sourcing*. URL: `https://martinfowler.com/eaaDev/EventSourcing.html` (visited on 11/13/2025).

— (Feb. 2017). *What do you mean by "Event-Driven"?* URL: `https://martinfowler.com/articles/201701-event-driven.html` (visited on 11/13/2025).

Gray, Jim et al. (1996). "The dangers of replication and a solution". en. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. DOI: `10.1145/233269.233330`. URL: `https://dl.acm.org/doi/epdf/10.1145/233269.233330` (visited on 12/28/2025).

Jacobs, Ian & Norman Walsh (Dec. 2004). *Architecture of the World Wide Web, Volume One*. URL: `https://www.w3.org/TR/webarch/` (visited on 12/27/2025).

Joint Task Force Interagency Working Group (Sept. 2020). *Security and Privacy Controls for Information Systems and Organizations*. en. Tech. rep. Edition: Revision 5. National Institute of Standards & Technology. DOI: `10.6028/NIST.SP.800-53r5`. URL: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf` (visited on 01/07/2026).

Lima, Stanley et al. (June 2021). "Improving observability in Event Sourcing systems". In: *Journal of Systems and Software* 181, p. 111015. DOI: `10.1016/j.jss.2021.111015`.

Malyi, Roman & Pavlo Serdyuk (Aug. 2024). "Developing a Performance Evaluation Benchmark for Event Sourcing Databases". en. In: *Vìsnik Nacìonal'nogo unìversitetu "L'vìvs'ka polìtehnìka". Serìâ Ìnformacìjnì sistemi ta merežì* 15, pp. 159–168. ISSN: 2524065X, 26630001. DOI: `10.23939/sisn2024.15.159`. URL: `https://science.lpnu.ua/sisn/all-volumes-and-issues/volume-15-2024/developing-performance-evaluation-benchmark-event` (visited on 11/03/2025).

Martin, James (1983). *Managing the data-base environment*. eng. Englewood Cliffs, N.J.: Prentice-Hall. ISBN: 978-0-13-550582-3.

Meyer, Bertrand (2006). *STANDARD EIFFEL*. en.

Michelson, Brenda (Feb. 2006). *Event-Driven Architecture Overview*. en. Tech. rep. 681. Boston, MA: Patricia Seybold Group, p. 681. DOI: `10.1571/bda2-2-06cc`. URL: `http://www.customers.com/articles/event-driven-architecture-overview` (visited on 01/02/2026).

RFC 2616: HTTP/1.1 (2025). URL: `https://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf` (visited on 12/27/2025).

Richards, Mark (2015). *Software Architecture Patterns*. en. O'Reilly. ISBN: 978-1-4919-2424-2. URL: `https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf` (visited on 01/08/2026).

Tilkov, Stefan (Dec. 2007). *A Brief Introduction to REST*. en. URL: https://www.infoq.com/articles/rest-introduction/ (visited on 01/08/2026).

U.S. Securities and Exchange Commission (Aug. 2012). *17 CFR § 242.613 - Consolidated Audit Trail*. en. URL: https://www.law.cornell.edu/cfr/text/17/242.613 (visited on 01/07/2026).

Vogels, Werner (Jan. 2009). "Eventually consistent". en. In: *Communications of the ACM* 52.1, pp. 40–44. DOI: 10.1145/1435417.1435432. URL: https://dl.acm.org/doi/epdf/10.1145/1435417.1435432 (visited on 12/28/2025).

Young, Greg (2010). *CQRS Documents by Greg Young*. en. URL: https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf (visited on 12/26/2025).

A Source Code

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