POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Migrating the User Domain from a Monolithic Recommendation Platform to a Microservice Architecture

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Simone Lucca**

Student ID: 244947
Advisor: Prof. Gianpaolo Saverio Cugola
Co-advisors:
Academic Year: 2025-2026

Abstract

This thesis presents the incremental migration of the user domain from the UX-Engine monolith — a large-scale personalization and content discovery platform developed by ContentWise/Moviri and serving over 100 million users globally — into a dedicated Spring Boot microservice.

The migration is guided by the Strangler Fig Pattern and Domain-Driven Design, with the user domain identified as a supporting subdomain whose tight coupling and dispersed logic made it one of the most challenging areas of the system to maintain.The new service reimplements all user-facing operations — creation, retrieval, update, rename, history deletion, and preference management — while preserving the existing database schema and the observable behavior of the legacy system. The shared database was retained as a deliberate architectural trade-off to enable coexistence between the monolith and the new service during the transition period.

Key implementation challenges addressed include the EAV-like schema inherited from the Moviri Application Library, a lost-update concurrency problem resolved through pessimistic locking, and the dual-write problem analyzed through the transactional outbox pattern and Change Data Capture with Debezium. The delete user history operation is analyzed as an inherently cross-domain workflow, with the Saga pattern identified as the correct long-term solution. The service was containerized and deployed on AWS EKS via a GitOps workflow with ArgoCD.

Validation was performed through unit tests, functional API tests against the existing ContentWise test suite, performance tests with Gatling, and a custom characterization testing framework based on database snapshot comparison. Results confirm behavioral equivalence with the legacy system, performance comparable to the monolith despite the additional network overhead of a Kubernetes deployment, and measurable improvements in code quality across all static analysis dimensions.

Keywords: microservices, monolith migration, Strangler Fig Pattern, Domain-Driven Design, Spring Boot, transactional outbox, Change Data Capture, characterization testing

Abstract in lingua italiana

Questa tesi presenta la migrazione incrementale del dominio utente dal monolite UX-Engine — una piattaforma di personalizzazione e content discovery sviluppata da ContentWise/Moviri, con oltre 100 milioni di utenti a livello globale — verso un microservizio dedicato basato su Spring Boot. La migrazione è guidata dallo Strangler Fig Pattern e dal Domain-Driven Design, con il dominio utente identificato come sottodominio di supporto il cui elevato accoppiamento e la logica dispersa lo rendevano una delle aree più difficili da manutenere nell'intero sistema.

Il nuovo servizio reimplementa tutte le operazioni rivolte agli utenti — creazione, recupero, aggiornamento, rinomina, cancellazione della cronologia e gestione delle preferenze — preservando lo schema del database esistente e il comportamento osservabile del sistema legacy. Il database condiviso è stato mantenuto come scelta architetturale deliberata, per consentire la coesistenza tra il monolite e il nuovo servizio durante il periodo di transizione. Tra le principali sfide implementative affrontate figurano lo schema EAV-like ereditato dalla Moviri Application Library, un problema di concorrenza di tipo lost update risolto tramite pessimistic locking, e il problema del dual write analizzato attraverso il transactional outbox pattern e il Change Data Capture con Debezium. L'operazione di cancellazione della cronologia utente è analizzata come un flusso intrinsecamente cross-domain, con il Saga pattern identificato come soluzione corretta nel lungo periodo.

Il servizio è stato containerizzato e distribuito su AWS EKS tramite un workflow GitOps con ArgoCD. La validazione è stata condotta attraverso unit test, test funzionali API eseguiti sulla suite di test esistente di ContentWise, test di performance con Gatling e un framework di characterization testing personalizzato basato sul confronto di snapshot del database. I risultati confermano l'equivalenza comportamentale con il sistema legacy, performance comparabili al monolite nonostante l'overhead di rete aggiuntivo introdotto dal deployment su Kubernetes, e miglioramenti misurabili della qualità del codice su tutte le dimensioni dell'analisi statica.

Parole chiave: microservizi, migrazione da monolite, Strangler Fig Pattern, Domain-Driven Design, Spring Boot, transactional outbox, Change Data Capture, characterization

testing

Contents

Introduction

Legacy systems built on monolithic architectures often reach a point where their initial simplicity becomes a liability. As business requirements evolve and system complexity grows, maintaining and extending these systems becomes increasingly challenging. This thesis presents a case study of migrating core functionality from a monolithic recommendation platform to a microservices architecture.

Background

Since its launch in 2007 as a spin-off from Moviri, a Milan-based technology consultancy, ContentWise has expanded considerably. Its core product, the UX-Engine (UXE), is a highly configurable personalisation and content discovery platform designed to adapt to the varying needs of each operator. Today it serves over 100 million users globally, processing approximately one billion personalisation calls per day, with offices in Milan, Los Angeles, and Singapore.

The platform serves large video operators that require a recommendation engine tailored to their specific content catalog and user base. Onboarding a new client involves an initialization process through which the system is configured to match the client's requirements, including catalog structure, recommendation strategy, and platform-specific parameters. Once deployed, the client interacts with the platform exclusively through its REST API. Content items — films, television series, and other streaming media — are ingested along with their descriptive metadata, enabling content-based filtering techniques that leverage attributes such as genre, cast, and editorial tags. Beyond content-based approaches, the platform supports multiple recommendation algorithms, allowing each deployment to adopt the strategy that best fits the characteristics of the client's catalogue and user interaction patterns.

The system is implemented as a large monolith running on JBoss [12] and relies on an event-driven architecture to manage internal interactions. Over time, the code base has grown substantially, and the lack of domain separation has made future development increasingly complex. The application is now large and complex, made of several modules

that emerged either from new requirements or from efforts to manage the problematic legacy monolith.

Although the team was well aware of the significant development debt accumulated in the user domain and across the monolith as a whole, migrating software so complex that it fit the description of a "big ball of mud" [6] was simply too costly to justify at the time. Their chosen strategy was straightforward and aligned with basic common sense, following the law of holes: "If you find yourself in a hole, stop digging" [1]. That is exactly what they did; from a certain point in time, whenever feasible, each new feature was implemented and released in a separate service. This tactic significantly slowed the progression of what was, in any case, unavoidable.

Objective and scopes

This project focuses on migrating the user business domain out of the UX-Engine monolith into a dedicated user management microservice. Rather than having the monolith invoke the new service, the new service takes over the user-facing API endpoints entirely: client applications that previously directed user-related requests to the monolith will instead call the new service directly. The underlying database remains shared for the duration of the migration, allowing the two systems to coexist without requiring a coordinated data layer transition. The user domain was selected as the extraction target due to the tight coupling and dispersed logic that had accumulated around it, making it one of the most challenging areas of the system to maintain and evolve. The objective of the work was to analyze the existing implementation, identify and consolidate the core user-related functionalities, and redesign them according to modern architectural principles — culminating in an incremental migration of the user domain and its deployment on a Kubernetes cluster.

In the end, the user-related endpoints were moved into a new Spring Boot–based microservice (Chapter 2), while the existing data layer was left unchanged. The new service was then deployed to a development environment and evaluated using functional tests (Section 3.2), performance tests (Section 3.3), and finally, characterization tests (Section 3.4).

Structure of the work

This thesis is organized as follows:

Chapter 1 introduces the conceptual foundations underlying the migration. It covers the monolithic and microservices architectural paradigms, the Strangler Fig Pattern as the

chosen incremental migration strategy, and Domain-Driven Design as the basis for service decomposition. The Spring Boot framework is then introduced, followed by a detailed analysis of the UX-Engine monolith, its internal architecture, and the technical issues that motivate the migration.

Chapter 2 describes the design and development of the new microservice. It examines the challenges of the existing data layer. It then covers the application layer structure, the Kafka-based event integration, and the deployment of the service on AWS EKS via a GitOps workflow with ArgoCD.

Chapter 3 describes the validation approach adopted for the new service. Unit tests check the core business logic in isolation, functional tests ensure behavioral parity with the legacy monolith using the existing ContentWise test suite, performance tests executed with Gatling measure throughput and response times under load, and characterization tests capture and preserve undocumented legacy behaviors uncovered during the migration. In addition, static code analysis with SonarQube has been used to assess improvements in the codebase structure.

Chapter 4 summarizes the outcomes of the project and reflects on the broader architectural implications. It also outlines the most relevant directions for future work, including Saga pattern implementation for cross-domain operations, the transactional outbox pattern, database decomposition, and the completion of the observability pipeline.

1 | Theory and background

This chapter introduces the theoretical and architectural concepts at the basis of the design change and code migration presented in this thesis. It first discusses monolithic architectures and the motivations for moving toward microservices. It then presents the Strangler Fig Pattern and Domain-Driven Design as the main conceptual foundations for incremental migration and service decomposition, then it briefly introduces Spring Boot, the framework used to implement the new microservice. The existing UXE monolith is then described in its architectural details to explain the reasons behind the decisions of transforming it into a microservices-based application.

1.1. Theoretical concepts

1.1.1. The monolith

The simplest mental model of a monolith software is one that is deployed on a single process, yet it is very common to have multiple instance of the same code onto multiple machines, for replication or scaling reasons [11].

As the application grows, this monolithic structure becomes a liability. Custom modules are created and added to the system to introduce new functionalities and they often lack clear documentation and depend on the structure of the existing code (strongly limiting reusability). When the codebase grows like this, it is often a nightmare to develop new features, and the monolith is often handled like a black box. A large number of requests will eventually awaken dormient concurrency issues, that force developers into the codebase, and cause them to waste a large amount of time trying to fix these issues.

Microservices offer a solution to this situation, providing a way to better organize code into strongly decoupled components, allowing to vary technology for each service, and supporting the adoption of new technologies in a safe way. This flexibility enables organizations to try new technologies without affecting the entire system. Additionally, microservices provide better fault isolation, as a memory leak in one service only affects that service, and other services continue to handle requests normally. This is particularly

important for large, complex systems where a single point of failure can have significant consequences.

Moreover, each service in a microservice architecture can be scaled independently of other services. This allows for more efficient use of resources, as services with different resource requirements can be deployed on hardware that's best suited to their needs. Furthermore, microservices eliminate long-term commitment to a technology stack, allowing for easier experimentation and adoption of new technologies. This enables organizations to respond quickly to changing business requirements and stay ahead of the competition.

Benefits of the monolith In spite of the limitations above, it is possible to see the monolith as a design choice rather than a deprecated solution. Indeed, a monolithic architecture is not inherently a poor choice; in many cases, it offers advantages that can make it the most suitable design option.

- Rapid development – there is no need for internal communication via messages across services. Simple function calls are sufficient.
- Easier to implement major changes – everything runs within a single codebase, without the external service dependencies, typical of microservices.
- Simpler to test – testing distributed systems is well known to be more complex and slower.
- Simpler to deploy - it is just one executable file.
- Straightforward to scale – while monoliths do not inherently scale as well as microservices, horizontal scaling can be achieved simply by running multiple instances behind a load balancer.

These characteristics explain why applications often born as monoliths. On the other hand, the limitations discussed above explain why the same applications often reach a point where they need to be redesigned and decomposed into smaller, better isolated components: the microservices.

1.1.2. Strangler Fig Pattern

The strangler fig pattern is an incremental migration strategy that allows functionality to be gradually moved from a legacy system to a new architecture without requiring a complete rewrite. Named after the strangler fig tree that grows around a host tree, this pattern involves building new functionality around the edges of the existing system while

slowly replacing internal components. As illustrated in Figure 1.1, the migration follows three steps: identify the functionality to extract, implement it as a new microservice alongside the monolith, and finally redirect incoming calls to the new service while leaving the original functionality in place as a fallback.

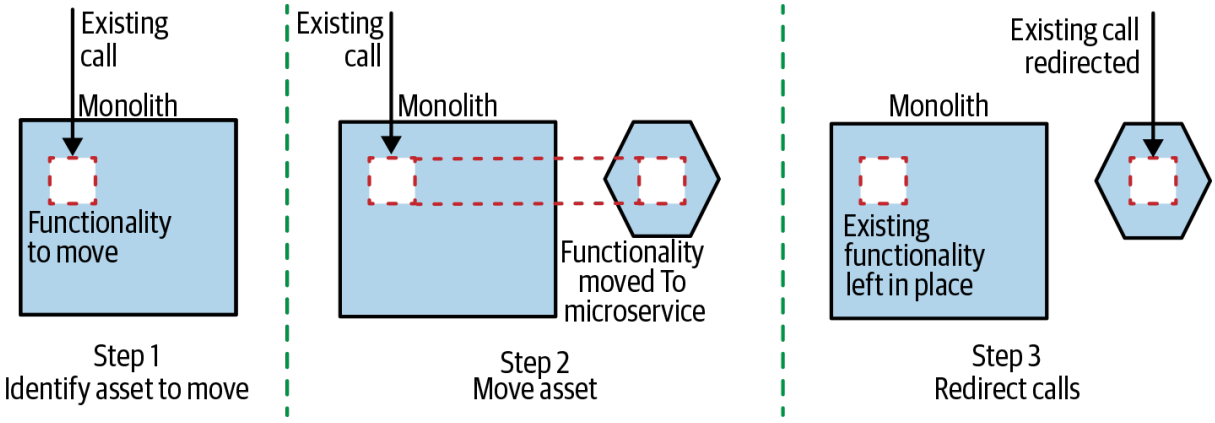


Figure 1.1: An overview of the strangler pattern [11]

The key advantage of this approach is that it allows the legacy system to continue to operate while the migration proceeds. New requests can be routed to the new services while existing functionality remains in the monolith. This reduces risk and allows for incremental validation of the new architecture. However, the pattern presents challenges when migrating functionality that has tight coupling to other parts of the monolith, requiring careful interface design and data consistency management.

1.1.3. Domain-Driven Design and Decomposition by Subdomain

Domain-Driven Design (DDD), introduced by Eric Evans in [4], is an approach to software development that centers the design of a system on the business domain it models. The central premise is that the structure and language of the code should match the business domain closely enough that a developer and a domain expert can communicate using the same terms without translation. This shared vocabulary is formalized as the *ubiquitous language*: a precise, unambiguous glossary agreed upon by engineers and stakeholders and used consistently in code, documentation, and conversation.

At the architectural level, DDD introduces the concept of a *bounded context*: a clearly delimited part of the system within which a particular domain model applies and the ubiquitous language is consistent. The same real-world concept, such as a "user", may have different meanings and attributes in different bounded contexts; a billing context cares about payment methods, while a recommendations context cares about preference

history. Keeping these models separate prevents the coupling that arises when a single overloaded entity tries to satisfy all consumers simultaneously.

Subdomains DDD distinguishes three types of subdomain based on their strategic importance to the business.

- **Core subdomains** represent the activities that differentiate the organization from its competitors. They are the primary source of business value and warrant the highest engineering investment.
- **Supporting subdomains** are necessary for the business to operate but do not provide competitive advantage on their own. They often implement well-understood processes that support the core.
- **Generic subdomains** address problems that are common across industries, such as authentication or billing, and are best solved by off-the-shelf solutions rather than custom development.

Decompose by subdomain Decompose by subdomain is a microservices decomposition strategy that derives service boundaries directly from the DDD subdomain analysis [13]. Each subdomain is implemented as one or more independent services, ensuring that service boundaries are aligned with natural business boundaries rather than arbitrary technical ones. This alignment has several practical consequences: services tend to be cohesive because all logic for a given business concern resides in one place, inter-service communication is minimized because operations that belong to the same subdomain do not cross a network boundary, and teams can own entire subdomains end-to-end, reducing coordination overhead.

In the context of this project, the UX-Engine monolith encompasses multiple subdomains — user management, recommendations, management of the events each user creates by interacting with the items, which are movies, search of movies in the catalog, and others — tightly coupled within a single deployable unit. User management is the focus of this thesis. It represents a supporting subdomain: it does not implement the recommendation logic that differentiates the product, but it provides the user identity and preference data that the core recommendation engine depends on. Extracting it as an independent microservice is a direct application of the decompose-by-subdomain pattern, establishing a bounded context with a well-defined API and isolating the user data model from the rest of the system.

1.1.4. Spring Boot

Spring Boot[17] is an open-source Java-based framework that simplifies the development of production-ready applications by providing opinionated defaults and automatic configuration. Built on top of the Spring Framework, Spring Boot eliminates much of the boilerplate configuration traditionally required for enterprise Java applications, allowing developers to focus on business logic rather than infrastructure setup.

The framework is particularly well-suited for microservices architectures due to several key characteristics. First, Spring Boot applications are self-contained, packaging the embedded web server directly within the executable JAR file. This eliminates external application server dependencies and aligns with the microservices principle of independent deployability. Second, the framework provides production-ready features out of the box, including health checks, metrics collection, and externalized configuration management through Spring Boot Actuator. These capabilities are essential for operating services in distributed environments where observability and operational control are critical.

Spring Boot's auto-configuration mechanism detects available libraries on the classpath and automatically configures beans with sensible defaults, significantly reducing the time required to bootstrap new services. For data access, Spring Data JPA provides a repository abstraction that eliminates repetitive data access code, while Spring Boot's built-in support for connection pooling and transaction management ensures efficient database operations. The framework's dependency injection container promotes loose coupling and testability, facilitating the development of maintainable microservices.

For this migration project, Spring Boot was selected as the implementation framework for the new user management microservice. The framework's maturity, extensive ecosystem, and alignment with microservices best practices, made it a natural choice for extracting functionality from the legacy monolith, while maintaining compatibility with existing infrastructure and operational practices.

1.2. The UX-Engine Monolith

1.2.1. Architecture Overview

The monolithic codebase exposes user operations through a REST API organized around resource-oriented endpoints. User operations follow standard CRUD patterns: `POST` for creation, `GET` for retrieval, `PUT` for updates, `DELETE` for removal, and `PATCH` for renaming operations. The API includes both top-level user resources and nested sub-resources for

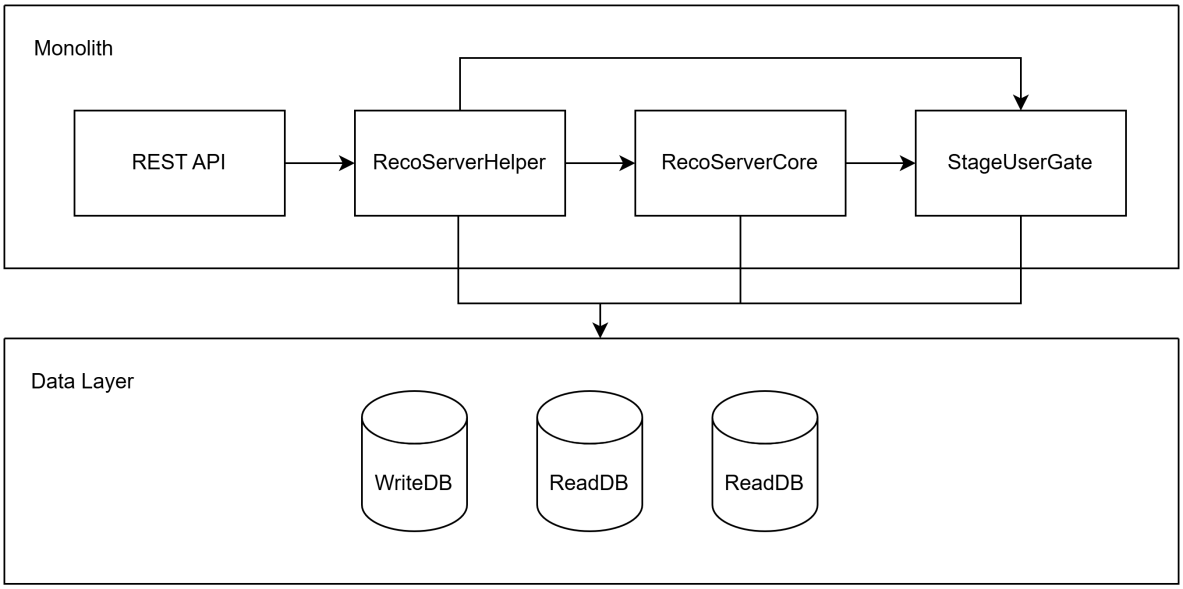


Figure 1.2: General internal structure of the monolith. The REST API delegates requests to a helper layer: it performs preliminary operations on the incoming data — including direct database access — before invoking the core business logic. The core components may then persist application state through `StageUserGate`, or bypass it entirely and access the database directly. The data layer exposes two read replicas, discussed in Section 1.3.2. The logic for routing queries to the appropriate write or read instance is embedded within the MAL library.

managing user preferences and queues. Each endpoint accepts tenant identifiers, user identifiers, and optional context parameters such as session tracking identifiers and device connection state.

The request processing was born as a strictly layered architecture, but in practice the boundaries among layers eroded significantly. In Figure 1.2, it is possible to see that every module communicates directly with the database, and distant layers communicate with each other. The core business logic class, `RecoServerCore`, grew to over 5600 lines of code and became responsible for not only user operations but also recommendations, search, item management, preference handling, caching coordination, analytics tracking, and license validation. This "God class" made the system difficult to understand, test, and modify. Each user operation required navigating through multiple abstraction layers only to find that business logic was scattered across the helper, core, and staging layers.

Some of the user's data persistence operations are implemented in a blob class called StageUserGate, which handles database storage by working directly with an `EntityDB-`

`Helper` abstraction. This helper is supplied by a custom data access framework named `MAL` (Moviri Application Library). StageUserGate is tightly coupled to the database schema, since it contains numerous direct references to the database helper.

The staging layer also manages the binding between `person` users and `terminal` users, where the latter represent physical devices such as televisions or smartphones. It additionally evaluates subdomain membership rules against user metadata to determine which logical partition a user belongs to.

By this point, the accumulation of poorly bounded responsibilities had turned the monolith into a big ball of mud [6]: a system where no discernible architecture remains.

Existing Services

The monolith is deployed alongside a set of pre-existing services. Among these, the most relevant is the `Events Service`. Inter-service communication is handled asynchronously through Apache Kafka.

As illustrated in Figure 1.3, the `Events Service` consumes Kafka messages produced by the monolith in response to user interactions — such as content clicks or viewing duration — and is responsible for updating the state of the system accordingly.

While other ancillary services exist within the ecosystem, they are less pertinent to the scope of this work. The focus here is on the services that are directly coupled with user-related functionality. Notably, every behavioural event recorded in the system is tied to a specific user, making the relationship between the `Events Service` and the user domain a key architectural dependency to account for during the migration.

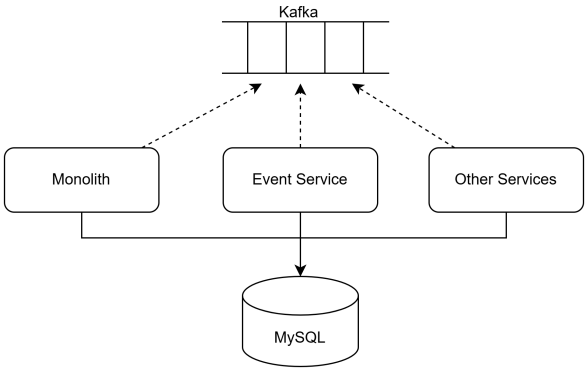


Figure 1.3: The existing services relevant to this work that will run alongside the new User Management Service.

1.2.2. Architectural Problems

A very significant architectural problem that affects the UX-Engine monolith is the violation of the single responsibility principle. The `RecoServerCore` class handles too many unrelated concerns, making it impossible to change user logic without risking impact to recommendation algorithms or search functionality. Business logic is distributed across the classes: RecoServerHelper, RecoServerCore, StageUserGate and their dependencies, with no clear separation of responsibilities, leading to duplicated validation checks and scattered business rules.

The tight coupling between layers prevents independent evolution. Changes to database schema require coordinating modifications in StageUserGate, the metadata reader configuration, RecoServerCore's metadata cleaning logic, and the REST layer's response builders. The lack of a proper repository pattern means database access code is mixed with business logic throughout the staging layer, making it difficult to test operations without a live database or to optimize query patterns independently.

Exception handling follows no consistent pattern, with eight different exception types thrown across various layers and no unified error handling strategy. Some exceptions represent technical failures while others represent business rule violations, but both are handled similarly at the API layer, resulting in generic error responses that provide little diagnostic value.

The interdependence of domains creates unnecessary coupling. User operations depend on the management of their associated terminal, subdomain evaluation rules, license validation, and analytics tracking. These cross-cutting concerns are woven directly into the user operation flow rather than being handled as separate aspects, making it difficult to modify one without affecting others.

The codebase was poorly documented, so the only way to grasp the requirements of each endpoint was through reverse engineering. The database access approach relied on familiarity with the existing EntityDBHelper class and its methods; instead of offering a proper data-layer abstraction, it was tightly coupled to the schema's structure, and the necessary modifications were similarly inadequately documented.

1.3. Data Layer

The UX-Engine stores its data in a MySQL relational database. This section describes the structure of the data layer, explains how the main application bottleneck arose and the scaling strategy adopted to handle growing traffic, and outlines the query access pattern

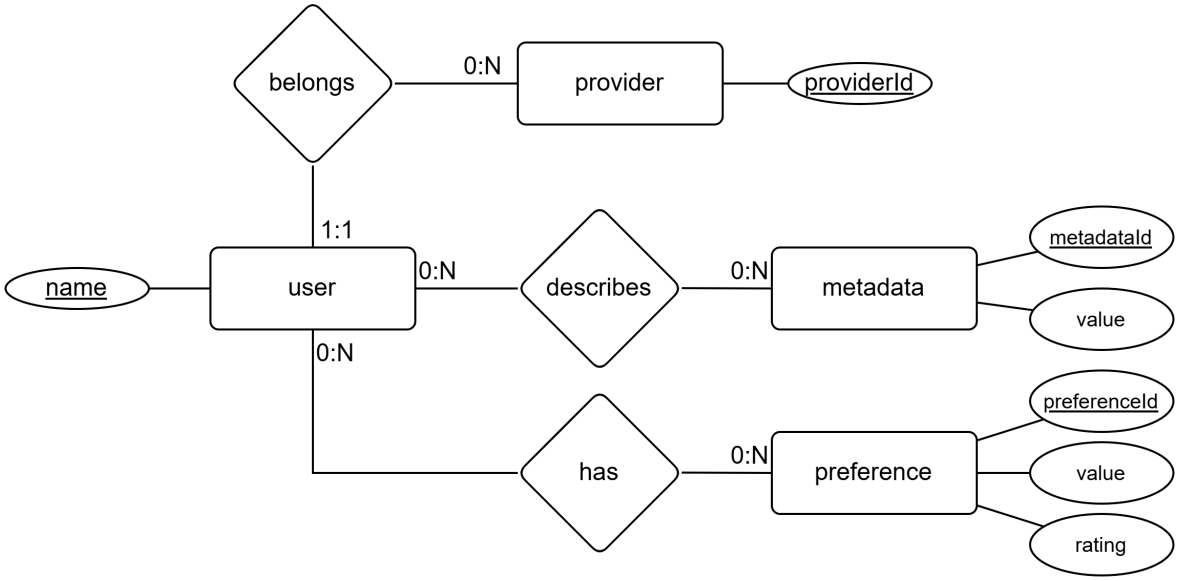


Figure 1.4: High-level ER diagram of the user data model, showing the relationships between users, provider, metadata entries, and preferences.

that most strongly shaped the migration. Together, these details provide the background necessary to understand the choices made throughout the migration.

1.3.1. Schema Structure

Figure 1.4 illustrates the high-level organization of the data model, emphasizing the core entities and their interrelationships. Each user is uniquely identified by a name and is associated with zero or more metadata fields, as well as a collection of preferences. Every metadata entry is composed of an id and a value, such as `metadataId=UserCountry` and `value=IT`. Similarly, each preference is defined by an id, a value, and a rating, for example `preferenceId=PrefActorsLastNameFirstArray`, `value=Cruise-Tom`, and `rating=4.0`.

In the implemented solution, preferences and metadata are stored into a single `rs_user` table, where each row contains the full set of metadata and preferences for a given user. User creation requires coordinating writes to four separate tables:

- `rs_user`: stores core user information, attributes, and preferences
- `rs_user_prov_lookup`: maintains the relationship between users and providers, storing the user's external identifier (a unique string used by providers) alongside the system's internal integer ID, as well as user-provider-specific metadata
- `rs_user_lookup`: provides an alternative user-provider relationship mapping with-

out metadata storage

- `rs_user_subdomain`: associates users with subdomains to track content interests (e.g., whether a user is interested in "sport.football")

This decision was primarly driven by the need for fast retrieval by user id, the drawbacks are different and are described in detail in Section 2.3, along with the reasons that motivated different decisions in the new user service.

1.3.2. Handling data load

As the application grew, the relational database became a bottleneck under increasing load. This section explains how horizontal scaling was introduced to address this problem, and provides the necessary background for Section 2.4.5, where this infrastructure plays a role in implementing the transactional outbox pattern.

Replication means keeping a copy of the same data on multiple machines that are connected via a network. Essentially, the reasons why replication is implemented are as follows:

- Keeping data geographically close to the user (to reduce latency)
- Allowing the system to continue working even if some parts have failed (to increase availability)
- Scaling the number of machines that can serve read queries (to increase the read throughput) [9]

In the UX-Engine scenario, the need for higher read throughput motivated the choice of a replication-based approach. The most widely used strategy for this is known as "leader-based replication," which works as follows:

1. One replica is chosen to act as the leader (also referred to as the master).
2. All other nodes function as followers. Whenever the leader writes new data to its own storage, it simultaneously records this change in a replication log and forwards it to each follower.
3. Every follower consumes the leader's log and updates its own local copy of the database applying the writes in exactly the same order as the leader.
4. For read operations, a client may contact the leader or any follower. However, all write operations must go through the leader, and followers appear read-only from the client's perspective.

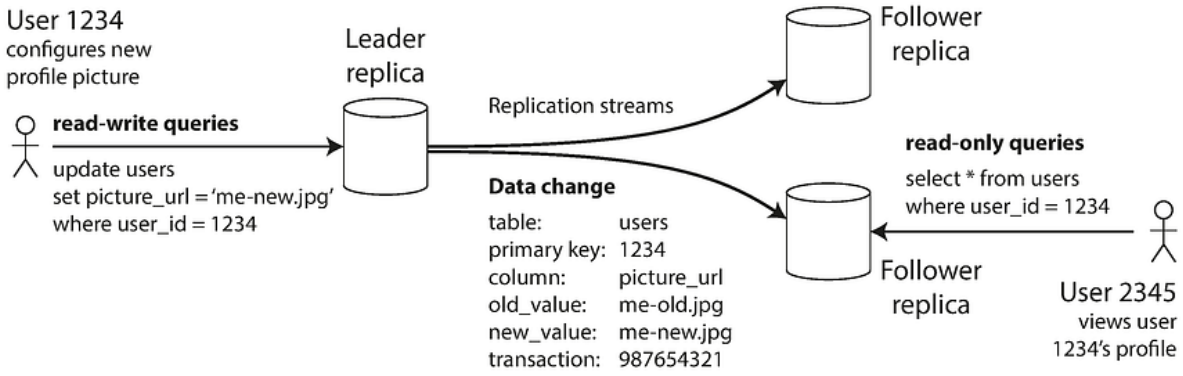


Figure 1.5: Leader-based (master-slave) replication

There are two other common methods for implementing replication:

- Multi-leader replication
- Leaderless replication

The leader-based method was clearly the most suitable choice for a system that was not designed to handle exponentially growing data loads.

A key property of any replicated system is whether replication is handled synchronously or asynchronously. With synchronous replication, all replicas must receive and apply updates before a transaction is finalized, which guaranties strong consistency but can add latency and lower availability. Asynchronous replication, on the other hand, lets the primary node confirm writes right away, while replicas are updated afterward in the background. This approach improves responsiveness and fault tolerance, although it can introduce short periods of data inconsistency. Considering these trade-offs, asynchronous replication was a fitting choice for the UX-Engine, where maximizing availability and performance is more important than enforcing strict real-time consistency. The replication mechanism follows MySQL's binary log (binlog) on the master node, where all write operations are recorded. Follower nodes continuously read this binlog and apply the same operations to their local copies, ensuring that data changes propagate from the master to all replicas in the order they were committed.

While leader-based replication successfully offloaded read traffic, it did not eliminate the database as a performance concern. Write operations remained confined to a single leader node, with millions of daily writes hitting a relational schema not optimized for that volume. The database therefore remained the primary bottleneck of the system – a constraint that directly influenced several decisions made during the migration.

1.3.3. Queries

When performing a migration, it is crucial to consider how the database is accessed. In this system, data access was mediated by a custom library, called Moviri Application Library (MAL), exposing a central class, EntityDbHelper, which provided direct interaction with the database schema. A peculiarity of this class is that it leverages a table named query.props into the database, which stores queries in the form of a couple: query name, fragment of SQL, as illustrated in Table 1.1. With this arrangement, SQL is externalized from the application code, allowing queries to be modified in production without redeploying the system. The main benefit of this choice is operational agility: bug fixes and performance optimizations can be rolled out rapidly, and shared query fragments can be reused across different components. However, this pattern erodes the usual engineering safeguards granted by code ownership. Changes become more challenging to version, review, test, and replicate across environments, and troubleshooting is harder because system behaviour depends on mutable state stored in the database. With the spread of DevOps practices, CI/CD pipelines, and immutable containerized deployments, this approach has become less desirable, because query behavior is controlled by mutable database state rather than versioned application code.

Overall, this design significantly increased the difficulty of understanding the interaction between business logic and the data layer.

name	`value`
ProviderExists.query	`p where exists (select null from rs_subdomain s where s.SUBDOMAINID = p.PROVIDERID and s.NAME = ?)`
Recom.GetUserTerminalId.query	`select l.provuserid from rs_user_lookup l, rs_user u where l.userid = u.termid and u.userid = ?  and u.usertypeid = ?`

Table 1.1: Some query definitions stored in the database

2 | Design and Implementation of the User Management Service

This chapter describes the design and implementation of the user management service. It begins by presenting the target architecture, explaining how the new service fits into the existing system and why a shared database was retained during the migration. It then examines the challenges posed by the existing data model and the custom data access library, which together shaped many of the implementation decisions. The core of the chapter covers the user-facing operations migrated from the monolith, discussing the logic, concurrency issues, and architectural trade-offs encountered for each. The chapter concludes with a description of the deployment environment used to validate the service.

2.1. New architecture

The new service was designed with Domain Driven Design principles (DDD) in mind (Section 1.1.3). User management is a supporting subdomain of the UX-Engine platform: it does not implement the recommendation logic that differentiates the product, but provides the identity and preference data the core subdomain depends on. The service would ideally be structured as a single bounded context with a well-defined API surface, owning its data model and exposing it exclusively through its interface. The internal layers map directly onto the DDD separation between application logic and domain logic, keeping business rules isolated from infrastructure concerns.

The user management service operates within an ecosystem of existing services, introduced in Section 1.2.1, and components that continue to rely on user data. In a microservices architecture, services communicate through well-defined interfaces: direct API calls for synchronous operations and event-driven messaging for asynchronous state propagation. The user management service exposes a REST API that other services can invoke to retrieve or modify user information, and publishes events to Kafka when user state changes, enabling cache invalidation and analytics processing across the platform, as shown in Figure 2.1.

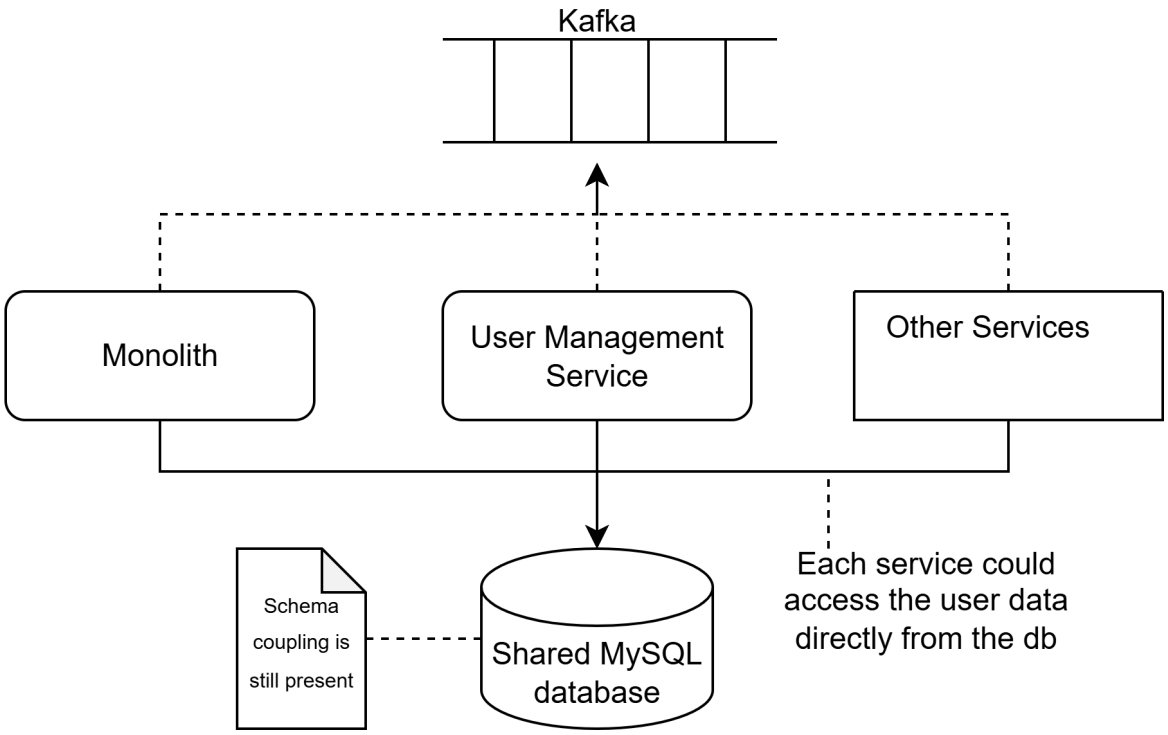


Figure 2.1: During the migration, the user management service uses the same database schema as the monolith. Kafka-based events are currently used to maintain cache consistency when user data changes, but they will become the primary communication mechanism once the data layer is fully decoupled.

2.2. Data Layer

Microservices best practices advocate for database isolation, where each service owns its data and other services access it only through well-defined APIs. As [10] puts it: "Don't share databases unless you really need to. And even then do everything you can to avoid it. In my opinion, sharing databases is one of the worst things you can do if you're trying to achieve independent deployability.". Shared databases create tight coupling between services and prevent independent schema evolution.

However, a well precise design choice made at the very beginning of my work by the company experts, was to let the user management service retain the shared database architecture during the transition period from the monolith to the new, fully service oriented architecture. Indeed, the database contains over 230 tables spanning multiple business domains, and the widespread direct database access from multiple system components of the monolith makes immediate schema separation hard if not infeasible. The

shared database enables the new service and legacy system to coexist while accessing the same data, supporting the incremental migration approach without requiring coordinated changes across all data consumers. Once service boundaries stabilize and direct database access has been eliminated in favor of API calls and event-driven communication, future development iterations will address schema separation to achieve full service autonomy.

Another design choice that we had to take before starting with the implementation of the user manager service, was to decide the right tooling for accessing the data layer. In particular, two alternatives were evaluated:

- Spring Data JPA [15]
- Continuing to use the existing libraries (MAL library)

JPA is a highly capable solution and, for a greenfield application with well-defined requirements, it would almost certainly have been the default choice. In this case, however, the context was different: the application was a legacy system, and extensive reverse engineering was required, so much that, after each API call, it was often unclear which tables were accessed. While the business logic itself was relatively simple, the data layer was tightly coupled and entangled with other business domains. In light of this, the final decision was to retain the custom library for database access.

2.2.1. Challenges of the Moviri Application Library

Migrating the persistence layer from the proprietary Moviri Application Library (MAL) to the Jakarta Persistence API (JPA) would introduce several strong structural and technical challenges. In the end, due to the architectural coupling between the database schema and the MAL framework, completing a full migration was cosnidered not economically or operationally viable within the existing project time and resource constraints.

On the other hand, the MAL framework suffers from documentation gaps, particularly around metadata mapping and the query.props pattern. The metadata system requires understanding a three-layer transformation process (REST → domain objects → generic metadata database columns), while query definitions are stored in database tables with undocumented naming conventions. New developers must reverse-engineer these patterns from existing code.

Migration resistance stems from deep framework embedding. The uxe-users service must manually bootstrap MAL at Spring Boot startup, creating temporary configuration files and setting system properties. Furthermore, the uxe-users service depends on uxe-caches, which itself directly instantiates EntityDBHelper for tenant ID retrieval. This transitive

dependency chain (uxe-users → uxe-caches → EntityDBHelper → MAL) means that implementing JPA into the user management service would require simultaneously migrating or replacing dependent libraries.

The non-standard approach creates knowledge concentration risks, as MAL's patterns differ significantly from JPA and other mainstream ORMs. The existing team has accumulated substantial institutional knowledge that is not easily transferred through documentation.

Given these constraints, moving to JPA would have involved: reverse-engineering the entire metadata mapping layer, creating custom JPA converters for the 20 distinct dynamic metadata columns, rewriting all database interaction and data handling logic—with the risk of losing some implicit requirements in the process—migrating or replacing the uxe-caches, and thoroughly validating everything against the monolith's existing behavior. This amounted to several months of work with a high likelihood of behavioral regressions.

2.3. User Data Model

2.3.1. User Attribute Structure

User data in the system is stored across several interconnected database tables, with the primary user record residing in the `rs_user` table. Each user record contains core identification fields such as the internal user ID, user type, terminal ID, and update timestamp, along with a flexible metadata structure that accommodates various user characteristics and preferences.

The metadata is stored as dynamically mapped columns in the `rs_user` table. The system distinguishes between two categories of metadata attributes based on their naming convention: array-type attributes and simple attributes.

Array-type attributes are identified by names ending with the suffix "Array". These fields are designed to store multiple values and typically represent user preferences, subscriptions, or accumulated characteristics. Examples include `PrefGenresArray`, `PrefActorsLastNameFirstArray`, `SubscribedPackagesArray`, and `UserLanguagesArray`. Each array-type attribute contains a set of string values stored in a single column, with individual entries separated by the "#" character.

Simple attributes, which lack the "Array" suffix, store single scalar values. These fields represent profile information that has a single definitive state at any given time. Examples include `UserCountry`, `Technology`, `BirthYear`, `UserPrefMdLanguage`, and `TerminalIP`. Each simple attribute holds a single string value representing the current state of that characteristic.

rs_user								
UserId	MD1	MD2	MD3	...	MD18	MD19	MD20	UserType
101	28.87.220.69	IT	EN		Adventure=5.0# Action=4.0# Comedy=3.0#	Cruise-Tom=4.0# Depp-Johnny=5.0# Johansson-Scarlett=5.0	IT#EN#	1
102	28.87.220.69	1976	EN		Romance=4.0# Detective=5.0# Musical=1.0#	Roberts-Julia=5.0# Hepburn-Audrey=5.0# Bullock-Sandra=4.5#	IT#EN#	2

rs_user_type								
UserType	MDDESC1	MDDESC2	MDDESC3	...	MDDESC18	MDDESC19	MDDESC20	UserTypeId
terminal	TerminalIP	UserCountry	UserPrefMd-Language		PrefGenresArray	PrefActorsLast-NameFirstArray	UserLanguages-Array	1
person	TerminalIP	BirthYear	UserPrefMd-Language		PrefGenresArray	PrefActorsLast-NameFirstArray	UserLanguages-Array	2

Figure 2.2: The `rs_user` table stores user metadata across 20 generic columns `MD1`–`MD20`. The semantic meaning of each column is defined per user type in the `rs_user_type` table via the descriptor columns `MDDESC1`–`MDDESC20`. For example, `MD2` holds `UserCountry` for terminal users (user 101, value `IT`) and `BirthYear` for person users (user 102, value `1976`). Array-type attributes such as `PrefGenresArray` and `PrefActorsLastNameFirstArray` store multiple values in a single column, with entries separated by the `#` character. This indirection is the defining characteristic of the EAV-like schema.

2.3.2. Database Schema Limitations

The metadata storage design presents significant performance and maintainability challenges. The table `rs_user` stores the actual metadata values in 20 generic columns named `MD1` through `MD20`, each with varying maximum lengths ranging from 10 to 4000 characters. Figure 2.2 shows the structure of the data, highlighting the need for a distinct mapping table to understand the meaning of each metadata.

The mapping between these generic columns and actual attribute names is maintained in the `rs_user_type` table, which contains corresponding descriptor columns `MDDESC1` through `MDDESC20`. Each user type defines its own mapping, specifying which metadata field name corresponds to which `MD` column. For example, for a `TERMINAL` user type, `MDDESC18` might specify "PrefGenresArray", meaning that genre preferences for terminal users are stored in the `MD18` column of `rs_user`.

This design represents a variant of the Entity-Attribute-Value (EAV) [18] anti-pattern with particularly severe drawbacks. To update a single user preference, the system must

perform the following steps:

1. Fetch the entire user row from the database, retrieving all 20 metadata columns regardless of which one will be modified.
2. Join with `rs_user_type` to retrieve the complete metadata descriptor mapping for that user type.
3. Iterate through all descriptor fields to identify which `MD` column contains the target attribute.
4. Parse the existing value from that column.
5. Modify the value according to merge or replace semantics.
6. Write the entire user row back to the database.

This approach has several negative consequences. Query performance suffers because the database cannot create indexes on specific logical attributes, only on the generic `MD` columns whose contents vary by user type. Filtering or searching users by a specific preference requires scanning all rows and interpreting the type-specific mappings, making such queries prohibitively expensive. The schema is not flexible, imposing a hard limit of 20 metadata fields per user, regardless of actual requirements. Adding new metadata fields requires either consuming an unused `MD` column slot or restructuring the schema.

The indirection through the type mapping table adds complexity to all queries and updates, as the application must maintain the mapping in memory and apply it consistently. Furthermore, this design makes the database schema nearly incomprehensible without external documentation. Examining the raw table structure provides no insight into what data is actually stored or how it is organized.

From a development perspective, this schema requires significant custom code to translate between meaningful attribute names and generic column positions. The reflection-based `MetadataReader` class in the code exists solely to manage this mapping and dynamically set values on the correct columns. The design also makes schema evolution difficult, as changing metadata definitions requires coordinating updates across the type mapping table, the application code, and any data migration scripts.

2.4. User Operations

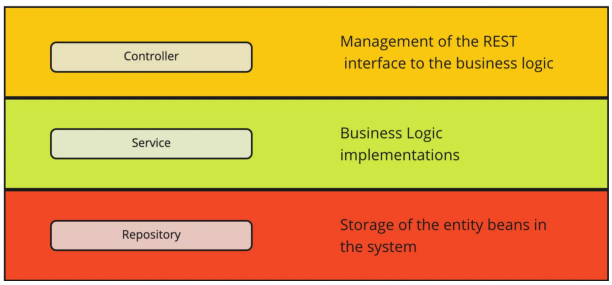


Figure 2.3: Layers in the controller-service-repository design pattern

2.4.1. Application Layer

Maintainability in software systems is closely tied to how clearly responsibilities are separated across the codebase. The user management service adopts the Controller-Service-Repository pattern to enforce distinct boundaries between the presentation, business logic, and data access concerns. The layers are shown in Figure 2.3.

The **Controller** layer sits at the top of this hierarchy and acts as the service's entry point, exposing endpoints consumed by external clients — including the user interface and peer services. It carries no business logic of its own; its sole responsibility is to receive requests and delegate them appropriately.

The **Service** layer encapsulates the business logic of the application. Whenever an operation requires reading or persisting data, the service delegates to the appropriate Repository. This ensures that domain rules remain centralized and decoupled from both the transport mechanism above and the data access mechanism below.

The **Repository** layer is the sole component that interacts directly with the MAL library to communicate with the underlying database. Confining this dependency to a single layer has a concrete architectural benefit: the challenges outlined in Section 2.2.1 are contained in one place, and any future change in the underlying data access technology would require modifications only at this level, leaving the rest of the service unaffected.

This layered structure represents a meaningful departure from the original monolithic implementation, where no such separation existed and responsibilities were diffused across the codebase. Beyond architectural clarity, the pattern also improves testability: each layer can be exercised in isolation by mocking its neighbors, without the need to spin up the full stack.

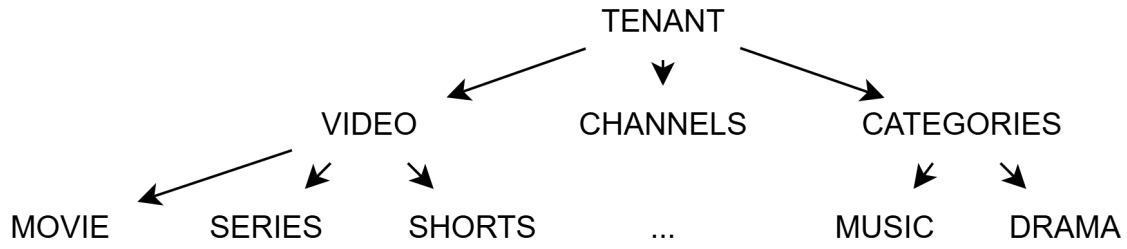


Figure 2.4: Example of a subdomain hierarchy

2.4.2. Create User

The `create user` operation exemplifies the earlier architectural problems. Although the developers intended a layered architecture, the god classes `RecoServerCore` and `RecoServerHelper` accumulated too many responsibilities, becoming hard to understand and evolve. As a result, even deprecated features were rarely modified, and functions ended up with long parameter lists where only a few arguments varied while the rest stayed constant.

When a user is created, the service allocates a set of subdomains to that user based on predefined rules specified in external configuration resources. These subdomains denote specific areas of interest within the platform, for example `TENANT.VIDEO` or `TENANT.CHANNELS`, and are persisted as entries in the `rs_subdomain` table.

These subdomains are organized in a hierarchical structure, where every entry except the root holds a reference to its parent node. As illustrated in Figure 2.4, the subdomain `VIDEO` is a child of `TENANT`, and `MOVIE` in turn descends from `VIDEO`. When a user matches a given rule, they are assigned to the corresponding leaf node in this hierarchy — for example, `TENANT.VIDEO.MOVIE` — capturing a fine-grained expression of their content preferences.

In the monolith, subdomain handling was managed by the StageUserGate class, which, as described in Section 1.2.2, violated the Single Responsibility Principle through its many conflated concerns. The new implementation addresses this by creating a dedicated SubdomainsEvaluator class focused solely on subdomain management.

In the monolith's events service, when an event is generated for a user that does not exist in the database, it triggers a function to create that user. This introduces a clear coupling: once the user management service is migrated, the monolith will also need to be updated to handle this edge case. For now, the database remains shared, but the overall complexity

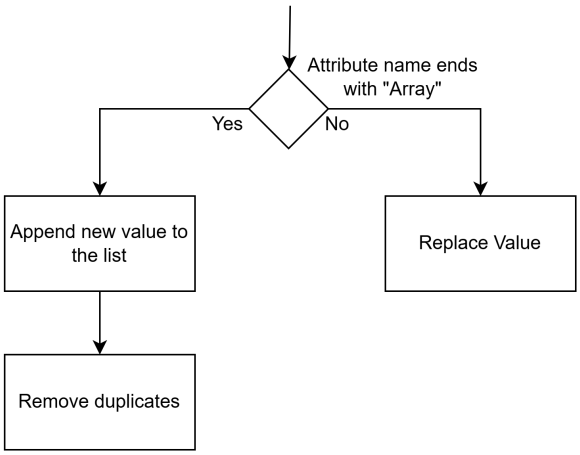


Figure 2.5: Logic used to update a metadata field

increases significantly if this user-creation logic is split between the monolith and the new user management service. Future updates must correctly address this scenario, and very often they will need the transactional outbox design pattern (Section 2.4.5).

2.4.3. Get User

The get user operation retrieves the user profile information and metadata from the database. The operation fetches the user record from the `rs_user` table along with the associated provider lookup information, retrieves the metadata descriptor mapping for the user's type, and translates the generic `MD` column values into their corresponding logical attribute names before returning the response.

2.4.4. Update User

The update operation in the monolith was implemented within the same method as create, resulting in messy code with unclear variable names. After careful analysis, the underlying logic became apparent:

Fetch user → change user by merging the attributes → save user

Update Merge Logic The update operation implements a merge logic that treats the two attribute categories differently (Figure 2.5). For array-type attributes, the system appends the newly provided value to the existing list. The merge operation maintains insertion order, with existing values appearing first followed by newly added values. Duplicate values are automatically eliminated.

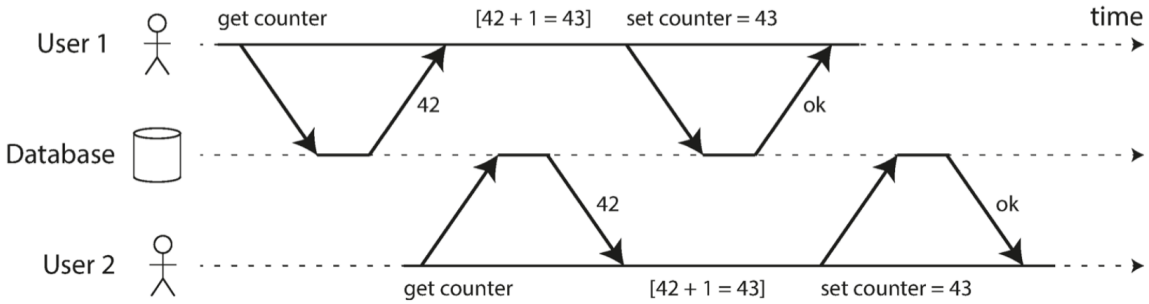


Figure 2.6: A race condition between two clients incrementing a counter

Simple attributes, follow replacement semantics. When an update provides a new value for a simple attribute, it completely overwrites the previous value. This behavior is appropriate for profile fields where only the current state matters. If a user's `UserCountry` is "US" and an update specifies "UK", the resulting value is simply "UK" with no trace of the previous value.

This merge-versus-replace decision is determined automatically based on the attribute name suffix. The convention-based approach provides flexibility for the system to handle different types of user characteristics appropriately: array metadata accumulate over time through merging, while profile attributes reflect current state through replacement.

Concurrency Problem: Lost Update The implemented code presented a concurrency problem due to lost update. Two threads may operate on the same user, but one of the two updates is completely discarded. As shown in Figure 2.6, User 1 sends the request, but it is effectively as if the request was never sent. Here is a scenario that may happen:

1. Thread A reads user (status = ACTIVE, metadata = favoriteGenre: "Action")
2. Thread B reads same user (status = ACTIVE, metadata = favoriteGenre: "Action")
3. Thread B updates user (metadata = favoriteGenre: "Comedy") and commits
4. Thread A updates user (metadata = favoriteGenre: "Drama") and commits
5. Result: Thread B's metadata change is LOST

In the case of recommendations, every datapoint is valuable, this is why it is necessary to solve this issue. Here are presented possible solutions to the lost update problem:

Atomic update operations A possible solution would be to perform the modification through a single atomic database update, so that the new values are computed and written directly by the database without transferring the record to the application layer. However, this approach was not adopted for two main reasons:

1. the update logic is heterogeneous: some columns require simple replacement, whereas others require updating the string representation of an array by appending new values to the existing content using the character '#' as a separator. Expressing both behaviors correctly within a single query would be complex and error-prone;
2. integrating such a query into the current persistence layer would also be difficult, since queries are assembled in a non-intuitive way and the corresponding data seeds used to initialize new database instances would need to be updated accordingly.

Although atomic database updates are generally an effective solution to prevent lost updates, in this case the required query complexity and the maintenance burden make this option impractical.

Optimistic locking Optimistic locking assumes that concurrent modifications of the same record are rare. Each row stores a timestamp that is read together with the entity and checked on update. The update succeeds only if the stored version still matches the one previously read, otherwise the system detects a conflict and the operation must be retried. This option was discarded because in the user table, the insertion time format have a granularity of seconds, that means that it is still possible for a transaction to update the previous timestamp with an exact same one if it is fast enough. There would be the possibility to change the schema and add a new column containing the exact insertion timestamp, but we already know that the db is struggling a lot with data load and we would add another burden to the mess.

Pessimistic locking Pessimistic locking is best suited for scenarios where conflicts are frequent or costly. When a record is read, it is immediately locked, preventing other transactions from modifying it until the current transaction commits. Since only writes are blocked—and write contention on the same user record is rare in this context—the risk of thread queuing remains negligible.

MySQL supports this through the `SELECT FOR UPDATE` statement, which acquires a row-level lock scoped to the current transaction. Given the low likelihood of concurrent updates to the same user, this approach was selected as the solution for preventing lost updates in the user management service.

2.4.5. Delete User

The delete user operation removes a user and all associated data from the system. This operation presents particular challenges due to the need to maintain consistency across multiple tables and external systems. When a user is deleted, the system must remove records from the core user table, the provider lookup table, the user-subdomain associations, trigger cleanup of related data such as user preferences and historical events and notify other processes to clean their user caches through a kafka message.

Dual write problem The dual-write problem arises when a single logical operation requires updating two or more separate systems that cannot participate in a single atomic transaction. In distributed architectures, this commonly occurs when persisting data to a local database while simultaneously publishing an event to a message broker or updating an external service. The fundamental challenge is that these systems operate independently with their own transaction boundaries, making it impossible to commit or roll back changes across both systems atomically.

If the application writes to the first system successfully but fails before completing the write to the second system due to network failure, process crash, or external service unavailability, the systems become inconsistent. The first write has been committed and cannot be automatically rolled back, while the second write never occurred. Even attempting to order the writes carefully cannot eliminate these failure modes. There is always a window between the two operations where failure leaves the system in an inconsistent state. Traditional distributed transaction protocols like two-phase commit exist to address this problem but impose significant performance costs and operational complexity that make them unsuitable for many modern distributed systems.

Transactional outbox pattern The transactional outbox pattern solves the dual-write problem by reducing it to a single write within a single transaction boundary. Instead of writing to the database and then attempting to publish an event to an external system, the application writes both the business data and a representation of the event into the same database within a single transaction, as shown in figure 2.7. The event is stored in a dedicated outbox table that acts as a persistent queue. A separate process, typically running asynchronously, continuously polls or listens to the outbox table, reads unpublished events, publishes them to the external system, and marks them as published. If the application crashes after committing the transaction, the event remains in the outbox table and will be published when the polling process next runs. This approach guarantees at-least-once delivery: the event will eventually be published, though

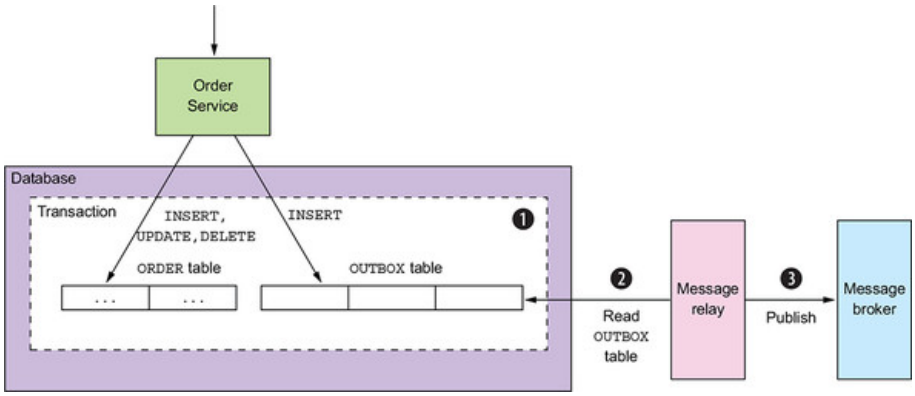


Figure 2.7: The transactional outbox pattern

it may be delivered multiple times if the publishing process crashes between publishing and marking the event as published. The pattern trades the impossibility of achieving atomic distributed transactions for the more manageable problem of ensuring idempotent message handling in downstream consumers.

Transactional outbox through polling The most straightforward implementation of the outbox pattern relies on periodic polling. A dedicated MessageRelay process executes a scheduled task that queries the outbox table at regular intervals to retrieve unpublished events:

```
SELECT * FROM OUTBOX ORDER BY created_at ASC
```

For each retrieved event, the MessageRelay publishes the corresponding message to Kafka. Once publication succeeds, the process removes the published events from the outbox table within a transaction:

```
BEGIN
  DELETE FROM OUTBOX WHERE ID IN (...)
COMMIT
```

While this approach is simple to implement and reason about, it presents significant scalability concerns. The periodic queries introduce constant load on the database regardless of whether new events exist, with query frequency determining both the additional database overhead and the latency between event creation and publication. As event volume grows, the outbox table accumulates more rows between polling intervals, making each query more expensive. Furthermore, coordinating multiple MessageRelay instances to avoid duplicate processing requires additional locking mechanisms that further increase database contention. The database, already identified as a performance bottleneck in the existing architecture, cannot sustain this additional periodic query load without impacting the

performance of primary business operations.

Change Data Capture Change Data Capture (CDC) provides an efficient mechanism for monitoring the outbox table and publishing events in response to new entries. Rather than polling the outbox table at regular intervals, CDC monitors the database's transaction log directly. Modern databases maintain a write-ahead log or binary log that records all committed transactions for replication and recovery purposes. CDC tools read this log, detect when new records are inserted into the outbox table, and immediately trigger event publication to external systems. This approach offers lower latency compared to polling-based solutions, eliminates the overhead of repeatedly querying the outbox table, and guarantees that events are published in the same order they were committed to the database. The CDC process runs independently of the application, ensuring that event publication continues even if application instances restart or fail.

The flexibility and throughput efficiency of this solution made it look like the best solution. The implementation included a dedicated outbox table in the same database schema, where deletion events are written atomically within the same transaction as the user deletion. Debezium [3], an open-source distributed platform for change data capture, monitors the MySQL binary log, detects new outbox entries, and publishes events to a Kafka topic for downstream consumers to process. Debezium provides connectors for various databases and handles the complexity of reading transaction logs, tracking processed events, and managing failures, making CDC implementation straightforward and reliable.

Upon completing the implementation, a critical configuration conflict emerged. Debezium requires specific binary log settings to properly monitor the MySQL transaction log:

- binlog_format = ROW
- binlog_row_image = FULL
- binlog_row_metadata = FULL

However, as described in Section 1.3.2, the existing MySQL infrastructure uses the binary log for replication across three replica nodes. These replicas operate efficiently with minimal binary log verbosity:

- binlog_format = ROW
- binlog_row_image = MINIMAL
- binlog_row_metadata = MINIMAL

The database was already the primary performance bottleneck of the application. Changing the binary log configuration to satisfy Debezium's requirements would have substantially increased the write load on the database by forcing MySQL to record complete row images rather than only changed columns. Given the existing performance constraints, this additional overhead was not feasible.

Alternative solution: CDC on read replica An alternative approach was considered to circumvent the master database performance constraints. Rather than configuring Debezium to monitor the master's binary log, the CDC process could instead monitor the binary log of a dedicated read replica. This replica's binlog configuration could be independently modified to meet Debezium's requirements (`binlog_row_image = FULL`, `binlog_row_metadata = FULL`) without impacting the master's write performance.

This approach introduces a delay in event publication equal to the replication lag from master to replica plus the CDC processing time. However, for user deletion operations, strict timing guarantees are not required. The database remains the source of truth, and any queries executed during the delay window will still reflect the correct deletion state.

The trade-off of this solution lies in increased operational complexity. The system becomes dependent on a specific read replica for correct event publication, requiring additional failure handling logic. If the designated replica becomes unavailable, event publication halts until the replica recovers or Debezium is reconfigured to monitor an alternative replica.

Listen to Yourself pattern The Listen to Yourself pattern offers a more radical solution that completely bypasses the database during request processing. Instead of writing to the database first, the service publishes events to Kafka immediately, then consumes its own events through a dedicated consumer group to asynchronously update the database. This inverts the traditional flow: Kafka becomes the temporary source of truth, and database writes happen as a side effect of event consumption.

This approach directly addresses the database overload problem by removing database transactions from the critical request path. The API responds as soon as Kafka commits the event, typically in milliseconds, while database updates happen asynchronously at a sustainable rate. The pattern also eliminates the dual-write problem, since only Kafka is written during request handling.

However, the pattern introduces a relevant consistency problem: the database lags behind the event stream by the consumer processing time. For user deletion, this means

queries may still return the deleted user until the consumer processes the deletion event, breaking read-your-own-write guarantees [2, 16]. A user might delete their account yet find the system still considers it active for a brief window afterward, potentially causing downstream consistency issues. Beyond this, the pattern demands careful event ordering via partition keys, idempotent consumer logic to tolerate duplicate deliveries, and robust error handling for failed event processing.

A particularly dangerous race condition arises when deletion events are used for cache invalidation across multiple services. Consider the following sequence:

1. User management Service publishes the deletion event to Kafka
2. Event Service consumes the message and clears its user cache
3. Event Service receives a subsequent request requiring user data, queries the database, and finds the user (deletion has not yet occurred)
4. User management Service consumes its own deletion event and removes the user from the database

The Event Service now caches data for a non-existent user. This stale cache entry can persist indefinitely, causing the Event Service to generate recommendations, analytics, or notifications for deleted users. Cache TTLs only limit how long stale data persists, not whether the race condition occurs. The cache entry created in step 3 remains valid for its full TTL period, even after the user is deleted in step 4. Mitigating this requires either including complete user state in deletion events (allowing services to distinguish between "user not found" and "user deleted"), or implementing tombstone markers in caches that explicitly track deleted entities rather than simply clearing cache entries.

Additionally, the system's availability model inverts: Kafka unavailability blocks all writes, even if the database is healthy. Consumer lag monitoring becomes critical, as excessive lag widens the inconsistency window and increases the probability of race conditions like the cache invalidation scenario described above. Alerting thresholds must account for the acceptable staleness window, which varies by operation type.

For UX-Engine's user deletion use case, the performance benefits related to the db load are significant, but the eventual consistency trade-offs, race condition risks, and operational complexity make this solution unfeasible for the current scenario.

Implemented solution The transactional outbox pattern was implemented to address the dual-write problem, allowing atomic writes of both database changes and event records within a single transaction. The event relay will use Debezium monitoring a read replica's

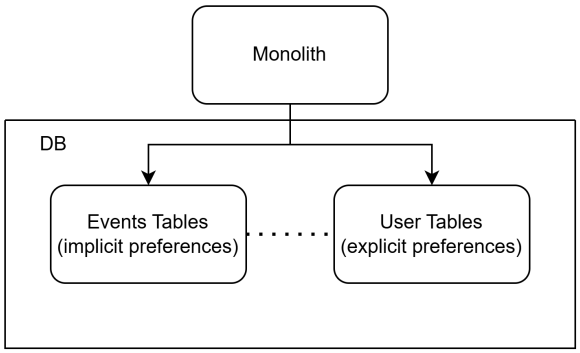


Figure 2.8: Implicit preferences (events) and explicit preferences (ratings) belong to different business domains — the event service and the user management service respectively — yet share the same internal user identifiers, creating an implicit dependency between the two domains.

binary log, as discussed in the CDC on read replica (Section 2.4.5), avoiding performance impact on the master database. However, full deployment is pending organizational review of the operational trade-offs, including replication lag implications and replica dependency management.

2.4.6. Rename User

The rename user operation changes a user's identifier while preserving all associated metadata and historical data. This operation is implemented as an atomic update to the provider user ID fields across all affected tables. The operation validates that the new user ID does not already exist before proceeding with the rename, and maintains referential integrity across the user, lookup, and subdomain association tables. A message is then sent to kafka to notify other services of the change. The current version has no problems with data consistency at the db level, since the database is shared, but the asynchronous message has been added for cache refresh and future developments.

2.4.7. Delete User History

The system tracks user interactions with the catalogue and application through events generated during navigation and consumption, referred to as *implicit preferences*. It also maintains *explicit preferences*, which represent ratings a user has assigned to specific content or content metadata. When a user requests history deletion, the UX-Engine recommendation platform must remove both preference types from the database.

As Figure 2.8 illustrates, implicit and explicit preferences belong to distinct bounded

contexts: implicit preferences fall under the event domain, while explicit preferences fall under the user management domain. This distinction has architectural consequences. The monolith's implementation of history deletion operates directly on the event tables, even though event-related functionality was partially extracted into a dedicated microservice in an earlier migration phase. As a result, the delete history operation spans a boundary that has already been drawn, introducing coupling between the monolith and the event service and obscuring data ownership.

The legacy service exposes a dedicated endpoint, `{tenant}/users/{external_user_identifier}/history`, for resetting the behavioural history of a user. Removing all events associated with a single user is not feasible synchronously, as one user may have generated thousands of event rows. The monolith therefore implements a two-phase logical deletion as shown in 2.9.

In the first phase, the user's external identifier in `rs_user_prov_lookup` is overwritten with a generated name of the form `_CLEAN_<random-string>`. Only this table is updated; the remaining tables that reference the user are left unchanged, introducing a transient inconsistency. This is sufficient to hide the user from the system, because MAL resolves user existence exclusively through `rs_user_prov_lookup`. Immediately after the rename, the original external identifier is re-registered as a brand-new user, which receives a fresh internal ID with no event rows attached. The link between internal IDs in the table rs_user _lookup and event rows is not enforced by a foreign key, so the orphaned rows from the old ID remain in the database.

The first phase exposes two correctness hazards worth documenting.

The primary hazard is a failure window between the two writes. The normal execution sequence is:

1. A transaction renames the user's external identifier in `rs_user_prov_lookup` to the generated `_CLEAN_` name.
2. The monolith's application layer creates a new user record under the original identifier, propagating the insertion across all dependent tables and assigning a fresh internal ID.
3. The link between the old internal ID and its event rows is now severed: the original identifier resolves to the new ID, while the orphaned event rows still reference the old one.

The two writes are not atomic. If the process crashes or is rescheduled after step 1 but before step 2 completes, the user ceases to exist in `rs_user_prov_lookup` while their

records remain intact in all other tables. The system is left in an inconsistent state with no automated recovery path: the user cannot log in, yet their data has not been deleted.

A secondary hazard arises from the gap between the rename and the re-registration. If a different user registers with the original identifier during this window, the monolith will associate the new registration with the incoming request and the behaviour becomes unpredictable — the new account may inherit state or conflict with the in-progress deletion.

In the second phase, a manually activated task scans for all entries whose external identifier starts with `_CLEAN_`, retrieves their internal IDs, and deletes the associated event rows, restoring consistency.

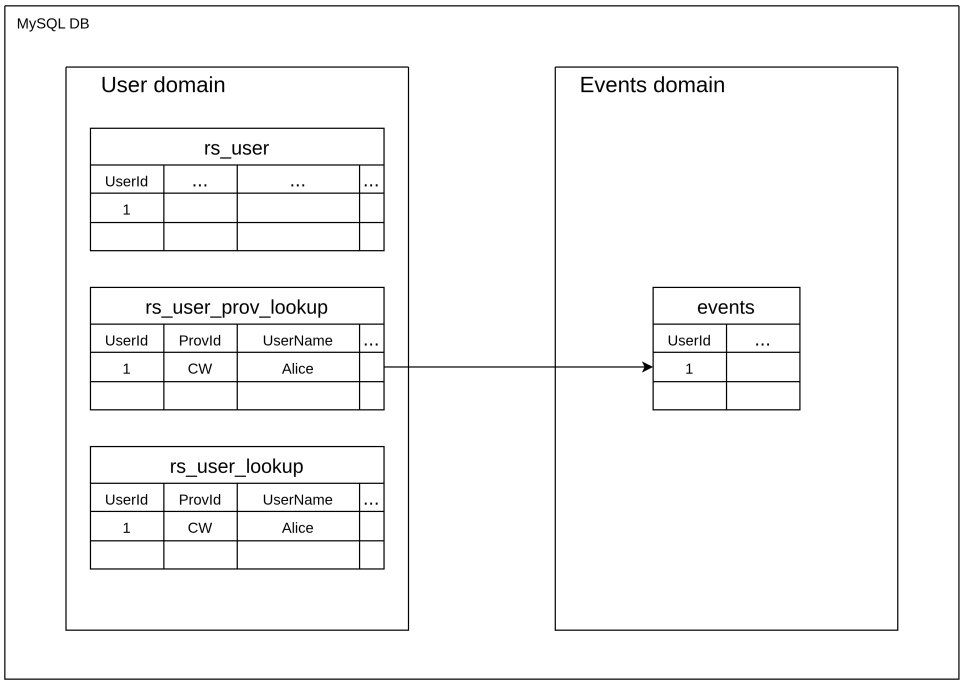
During a migration, one may be tempted to re-implement this feature in the new user management service immediately. However, thinking with a microservices mindset requires stepping back and questioning whether the user management service is the right owner of this functionality at all. The legacy implementation is tightly coupled to the shared database and relies on an internal naming convention (`_CLEAN_`) that only makes sense within the monolith's own data access layer.

The user domain, events domain and many others, in the current implementation are implicitly communicating by sharing the same internal ids. Only through this, the task will be able to eliminate correctly the history, by just running a database transaction. When designing microservices, the made assumption is not correct.

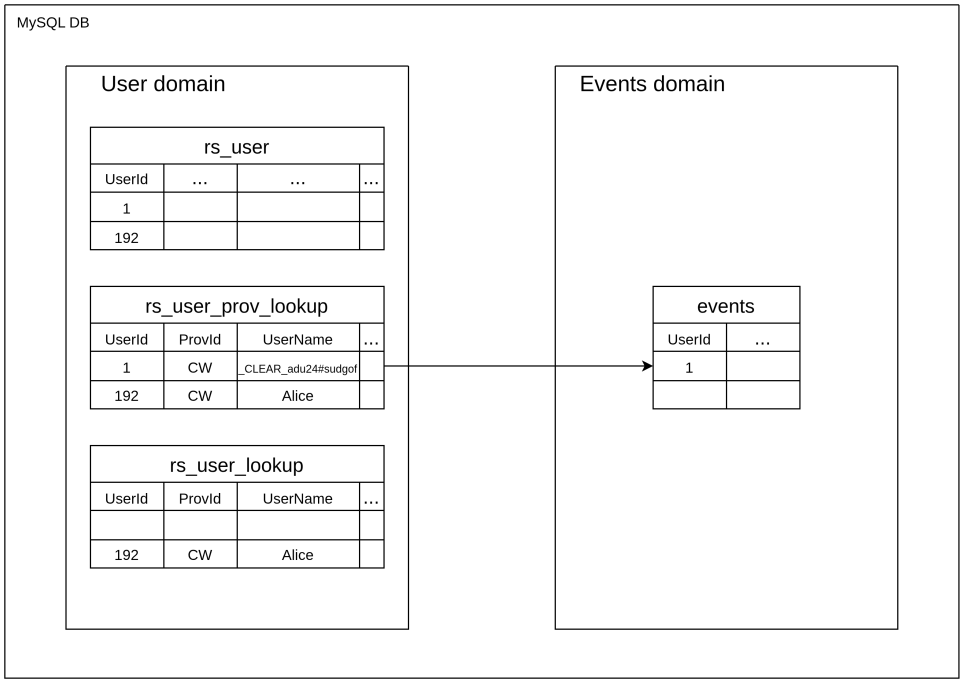
Reproducing this mechanism in a new service would carry the same fragility into the new architecture without addressing any of the underlying problems. Keeping the logic as is would create a structural obstacle when extracting the user and events data layer from the existing monolith. Carrying forward implementation debt of this kind is particularly costly in a migration context, where each service boundary that is drawn incorrectly must later be redrawn at higher effort and risk.

The decompose-by-subdomain pattern [14], grounded in Domain-Driven Design, prescribes that each service should own the data and operations that belong to its bounded context. Applying this lens, delete user history is not a clean fit for the user management service: the bulk of what needs to be deleted is found in many other business domains. The operation is inherently cross-domain.

The architecturally correct solution is therefore to treat delete user history as a coordinated workflow spanning multiple services, rather than a single-service operation. The Saga pattern [13] addresses distributed transactions by decomposing them into a sequence of local transactions, each owned by a single service, with compensating transactions that



(a) Initial state of the schema



(b) The user id is modified by renaming the old record and inserting a new user with the same name. The user is removed from the rs_user_lookup table, but not from the rs_user table. The events are left intact until the task is manually run

Figure 2.9: Delete user history: two-phase logical deletion. Hides the user by renaming their identifier and immediately re-registering them with a fresh internal ID. Then restores consistency by deleting the orphaned event rows left behind by the old ID.

undo completed steps in the event of a downstream failure. In the choreography variant, services react to events published by the previous step; in the orchestration variant, a central coordinator explicitly directs each participant.

Applied to this operation, the user management service would delete explicit preferences, the event service would delete event rows, and a coordinating process would ensure both steps complete or are compensated consistently. This eliminates the dependency on a shared database and on service-private naming conventions such as `_CLEAN_`, and makes the workflow extensible: if additional subdomains accumulate user history data in the future, a new participant can be added without modifying existing services.

Alternatively, if one considers the user history as part of the user domain, there is also the possibility of including the interface into the user management service, and using it as the initiator of the saga transaction.

However, the team decided to implement the feature in the new service using the same `_CLEAN_` mechanism, as it is unclear if the migration will ever bring services to own their data. This is a defensible choice under that assumption: when all services read and write the same schema, there is no distributed transaction problem, and the `_CLEAN_` convention is simply a naming contract within a single database. The Saga pattern would definitely be a plus, but it also requires more resources to be implemented. What matters is that the decision is conscious and documented, accepting a shared database is a deliberate trade-off between bounded-context isolation and operational simplicity [11].

The analysis above remains valuable regardless of outcome: surface indicators such as the endpoint URL and the tables touched during the operation point squarely toward the user domain and could easily mislead a developer into treating this as a straightforward user feature to migrate. Stepping back to examine data ownership across subdomains reveals the cross-domain nature of the operation and prevents a service boundary from being drawn in the wrong place.

2.4.8. Create User Preference

The create user preference operation updates the metadata of an existing user by writing to the appropriate column of the `rs_user` table, consistently with the update mechanism described in Section 2.3. Preferences are modeled as metadata attributes prefixed with `Pref` and are stored as array-encoded strings in the form `#p1=2.0#p2=5.0#`. In this representation, `p1` and `p2` denote items, entities, or areas of interest associated with the user, while the corresponding numeric values, ranging from 1 to 5, express the strength of the user's preference.

The endpoint in the new microservice preserves the observable behavior of the legacy implementation, but its internal design has been streamlined considerably. Superfluous calls and much of the convoluted control flow inherited from the monolith were eliminated, producing a more concise, understandable, and maintainable solution.

2.4.9. Get User Preference

The preferences described in Section 2.4.8 are referred to as *explicit preferences*, as they originate from ratings or feedback that the user provides directly. However, they represent only a portion of the preference information used by the system. A large part of the user profile is derived from *implicit preferences*, which are inferred from user interactions with the platform. Examples include actions such as clicking on a movie, the amount of time spent watching specific content, and other behavioral signals.

The Contentwise platform provides an endpoint to retrieve these inferred preferences. Unlike explicit preferences, however, implicit preferences are not stored persistently in the database. Instead, they are computed dynamically at runtime by the recommendation engine. Migrating this functionality would therefore require migrating significant parts of the recommendation system itself.

From an architectural perspective, strong coupling between components often indicates that they should remain within the same service boundary. Extracting this functionality into the new user management service would have required replicating a large portion of the recommendation logic, mixing two distinct subdomains. Because of this tight coupling, the decision required careful evaluation by team members with extensive knowledge of the system.

For this reason, the implicit preference endpoint in the new service was implemented as a synchronous call to the monolithic application. The microservice forwards the request to the monolith and returns the response unchanged to the client.

The retrieval of explicit preferences, on the other hand, was implemented in the new service using a mechanism similar to the one used for retrieving user data.

2.4.10. Delete User Preference

This endpoint was implemented following the same approach used in the monolithic system. The service first retrieves the corresponding row from the `rs_user` table, modifies the relevant attribute, and then persists the updated record back to the database. To avoid concurrency issues such as lost updates, the operation is executed within a locking

mechanism that ensures exclusive access to the row during the update.

2.5. Deployment

The user management service was deployed to a development environment hosted on Amazon Web Services (AWS), leveraging the Elastic Kubernetes Service (EKS). The primary objective of this deployment was to validate the service in a realistic network environment, supporting the testing activities described in Chapter 3. Accordingly, the architecture was intentionally simplified relative to what a production environment would require. In this configuration, all inbound user requests are received by an Apache HTTP Server acting as a reverse proxy, co-located on a single all-in-one EC2 instance alongside the monolithic application and the database. Apache routes user-related API calls to the user management service running on EKS, which in turn accesses the database on the same EC2 instance. This colocation introduces a routing overhead that is acceptable in a testing context but would represent an architectural liability in production. A production-grade deployment would require dedicated, independently scalable instances for the load balancer, the database, and the monolith, eliminating the tight coupling introduced by the all-in-one EC2 instance and reducing the latency introduced by the current routing path.

2.5.1. Containerization

The service is packaged as a Docker container image built from a Spring Boot application. The containerization process follows a standard pattern: the application is compiled into an executable JAR file using Maven, then packaged into a Docker image based on Amazon Linux 2 with Amazon Corretto JDK 17. The resulting container image is pushed to AWS Elastic Container Registry (ECR), from where it can be deployed to the Kubernetes cluster.

2.5.2. GitOps-Based Deployment

The deployment strategy adopts GitOps principles using ArgoCD as the continuous deployment tool. In a GitOps workflow, the desired state of the system is declared in Git repositories using Kubernetes manifests and Helm charts. ArgoCD continuously monitors these repositories and automatically synchronizes the actual cluster state with the declared desired state.

This approach provides several advantages. First, Git becomes the single source of truth

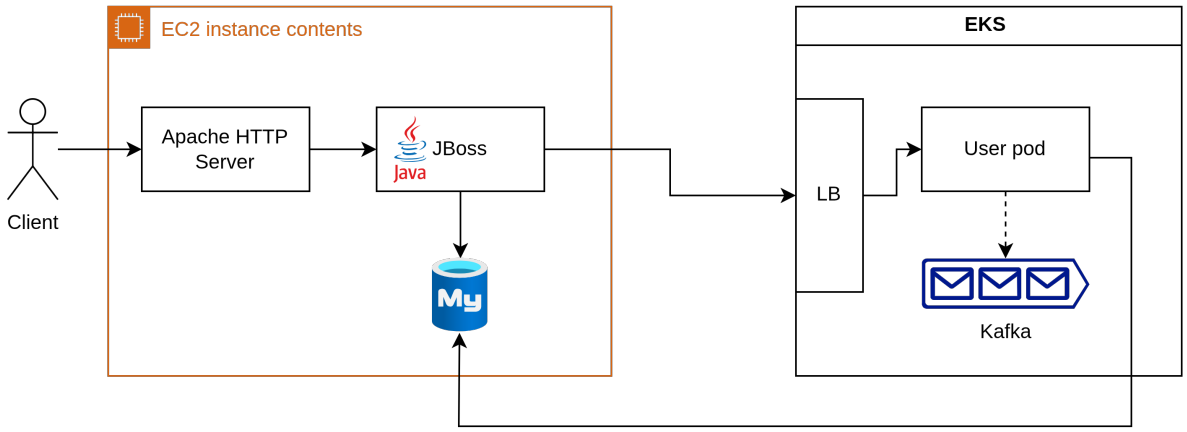


Figure 2.10: Architecture used for testing, combining a allin1 ec2 instance and kubernetes cluster on eks

for infrastructure and application configuration, enabling version control and audit trails for all changes. Second, deployments become declarative and reproducible, as any divergence between the repository and the running system is automatically detected and corrected. Third, rollback operations are simplified to reverting Git commits, rather than executing complex manual procedures.

The infrastructure repository contains environment-specific configurations organized by deployment stage. The develop environment configuration includes Helm chart definitions that specify resource allocations, environment variables, service endpoints, and networking policies for the user management service.

Client requests to the user management service are routed through a single-tier architecture. At the edge, an Apache HTTP Server acts as the frontend reverse proxy, receiving incoming HTTP requests and forwarding them to the appropriate backend services. The Apache server is configured to route requests based on URL paths, directing user-related API calls to the Kubernetes service endpoint.

The deployment was carried out in a development environment with the sole purpose of supporting the testing activities described in Chapter 3. The monolithic application and the database were deployed on a single all-in-one EC2 instance, while the user pod was deployed on Amazon EKS and connected to Kafka. For database access, the user pod relies on the all-in-one EC2 instance. The deployment is shown in 2.10.

2.5.3. Observability

A fundamental feature needed in any production service is a good logging strategy, in order to record system/application activity for debugging and troubleshooting, monitor performance, enhance security (auditing/breach detection), track user behavior for business insights, and ensure compliance. The logging strategy was divided in application logging and auditing. The auditing is done asynchronously into file, while the application logs are simply printed to standard output synchronously. Fluentbit will then be used to monitor the standard output through its event driven architecture and will forward the logs to splunk and persistent volumes on aws

3 | Testing

3.1. Unit tests

The test suite targets components where correctness can be verified without external infrastructure. This boundary was a deliberate design choice: the repository layer depends on `EntityDBHelper`, a proprietary framework class that cannot be instantiated without a live database connection and is not abstracted behind an interface, making mock-based testing infeasible. Its correctness is instead validated indirectly by the characterization tests described in Section 3.4.

Unit tests (JUnit 5 + Mockito) cover the pure business logic:

- **mergeUser**: the merge logic applied during user metadata updates. Each attribute is handled by one of three strategies: scalar overwrite, array union, or discard. All meaningful branches (null inputs, duplicates, type-based dispatch) are covered.
- **Kafka services**: Kafka interaction is mocked via `KafkaTemplate` stubs. Tests verify message structure, UUID uniqueness, timestamps, and failure handling. The message format is a contract with downstream consumers, making isolation testing particularly valuable here.

3.2. Functional API tests

The existing ContentWise functional test suite was run against the new user management service as a first validation layer. The suite is composed of REST-assured integration tests that exercise the four core user operations: create, get, update, and delete. They cover both the happy path and the main edge cases. Each test asserts on the HTTP response body as well as on the resulting database state, querying the MySQL schema directly through JDBC utilities. Passing this suite confirmed that the new service honors the same observable contract as the legacy monolith for all tested scenarios.

3.3. Performance tests

Between the architectural characteristics, the performance tests aim to indicatively measure the following:

- Response time
- Throughput
- Scalability
- Load handling capacity
- Endurance
- Error Rate
- Resource Utilization

The project did not specify explicit metrics and intervals, except that they should be at least comparable to the legacy code. The tests were executed in the development environment presented in Section 2.5. Gatling [7] was set up to test one endpoint at a time.

Request payloads were constructed by sampling random rows from CSV files, ensuring that each request was valid and expected to succeed while still exercising a realistic spread of inputs. Care was taken to use a sufficiently large user population: too narrow a set of users would have made cache hits dominant, producing latency figures lower than those observed under real traffic and masking the true cost of database reads.

The load was introduced gradually, following a ramp-up profile rather than an immediate spike, as shown in Figure 3.1. This approach reflects standard load-testing practice: an abrupt burst of traffic would produce an artificially pessimistic picture by overwhelming the JVM warm-up phase, the connection pool's initial provisioning, and any lazy-initialization logic. Raising the arrival rate incrementally instead exposes the *natural* operating curve of the service.

The primary goals of this phase were:

- **Heap stability / memory leaks.** Monitoring JVM heap usage over the duration of the ramp reveals whether allocations grow unboundedly, or stabilize at a plateau consistent with the current load level.
- **Single-pod saturation point.** By continuing to increase the request rate until throughput stops growing or latency sharply degrades, it is possible to identify

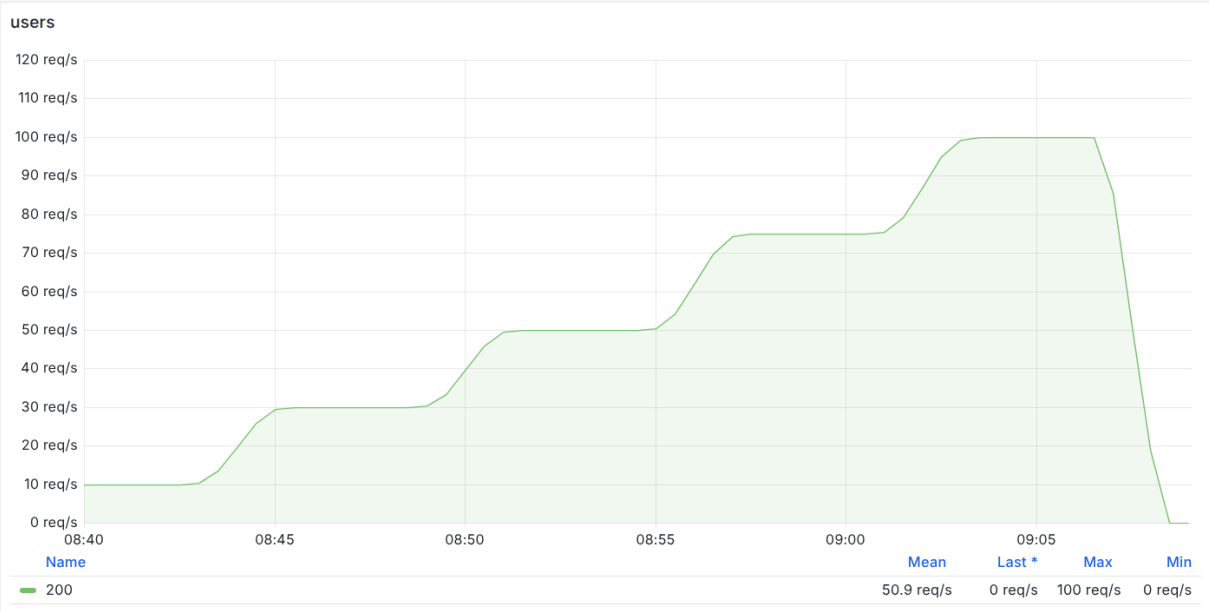


Figure 3.1: testing ramp

the maximum sustainable throughput of a single pod before horizontal scaling is required.

- **Connection handling.** Observing whether TCP connections are dropped, queued, or refused at high concurrency indicates whether the connection-pool configuration and the Kubernetes readiness/liveness probes are tuned appropriately.

The response times were measured by a monitoring tool, in this case it was Grafana [8]. The time difference between the arrival of a request to the Apache HTTP server and the response on the same machine was considered as latency. In the case of the legacy service, no network is required, as it is possible to see in the deployment in Figure 2.10.

3.3.1. Overall results

Across the five tested endpoints the new microservice delivered performance comparable to the legacy monolith. This result is particularly significant given that the legacy system was running the monolith locally, communicating with the database over localhost with essentially zero network overhead. The new service, by contrast, is deployed as a containerized workload on Kubernetes, with requests traversing the cluster network and the Kubernetes service layer. The fact that latency and throughput remain in the same order of magnitude under these conditions confirms that the architectural overhead of the migration is negligible.

Table 3.1 summarises the key metrics for each endpoint.

Table 3.1: Performance testing results: maximum throughput and response time percentiles for each endpoint. Response time is recorded at the API gateway and represents the duration between receiving a request and sending the response.

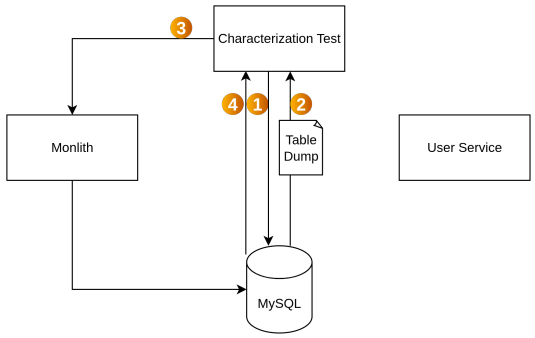
Endpoint	Service	p50 (ms)	p95 (ms)	p99 (ms)
Create User	Legacy	6.7	10.9	15.7
	New	8.7	13.3	14.7
Get User	Legacy	1.6	4.1	4.9
	New	5.4	8.2	11.1
Update User	Legacy	7.8	12.1	16.8
	New	8.3	9.9	11.0
Create Preference	Legacy	1.8	4.8	13.7
	New	9.3	14.3	14.9
Get Preference	Legacy	3.8	6.8	10.2
	New	8.0	10.0	17.0

Both services handle well above the production peak of approximately 20 req/s: the legacy reached 70 req/s and the new service was tested up to 300 req/s in a stress test, with a negligible error rate throughout. Write latency is closely matched (p50 around 9 ms, p99 below 17 ms for both). Read latency is modestly higher on the new service (p50 5–8 ms versus 1.6–3.8 ms on the legacy), a direct consequence of the intentional removal of the per-user cache for correctness reasons (Section 3.4). Latency bands remain stable across all load steps, with a brief startup spike attributable to JVM warm-up that settles immediately.

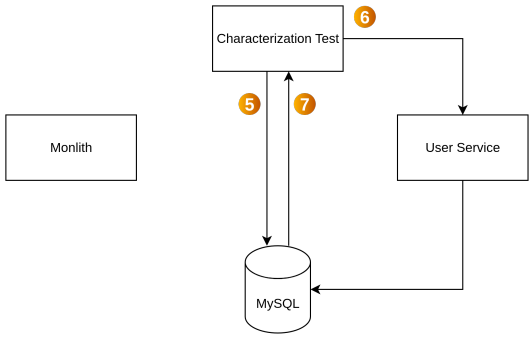
3.4. Characterization Tests

Passing the functional tests (Section 3.2) was clearly not enough to guarantee backward compatibility with the legacy service. It is often the case that, during implementation, certain behaviors emerge unexpectedly, without the developer being aware of them. The client may depend on these features (or bugs). For this reason, during a migration it is important to be aware of all possible edge cases.

Characterization tests, also known as *golden master tests*, are a testing technique introduced by Michael Feathers in the context of working with legacy code [5]. The idea is to put in place a mechanism that spots differences from the legacy system's current behavior, rather than testing against explicit requirements. The most straightforward way to write them would be to access the legacy code repository, write a test case for a specific edge case of an endpoint, and check the result. Unfortunately, the responses returned by the migrated endpoints are not descriptive of the internal state change of the database.



(a) Phase 1 — legacy service: send requests and capture reference snapshot



(b) Phase 2 — new service: restore, replay, capture and compare

Figure 3.2: Characterization test workflow. Both phases operate on the same database instance: Phase 1 produces the reference snapshot from the legacy service; Phase 2 restores the initial state, replays the identical request sequence against the new service, and compares the resulting snapshot against the reference.

Only the GET endpoint can give insights into the internal state, but it is insufficient: as explained in Section 2.3, a total of 4 tables are affected in the database, whereas the GET endpoint depends only on a subset of them. The decision was to test the system as a grey box, monitoring only the database, since the business logic is solely responsible for data manipulation and database communication.

The final implementation is done is a separate service that uses spring boot and works by monitoring the involved tables through snapshot comparison. The workflow, illustrated in Figure 3.2, consists of two phases. In the first phase, the database is cleaned up (1), a dump of the monitored tables is taken (2), the requests are sent to the legacy service (3), and a snapshot of the resulting state is captured (4). In the second phase, the database is restored from the dump (5), the same requests are replayed against the new service (6), a second snapshot is taken (7), and the two snapshots are compared — the test succeeds if they are equal, modulo the columns listed in the exclusion list.

The correctness of the comparison relies on the following assumptions about the environment and the behavior of the tested services.

1. **Shared database.** Both services connect to the same MySQL instance; the framework takes a single snapshot per phase and cannot reconcile separate schemas.
2. **Deterministic replay.** Rows are compared positionally after sorting by primary key, so the same request sequence must produce the same rows in the same order. Non-deterministic ID generation causes spurious failures.

3. **Complete cleanup.** The cleanup SQL must fully reset all touched tables before each run. Test users are identified by the `gm_` prefix, which all test scenarios must use.
4. **Synchronous persistence.** The `quiesce-delay-ms` parameter is the only wait mechanism for asynchronous side effects such as Kafka-driven writes. All tested operations are assumed to complete their database writes before returning an HTTP response.
5. **Stable exclusion list.** Columns in `excluded-columns` (`insertts`, `updatets`, `userid`, `providerid`, `userhashid`) are fully enumerated. Any auto-generated column omitted from the list will cause spurious failures on every run.
6. **No cache leakage between phases.** After the database restore, the new service must start from the same logical state as the legacy service. In-memory caches that survive the restore and are not flushed between phases will cause divergent writes that are indistinguishable from real bugs.

3.4.1. Extensibility to other services

Despite prioritizing simplicity, the framework is sufficiently configurable to be reused across the Contentwise migration, where many services share the same MySQL instance. Adopting it for a different service requires no code changes, only configuration and test wiring.

Externalised database configuration All schema-specific knowledge: watched tables, excluded columns, cleanup SQL, and timing parameters is declared in `application-db.yml` rather than hardcoded, so the same binary can be pointed at a different schema without modification.

Decoupled request generation Request generation is decoupled behind a `RequestGenerator` interface. The built-in `CsvRequestLoader` handles the common case of replaying a pre-recorded HTTP sequence; services with more complex requirements can substitute a custom implementation without touching framework code.

3.4.2. Results

The usefulness of this tool was clear from the outset: several undocumented behaviors were identified early on. The requests were generated by hand, targeting edge cases with unspecified behavior. Two representative findings are described below.

Malformed metadata field. A request to update user metadata contained an incorrectly formatted `UserCountry` field, a boolean value instead of a list of countries:

```
{"metadata": {
    "UserCountry": true,
    "Technology": ["cable"]
}}
```

Both services correctly rejected the insertion of the malformed field. However, the monolith additionally rejected the insertion of the correctly formatted field, whereas the new service inserted it and failed silently on the invalid attribute. This could be an intentional design decision, but a more in-depth analysis would be needed to determine whether any clients depend on this behavior. The final decision was to align the new microservice with the legacy behavior.

Cache inconsistency across instances. A further source of discrepancies was the caching mechanism used by the new service to reduce the frequency of database reads. When multiple instances ran in parallel, a user that had just been created was not yet visible to other instances, causing the service to report it as non-existent. The tradeoff between consistency and performance was acceptable in other contexts, but not in this case, where correctness was the primary requirement. The decision was therefore to remove the per-user cache and rely entirely on the database to determine whether a user exists.

3.5. Static Code Analysis

To empirically validate the quality improvements achieved through the migration, both the microservice and the monolith were analyzed using SonarQube, a widely-adopted static analysis platform that evaluates code quality across multiple dimensions including complexity, maintainability, security, and reliability.

3.5.1. Analysis Methodology

The analysis compared the newly developed user microservice against the monolith codebase. To account for differences in codebase size, all metrics were normalized per 1000 lines of code, enabling meaningful comparison between systems of vastly different scales. The microservice contains 6333 lines of code, while the monolith comprises 255645 lines.

3.5.2. Quality Metrics Comparison

Table 3.2 presents the normalized comparison of key quality metrics between the two systems.

Table 3.2: Static Analysis Comparison: Microservice vs Monolith (normalized per 1K LOC)

Metric	Microservice	Monolith	Improvement
Cognitive Complexity	76.1	192.6	60% ↓
Cyclomatic Complexity	105.2	189.2	44% ↓
Code Smells	12.2	73.9	84% ↓
Technical Debt (hours)	2.2	8.3	73% ↓
Code Duplication	3.5%	16.2%	78% ↓
Ratings (A=best, E=worst)			
Security Rating	A (1.0)	E (5.0)	Best
Reliability Rating	A (1.0)	E (5.0)	Best
Maintainability Rating	A (1.0)	D (4.0)	Best

3.5.3. Results Interpretation

The static analysis reveals substantial quality improvements across all measured dimensions. The 60% reduction in cognitive complexity demonstrates that the microservice is significantly easier to understand, while the 84% reduction in code smells reflects superior adherence to modern design patterns such as dependency injection and SOLID principles. The 73% decrease in technical debt indicates that future modifications will require substantially less effort compared to equivalent changes in the monolith. The microservice achieved perfect A ratings for security, reliability, and maintainability, compared to the monolith's E ratings, indicating the absence of known vulnerability patterns and bug-prone code.

3.5.4. Limitations and Caveats

While these metrics demonstrate measurable improvements, the results should be interpreted as indicative rather than conclusive. The comparison involves a newly developed microservice implementing a focused subset of functionality against a large legacy system that has accumulated technical debt over more than 15 years of organic growth. Several factors contribute to the observed differences beyond architectural choices alone.

First, the microservice benefits from modern technology advantages, which provide better language features, libraries, and development tools. Second, greenfield development nat-

urally produces cleaner code than maintaining legacy systems, as developers can apply current best practices without navigating existing constraints. Third, the microservice's smaller scope (6333 lines versus 255645 lines) inherently reduces complexity, as it addresses a single bounded domain rather than the monolith's numerous interconnected concerns.

These metrics validate that the migration approach successfully produced higher-quality code for the extracted functionality. However, they do not predict whether this quality will be maintained as the service evolves, nor do they account for the increased operational complexity of managing distributed systems. The results provide empirical evidence supporting the migration strategy while acknowledging that long-term maintainability depends on continued adherence to engineering discipline and appropriate architectural boundaries.

3.6. Summary

The three testing layers together provide strong evidence that the new user microservice is a faithful and production-ready replacement for the corresponding functionality in the legacy monolith. Unit tests verified the correctness of the core business logic in isolation. Functional tests confirmed that the service honors the same observable API contract as the legacy system across all primary user operations and their main edge cases. Characterization tests went further, surfacing undocumented legacy behaviors that would not have been caught by specification-based testing alone, and ensuring the new implementation matches the monolith's database-level side effects precisely. Performance results showed that the new service operates within the same latency and throughput range as the legacy system despite the added network overhead of a containerized Kubernetes deployment, and handles loads well above the current production peak. Finally, static analysis provided empirical confirmation that the migration produced measurably higher-quality code across all evaluated dimensions.

4 | Conclusions and future developments

4.1. Conclusions

This thesis presented the incremental extraction of the user domain from the UX-Engine monolith into a dedicated Spring Boot microservice, guided by the Strangler Fig pattern and Domain-Driven Design. The migration focused on isolating core user functionality while preserving the existing database and external behavior of the legacy system.

The main objective was achieved: the selected user-related endpoints were successfully moved to the new service, containerized, and deployed in an AWS EKS development environment through a GitOps workflow. Functional and characterization tests showed that the new implementation preserves the expected behavior of the monolith and helped reveal undocumented legacy behaviors that had to be considered during the migration.

Performance results show that the extracted service remains comparable to the legacy implementation, while supporting loads well above the current production peak. Static analysis also indicates a clear improvement in maintainability and code quality compared to the monolithic implementation.

4.2. Future Developments

4.2.1. Check monolith dependencies on the user management service

As shown in the example presented in Section 2.4.2, some parts of the monolith still depend on the user module. It is therefore important to actively refactor the monolith to remove internal calls to the user management service and to handle these dependencies asynchronously whenever possible, thereby reducing coupling and avoiding inconsistencies.

4.2.2. Delete User History

The migration chapter (Section 2.4.7) analyzed the delete-user-history endpoint as an inherently cross-domain operation and left it untested, pending a deliberate architectural decision. The correct path forward depends on whether the shared MySQL instance is ever decomposed into per-service databases.

If database separation is pursued, the Saga pattern remains the appropriate solution: the user management service deletes preferences, the events service deletes event rows, and a coordinator ensures both steps complete or are compensated. This avoids encoding the `_CLEAN_` naming convention into the new architecture and preserves clean bounded-context boundaries.

If the shared database is retained as a permanent constraint, the picture changes. The Saga pattern exists to coordinate writes across independent data stores; when all services share a single schema, there is no distributed transaction problem. In that case, reimplementing the `_CLEAN_` mechanism is architecturally acceptable: the rename and deferred cleanup are simply a SQL-level operation executed within the shared schema, and the cross-domain coupling that made the pattern dangerous in a microservices context does not materialize. Newman [11] explicitly acknowledges the shared database as a valid intermediate architectural state, not merely a failure mode. The critical requirement is that the choice be made consciously and documented: accepting a shared database is a deliberate trade-off between isolation and operational simplicity, not an oversight.

4.2.3. Implement the infrastructure required for the outbox pattern

During the migration, the number of workflows that both modify the database and publish a message to Kafka will continue to grow. As a result, introducing an outbox table in the database schema will eventually become necessary. However, since the database is already an active bottleneck, adding an outbox table may further increase the load on it. In that case, the migration strategy should be reconsidered by first separating the data domains, making it possible to introduce an independent database instance and reduce the pressure on the current write node.

4.2.4. Event listener for subdomain updates

At present, subdomains are retrieved from the cache. As a consequence, even if the subdomains are updated, the service remains unaware of those changes until it is rescheduled.

This limitation is caused by a dependency on the existing caching library. The ideal solution would be to modify the monolith logic responsible for updating subdomains so that it also produces a Kafka message. The user management service could then consume that message and refresh the subdomains, using the database as the source of truth.

4.2.5. Production deployment

A gradual deployment strategy would be valuable to validate the correctness of the service under real production traffic. In this approach, the service would be deployed in production without immediately handling all requests, thereby reducing the risk of issues caused by real-world usage patterns.

4.2.6. Complete the observability pipeline

The logging infrastructure is partially in place: application logs are currently printed to standard output and auditing is handled asynchronously to file. The next step is to deploy Fluentbit as a log forwarder, monitoring the standard output of each pod through its event-driven architecture and forwarding logs to Splunk and persistent volumes on AWS. This would complete the observability pipeline and make the service production-ready from a monitoring and auditing standpoint.

4.3. Final considerations

The architectural concern underlying all the decisions discussed in this thesis — shared database, deferred Saga implementation, absence of CDC support — points to a more fundamental issue: the MySQL database is itself the primary bottleneck of the UX-Engine platform.

This became apparent during the project: the database instance could not be reconfigured to use full binary logging (`binlog_row_image = FULL`), a prerequisite for Change Data Capture tools such as Debezium and, consequently, for the transactional outbox pattern. The restriction was not a configuration oversight but a sign of an already-strained system operating near its practical limits. The microservices architecture will eventually need this feature, that will become painful to implement.

This has a direct implication for the migration strategy. Decomposing the application layer while retaining a shared database does not resolve the bottleneck, but worsen it. Operations that were previously in-process function calls now cross a network boundary, adding latency and potentially triggering additional database reads that the monolith

could have avoided through in-process state.

If the long-term goal is to relieve load on the database, migrating services in isolation is insufficient. The database itself must either be sharded, or decomposed so that each service owns its data. The latter is architecturally correct but represents a substantially larger investment and greater architectural maturity. The improvements introduced in this migration could have been obtained by rebuilding the monolith into a modular version.

None of this invalidates the work done. The improvements in code quality, maintainability, and independent deployability of the application tier are real and documented. But they address a different class of problem from the performance bottleneck. Recognizing this distinction is important for setting realistic expectations about what a service-oriented architecture with a shared database can and cannot deliver.

Bibliography

[1] Law of holes. Wikipedia, 2024. Available at: https://en.wikipedia.org/wiki/Law_of_holes.

[2] daily.dev. Listen to yourself pattern: Is it an alternative to the outbox pattern? Online, 2024. Available at: https://app.daily.dev/posts/QOCkAWO8e.

[3] Debezium Community. Debezium connector for mysql, 2026. URL https://debezium.io/documentation/reference/stable/connectors/mysql.html.

[4] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003. ISBN 978-0321125217.

[5] M. C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 978-0131177055.

[6] B. Foote and J. Yoder. Big ball of mud. *Pattern Languages of Program Design*, 4: 654–692, 1997. Available at: http://www.laputan.org/mud/.

[7] Gatling Corp. Gatling documentation. https://docs.gatling.io, 2025. Accessed: April 2025.

[8] Grafana Labs. Grafana: The open observability platform. Online, 2024. Available at: https://grafana.com.

[9] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Sebastopol, CA, 2017. ISBN 978-1449373320.

[10] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Sebastopol, CA, 2015.

[11] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Sebastopol, CA, 2019.

[12] Red Hat. JBoss Enterprise Application Platform. https://www.redhat.com/en/technologies/jboss-middleware/application-platform. Accessed: 2025.

[13] C. Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, Shelter Island, NY, 2019.

[14] C. Richardson. Pattern: Decompose by subdomain. microservices.io, 2024. Available at: `https://microservices.io/patterns/decomposition/decompose-by-subdomain.html`.

[15] Spring Framework. Spring data jpa. Online, 2024. Available at: `https://spring.io/projects/spring-data-jpa`.

[16] Supermegapotter. Read your own writes. Medium, 2024. Available at: `https://medium.com/@supermegapotter/read-your-own-writes-391b1fded68b`.

[17] VMware, Inc. Spring boot. `https://spring.io/projects/spring-boot`, 2026. Accessed: 2026-03-30.

[18] Wikipedia contributors. Entity–attribute–value model — Wikipedia, the free encyclopedia, 2025. URL `https://en.wikipedia.org/w/index.php?title=Entity%E2%80%93attribute%E2%80%93value_model&oldid=1318918750`.