TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Vadim Filonov 223105IAIB

Developing a Scalable Entertainment Platform with Spring Cloud: A Full Cycle from Design to Implementation

Bachelor's Thesis

Supervisor: Ali Ghasempour
MSc

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Vadim Filonov 223105IAIB

Skaleeritava mikroteenuse rakenduse arendamine ja juurutamine Spring Cloudi abil: täielik elutsükkel projekteerimisest tootmiseni

Bakalaureusetöö

Juhendaja: Ali Ghasempour

MSc

Tallinn 2025

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Vadim Filonov

04.06.2025

Abstract

Modern entertainment platforms must support high user loads, deliver real-time features, and remain both maintainable and scalable in rapidly changing environments. As industry giants like Netflix and Uber have shown, monolithic architectures often become bottlenecks, limiting productivity, and hindering rapid innovation. These limitations have led many companies to undergo complex and costly migrations to microservices, often only after critical points of failure have been reached.

The primary goal of this thesis is to provide a comprehensive open source reference solution by designing and implementing a cloud-based entertainment platform built on a microservices architecture using Spring Cloud technologies. By addressing scalability, modularity, and real-time performance issues from the outset, the platform aims to serve as a practical foundation for other developers and organizations looking to build similar systems while avoiding common architectural pitfalls.

The platform includes eight microservices, each responsible for a distinct domain. The backend is implemented using the Spring ecosystem, while Angular-based frontend applications provide dedicated interfaces for both users and administrators. Extensive testing was performed, including unit, integration, performance, scalability, and user testing, to ensure correctness, efficiency, and user experience under realistic conditions.

The results show that the architecture can reliably handle distributed workloads, gracefully recover from service failures, and scale horizontally across services. The entire platform is released as open source software to help other developers and organizations accelerate their own development efforts and avoid the pitfalls of building scalable platforms.

The thesis is in English and contains 43 pages of text, 5 chapters, 3 figures, 1 table.

Annotatsioon

Skaleeritava mikroteenuse rakenduse arendamine ja juurutamine Spring Cloudi abil: täielik elutsükkel projekteerimisest tootmiseni

Kaasaegsed meelelahutusplatvormid peavad toetama suurt kasutajakoormust, pakkuma reaalajas funktsioone ning jääma kiiresti muutuvas keskkonnas nii hooldatavaks kui ka skaleeritavaks. Nagu on näidanud sellised tööstushiiglased nagu Netflix ja Uber, muutuvad monoliitsed arhitektuurid sageli kitsaskohtadeks, piirates tootlikkust ja takistades kiiret innovatsiooni. Need piirangud on pannud paljusid ettevõtteid läbima keerukaid ja kulukaid üleminekuid mikroteenustele, sageli alles pärast kriitiliste rikete saavutamist.

Selle lõputöö peamine eesmärk on pakkuda terviklikku avatud lähtekoodiga lahendust, kavandades ja rakendades pilvepõhise meelelahutusplatvormi, mis on üles ehitatud Spring Cloudi tehnoloogiaid kasutavale mikroteenuste arhitektuurile. Platvorm, mis käsitleb skaleeritavuse, modulaarsuse ja reaalajas jõudluse küsimusi, on mõeldud praktiliseks aluseks teistele arendajatele ja organisatsioonidele, kes soovivad luua sarnaseid süsteeme.

Platvorm sisaldab kaheksat mikroteenust, millest igaüks vastutab kindla valdkonna eest. Tagaserver on rakendatud Springi ökosüsteemi abil, samas kui Angularil põhinevad esiotsa rakendused pakuvad kasutajaliideseid. Õigsuse, tõhususe ja kasutajakogemuse tagamiseks viidi läbi ulatuslik testimine realistlikes tingimustes.

Tulemused näitavad, et arhitektuur suudab usaldusväärselt hakkama saada hajutatud töökoormustega, sujuvalt taastuda teenusetõrgetest ja skaleeruda horisontaalselt teenuste vahel. Kogu platvorm avaldatakse avatud lähtekoodiga tarkvarana, et aidata teistel arendajatel ja ettevõtetel oma arendustegevust kiirendada ja lõkse vältida.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 43 leheküljel, 5 peatükki, 3 joonist, 1 tabel.

List of abbreviations and terms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
DTO	Data Transfer Object
EKS	Elastic Kubernetes Service
ELK	Elasticsearch, Logstash, Kibana
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
OAUTH	Open Authorization
REST	Representational State Transfer
RPS	Requests Per Second
S3	Simple Storage Service
SMTP	Simple Mail Transfer Protocol
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Table of contents

List of figures

List of tables

1 Introduction

Many of today's most successful technology companies, such as Netflix, Amazon, and Uber — began with monolithic architectures that met their short-term goals. However, as their platforms scaled, these systems became increasingly difficult to maintain, extend, and operate under variable user loads. Eventually, each company faced critical limitations: long deployment times, tightly coupled components, inflexible scaling, and frequent outages during peak traffic. In response, they transitioned to microservice architectures that allowed them to scale individual components independently, develop and deploy updates faster, and improve overall system resilience [1].

These real-world cases highlight a common challenge: while monoliths can be fast to build, they do not scale gracefully with business growth. This thesis addresses this challenge from the outset. Rather than repeating the mistakes of the past, the goal is to design and implement a scalable entertainment platform that is cloud-native and microservice-based from the beginning, ready to handle dynamic user demand, particularly during high-traffic events such as live esports events.

The system developed in this work consists of eight Spring Cloud [2] based microservices, each responsible for a clearly defined functional domain, such as user authentication, virtual currency management, in-app game mechanics, and real-time betting. Frontend applications for both users and administrators were implemented using Angular, following a clean client-server architecture. Several architectural patterns and technologies were integrated to ensure system resilience and observability. These include Circuit Breakers for fault isolation, the Outbox pattern for reliable event delivery, and distributed tracing for diagnosing service interactions. Logging and monitoring are managed using Prometheus, Loki, and Grafana to provide comprehensive system visibility.

In addition to development, this work involved extensive validation of system behavior and performance. Unit tests were written for individual components, while integration, performance, scalability, and user testing were conducted to evaluate the platform under realistic usage scenarios.

Thanks to the microservice architecture, the system maintains performance and stability under peak loads, which will help prevent users from leaving for competitors due to failures or slow system operation. Moreover, the flexible architecture simplifies the implementation of new features that will not only retain existing users due to stable and interesting functionality, but also actively attract new ones, offering them a modern and dynamically developing service.

Unlike existing platforms, this application combines game mechanics with a real esports market in a way not currently offered elsewhere. A dual-currency virtual economy allows users to gradually earn value through gameplay and use it for real-time betting without the need to deposit real money. This combination of features remains unique in the software landscape today. Furthermore, while several open-source microservice-based systems exist for domains such as e-commerce, there are no comparable open solutions tailored for building scalable, interactive entertainment platforms.

This thesis primarily aims to provide a production-ready open-source implementation of a modern cloud-native entertainment platform. The entire system is released as open-source software, with links included in Appendix 2 – GitHub Repositories, offering a reference architecture that others can learn from, build upon, and adapt to their own needs, helping them avoid the architectural pitfalls that even the world's largest companies once faced.

2 Background

2.1 Microservice Architecture

The microservice architecture is a design paradigm in which software is composed of small, autonomous services, each dedicated to a specific business capability. This approach contrasts with traditional monolithic architectures, where all functionality resides within a single, tightly coupled application. Drawing from Robert C. Martin's Single Responsibility Principle — "gather together those things that change for the same reason, and separate those things that change for different reasons" [3] — microservices extend this principle to the architectural level by enforcing clear boundaries between components. Each service operates in its own process, communicates via lightweight protocols such as HTTP or messaging queues, and can be developed, deployed, and scaled independently.

Microservice architecture is widely adopted in large-scale systems across various industries where flexibility, scalability, and continuous deployment are critical. Examples include e-commerce platforms like Amazon, streaming services like Netflix, and financial systems such as those used by PayPal. These systems benefit from the ability to independently evolve and scale different parts of the application — for instance, user management, payment processing, or content delivery without disrupting the entire system. This modular design facilitates distributed development, encourages technological diversity, and supports resilience in the face of failure. Microservices are especially suited to cloud-native platforms that demand agility, scalability, and fault isolation.

2.1.1 Advantages and Challenges

Microservices enable smaller, autonomous teams to build and maintain services with greater independence. Services can be scaled individually based on demand, allowing high-traffic components to be replicated without affecting the rest of the system, and faults in one service do not necessarily disrupt the entire application. Furthermore, teams can choose the most appropriate technology stack for each service, promoting flexibility and innovation.

Microservices also support faster development cycles and continuous delivery. Because services are loosely coupled, changes to a single service can be deployed without having to completely rebuild the system, reducing deployment risk and allowing for faster release of features. This modularity enables parallel development and makes it easier for new developers to onboard, who can focus on a single, bounded context rather than trying to understand a large, monolithic codebase. Microservices also promote better system organization through domain-driven design, helping to align technical architecture with business capabilities.

In contrast, monolithic systems, where all components are combined into a single deployable unit, are often simpler to develop initially, but become more difficult to scale and maintain as complexity increases. A single failure can bring down the entire application, and even minor changes may require a complete redeployment. Monoliths are also more difficult to scale effectively, since the entire application must be replicated, even if only one part requires more resources.

However, these benefits come with challenges inherent to distributed systems. Managing multiple independently deployed services increases operational complexity, requiring tools for service discovery, routing, configuration management, and health monitoring. Ensuring data consistency is also more difficult than in monolithic systems, since transactions often span multiple services. Observability becomes essential to diagnose issues across service boundaries, which in turn demands centralized logging, metrics collection, and tracing mechanisms. When properly managed, microservice architectures offer a powerful foundation for building scalable, robust systems, but they require significant infrastructure and design discipline to realize their potential.

2.1.2 Lessons from Industry: Netflix's Migration from Monolith to Microservices

A compelling example of the need for a microservices architecture comes from Netflix, an early major adopter of this paradigm. In 2008, Netflix suffered a critical failure that prevented it from sending DVDs to users for several days [4]. This failure exposed the risks of vertically scalable, tightly coupled systems and became the catalyst for a long-term architectural transformation. Rather than "lift and migrate" its monolithic stack to the cloud, Netflix opted for a complete rebuild using cloud-native microservices.

The company spent seven years gradually deconstructing its legacy systems and replacing them with hundreds of loosely coupled services. This change not only improved scalability and fault isolation, but also enabled rapid global expansion. For example, after completing the migration, Netflix was able to support eight times more users and stream video in 190 countries by dynamically allocating computing resources using AWS. Using horizontal scaling and redundancy, Netflix has achieved nearly 100% availability, even during infrastructure outages.

Netflix's experience shows that while monolithic architectures may be simpler to start with, they become brittle and difficult to scale at an enterprise level. Their migration highlights the importance of designing distributed, resilient systems early on — a principle this thesis follows by implementing a Spring Cloud based microservices platform from the start, thereby avoiding the limitations that have held companies like Netflix back for years.

2.2 Technologies for Distributed Systems

To address the complexities of microservice-based development, this thesis relies on technologies from the Spring Cloud ecosystem. Compared to alternative technologies and frameworks, the Spring Cloud stack offers tight integration with the Java ecosystem, a rich set of community-supported tools, and a low barrier to entry for developers already familiar with Spring Boot [5]. Built on Spring Boot, Spring Cloud provides abstractions and tooling to make it easier to solve common microservices problems such as service discovery, routing, load balancing, and fault tolerance.

The backend of the platform consists of eight microservices, each integrated with Spring Cloud components. Spring Cloud Netflix Eureka [6] handles service registration and discovery, enabling dynamic interaction between services. Spring Cloud Gateway [7] serves as the API gateway, managing routing, authorization via JWTs, and request forwarding. Spring Cloud OpenFeign [8] provides a declarative HTTP client abstraction for inter-service communication.

For asynchronous messaging, Apache Kafka [9] and Redis Pub/Sub [10] are used. Kafka enables reliable, high-throughput, event-driven interactions, particularly where message durability and eventual consistency are essential. Redis Pub/Sub supports low-latency messaging, suited for real-time notifications and user feedback.

Fault tolerance is achieved using Resilience4j [11], which introduces circuit breakers and fallback strategies to protect services from cascading failures. Together, these technologies form a cohesive stack that supports the deployment of a robust and responsive backend architecture.

2.3 Cloud-Native Architecture

Cloud-native systems are designed to operate in dynamic, scalable environments where service availability, elasticity, and resilience are paramount. This platform was built with cloud-native principles in mind: services are stateless where possible, containerized for portability, and designed for fault recovery and observability.

Scalability is a core requirement for modern entertainment platforms, which must handle varying user loads, often in real time. [12] Scalability in microservice systems can be approached in two ways:

Horizontal scaling involves adding more instances of a service to distribute the load across multiple nodes. This is ideal for stateless services that can be easily replicated without shared memory or session persistence. Horizontal scaling improves fault tolerance and enables services to continue functioning even if individual instances fail. It also aligns well with containerized environments and orchestration platforms like Kubernetes.

Vertical scaling, on the other hand, refers to increasing the resources (CPU, memory, etc) of a single instance to handle more workload. This is often used for stateful components such as databases, Kafka brokers, or services with intensive computations. While vertical scaling can offer immediate performance gains, it has hardware limits and often requires downtime or restarts, making it less flexible for dynamic scaling scenarios.

The platform's architecture favors horizontal scaling wherever possible to ensure elasticity and high availability. For example, user traffic peaks can be handled by launching additional service instances, while Redis and database nodes may be scaled vertically as needed for performance.

2.4 Database

In line with microservice architecture principles, the platform employs multiple independent PostgreSQL [13] instances, each microservice owning and managing its own database. This approach enforces strong data encapsulation, allowing services to evolve independently without tight coupling at the database layer.

PostgreSQL for Autonomous Microservice Data Management

Each microservice uses its dedicated PostgreSQL database instance or schema, which provides the following benefits:

- **Service autonomy:** Services are responsible for their own data, preventing direct access or interference by other services.
- **Independent scaling and deployment:** Databases can be scaled, backed up, or migrated independently, aligning with service lifecycle and demand.
- **Clear data ownership:** Each microservice controls its schema, data models, and migrations, enabling more agile and safer development cycles.
- **Improved fault isolation:** Failures or performance bottlenecks in one service's database do not directly impact others.

PostgreSQL was selected for these independent instances due to its robust support for:

- ACID transactions that guarantee data integrity in critical domains like user management and payment processing.
- Rich querying and indexing capabilities suitable for complex data operations.
- Extensibility and JSON support, allowing semi-structured data handling when needed without sacrificing relational consistency.

Comparison with Alternative Approaches

Using multiple PostgreSQL instances contrasts with monolithic architectures that typically share a single database, which often leads to tight coupling, complicated schema changes, and scaling difficulties.

Alternative database strategies considered include:

- Shared database schemas (anti-pattern in microservices): simplifies data access but tightly couples services and limits independent deployment.

- NoSQL databases (e.g. MongoDB, Cassandra): offer schema flexibility and horizontal scaling but often lack transactional guarantees crucial for core services.
- Single PostgreSQL with schema separation: improves some isolation, but can still create resource contention and complicate migrations.

The chosen approach balances the need for strong data consistency and service independence, making PostgreSQL multiple-instance deployments ideal for this platform.

2.5 Messaging and Asynchronous Communication

In a microservice architecture, asynchronous communication plays a vital role in improving system scalability, decoupling service dependencies, and improving fault tolerance. Unlike synchronous communication, where services wait for direct responses, messaging allows services to interact indirectly via message brokers or pub/sub systems, leading to improved responsiveness and resilience. [14]

This thesis leverages a combination of Apache Kafka and Redis Pub/Sub to implement asynchronous communication between services, each chosen for specific scenarios based on performance, reliability, and use case fit.

Apache Kafka serves as the primary messaging backbone for durable and high-throughput communication. It is used extensively for event-driven workflows where reliable message delivery and eventual consistency are essential.

Redis Pub/Sub is used for lightweight, low-latency messaging scenarios where persistence is not required. A typical use case includes sending real-time user notifications, such as balance updates. Because Redis Pub/Sub operates entirely in memory and does not store messages, it is ideal for transient, real-time communication where performance and immediacy are prioritized over durability.

These two technologies complement each other: Kafka provides guaranteed delivery and message replay for critical workflows, while Redis enables instantaneous message broadcasting for user-facing feedback. This hybrid messaging strategy allows the platform to balance durability, speed, and complexity, depending on the context.

The adoption of asynchronous communication also improves system decoupling, allowing microservices to evolve, scale, and fail independently. This design aligns with cloud-native principles and supports the development of a resilient and modular architecture.

2.6 Monitoring and Diagnostics

In distributed systems, especially those based on microservice architecture, monitoring and diagnostics are essential for maintaining system health, identifying performance bottlenecks, and enabling rapid incident response. Without proper observability, debugging faults in a dynamic, multi-service environment becomes significantly more complex.

This platform integrates a complete observability stack composed of Prometheus, Loki, and Grafana, providing real-time insight into both infrastructure and application-level metrics.

Prometheus [15] is used to collect and store time-series metrics from each microservice. Metrics such as request latency, error rates, memory usage, and other service insights are exposed via HTTP endpoints and scraped periodically.

Loki [16] is employed for centralized log aggregation. Logs from all services are collected and indexed, enabling efficient search and correlation across the system.

Grafana [17] serves as the visualization layer, offering interactive dashboards that combine logs and metrics into a coherent interface.

With this monitoring and diagnostics stack, the platform provides high observability, helping with both operational readiness and results analysis. This directly contributes to system reliability, maintainability, and the ability to meet user expectations under varying load conditions.

2.7 Frontend Technologies

The frontend layer of the platform is responsible for enabling user interaction with the system through responsive and intuitive interfaces. Two separate Angular-based web applications were developed, one for end users and another for administrators, each tailored to their specific needs.

Angular was selected as the frontend framework primarily due to the author's prior familiarity and experience with it. In addition to personal proficiency, Angular's component-based architecture, TypeScript support, and built-in tooling.

The user interface allows users to register and log in (including via OAuth providers), perform in-app actions such as clicking to earn virtual currency, complete tasks, and place bets on esports events. It communicates with backend services through secure RESTful APIs and WebSockets, enabling real-time data updates such as live betting odds and instant balance changes.

The admin interface is used to manage user accounts and platform content, including esports events, promotional tasks, and user analytics. It also provides real-time dashboards that visualize user activity and service statistics, leveraging data exposed by internal microservices.

Both applications were designed with responsiveness and usability in mind, ensuring compatibility across different screen sizes and devices. Authentication tokens are securely handled in the browser, and role-based access control is enforced via route guards and API-level authorization.

3 Implementation

The implementation phase was driven by a clear architectural vision and an iterative development strategy. It began with thorough planning, including selecting the core technologies that would support a scalable, resilient microservice system. The process followed a modular approach, starting from foundational services and infrastructure, then progressively building domain-specific services, testing, and frontend functionality. Each component was designed and integrated with cloud-native principles in mind, ensuring flexibility, observability, and robustness at scale.

3.1 System Architecture Setup

The design of the system began with an analysis of how microservice-based systems are typically structured, with a strong focus on scalability, modularity, and maintainability. The architecture was laid out following domain-driven design principles, ensuring that each service aligned with a specific business capability.

To support development and deployment workflows, the project was split into three main codebases:

- **Backend monorepo:** contains all microservices, shared configurations, and libraries.
- **Frontend repository:** responsible for the user interface and admin panel functionality.
- **Test repository:** dedicated to integration and end-to-end tests, ensuring each component could be tested both in isolation and as part of the full system.

The initial architecture diagram of the system is shown in Figure 1. It illustrates how the various microservices interact with other components of the system. The API Gateway acts as a single entry point for client interactions with the server, routing requests to the appropriate services registered via the Service Discovery component. Each service runs independently with its own PostgreSQL database, supporting separation of concerns and independent scaling. Supporting components such as Redis [18] for caching, ELK for logging, and Grafana for monitoring are included to ensure observability and performance.

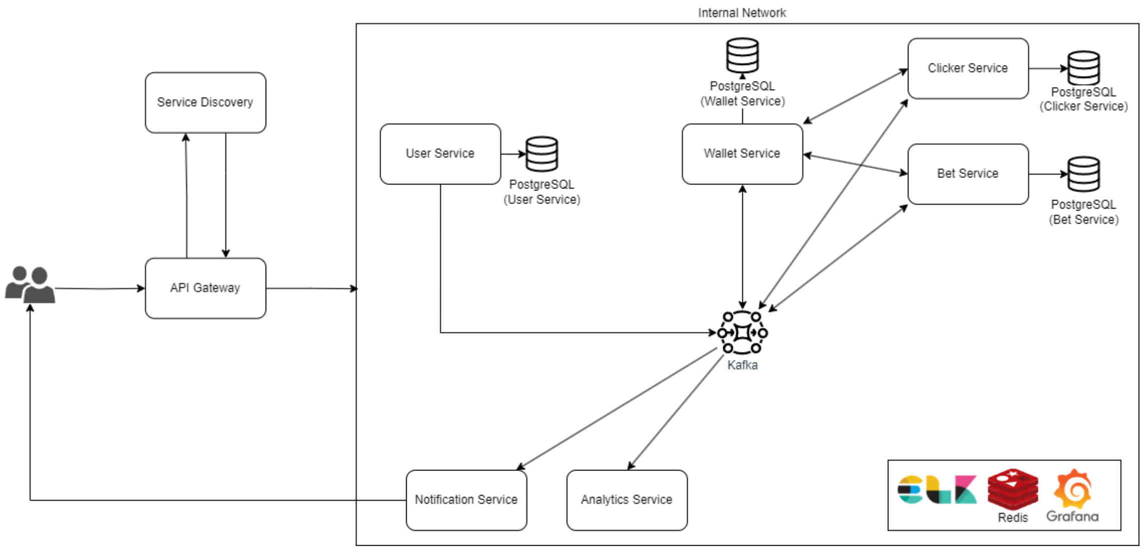

Figure 1. Initial architecture diagram.

3.2 Environment Configuration

The project environment was structured to ensure consistency, flexibility, and ease of deployment across development, testing, and production stages. For the backend, a Maven-based monorepo architecture was adopted. Maven [19] is a build automation and project management tool for Java projects. It organizes code into modules and manages builds through a centralized pom.xml file. In this project, each microservice is registered under a common parent.pom file, enabling unified dependency resolution, plugin configuration, and version control across the entire backend ecosystem.

Each microservice relies on *application.yml* files for defining configuration settings such as database connections, server ports, service URLs, and Kafka topics. These *YAML* files are a standard in Spring Boot for externalized configuration. To adapt the system across different environments (e.g. local development, testing, production), dynamic values from *application.yml* were externalized into a *.env* file, allowing environment-specific overrides without changing the codebase. These environment variables are then loaded at runtime using Spring profiles, making the system highly portable and environment-aware. However, instead of using a more complex centralized configuration server like Spring Cloud Config [20], this project chose a simpler *.env* file-based approach.

The choice to use *.env* files was intentional: during the early stages of development, configuration values were frequently changing, and there was no requirement for dynamic runtime reloading of properties. Using *.env* files allowed for quick local changes without the overhead of maintaining a separate configuration server. These files are loaded into the environment at startup and mapped into the application via Spring profiles, which makes it easy to switch between environments with minimal effort.

Eureka Server was set up as a central service registry, allowing services to register and discover each other dynamically without hardcoded addresses. This further decouples service communication and enables horizontal scalability.

On the frontend, built with Angular, configuration is handled using the *environment.ts* file. This file contains settings like the backend API base URL. When the project is built with the production parameter, Angular automatically replaces this file with *environment.prod.ts*, ensuring the correct backend endpoints are used in production without modifying the codebase.

For testing, a dedicated Maven-based repository was created. This includes configuration for integration and end-to-end testing for each microservice. At the core of this test suite is an abstract utility class, which abstracts common setup logic. For example, instead of writing repetitive code to register a new user, developers can call *createUser()* function, which handles the full initialization and returns a ready-to-use user context. This significantly speeds up test development and improves maintainability.

3.3 Core Microservice Development

After laying the foundation of the system and setting up the environment, development proceeded with the implementation of the core business services. Each microservice was designed to encapsulate a distinct domain, following the principles of single responsibility and bounded context from domain-driven design.

3.3.1 User Service

The User Service serves as the central component for handling security-related functionality within the system. Its primary responsibilities include user authentication, authorization, and secure token management. The service supports two authentication approaches: traditional email/password login and OAuth 2.0 third-party login.

Traditional Authentication

Users can register with an email address and password. To prevent unauthorized account creation and verify ownership, the system requires users to confirm their email address via a verification link. For email delivery, the service integrates with Google SMTP.

OAuth Authentication

Alternatively, users can sign in using third-party identity providers via OAuth 2.0. Supported providers include Google, Facebook, Discord, and GitHub. Upon successful authorization, the system retrieves the user's identity from the provider. For future functionality, the system uses usernames, so after registration, the user will need to enter a username. The system itself can also suggest a username to the user, for example, if a user registers via Google and his email is *johndoe@gmail.com*, then the user will be offered the username *johndoe*, the user only needs to click the confirmation button.

Referral System

The user service also includes a referral system. A user can register using a referral link provided by another user. If the link is valid, the referring user becomes linked to the new registrant, allowing for future reward logic and tracking mechanisms.

Asynchronous Notification of Successful Registrations

To support event-driven system architecture and decoupled service communication, the User Service notifies other microservices about successful user registrations using Kafka. When a user completes the registration process either through traditional means or via OAuth the service publishes a *UserRegistered* event to a designated Kafka topic.

Kafka is configured to ensure that registration events are delivered with "at-least-once" semantics. This means that even in the case of temporary network failures or service restarts, the event will eventually reach all subscribed consumers. Although this introduces the possibility of duplicate events, subscribing services are designed to handle idempotent processing by checking unique user identifiers.

3.3.2 API Gateway

The API Gateway acts as a single entry point for all client requests, directing them to the appropriate backend microservices. It not only routes requests efficiently but also embeds essential security and resiliency mechanisms at the edge of the system. It acts as the system's central guard, performing identity checks, enforcing roles, and isolating routes per service, all while ensuring that external access adheres to CORS policies. This separation of concerns simplifies the internal microservices while improving scalability, maintainability, and overall system security.

Routing and Service Segregation

Built using Spring Cloud Gateway, the gateway defines multiple service-specific routes using *RouteLocator*. Each route is mapped to a specific microservice path and secured using a custom filter chain. The code below shows the route configuration for the user service.

```
.route("user-service", r -> r.path("/api/*/users/**")
        .filters(f -> f.filter(userServiceAuthFilter)
                .circuitBreaker(config -> config
                        .setName("user-service")
                        .setFallbackUri("forward:/fallback")))
        .uri("lb://user-service")
```

Each route includes a circuit breaker configuration to ensure fault tolerance. In case of service failure or timeout, the request is redirected to a fallback URI to prevent system-wide degradation. The *lb://* prefix in the URI indicates that Spring Cloud Gateway uses client-side load balancing and service discovery to dynamically route requests to available instances of the specified microservice.

The circuit breaker pattern is a resilience mechanism designed to detect failures and prevent repeated attempts to access a service that is likely to fail. It operates as a finite state machine with three primary states: Closed, Open, and Half-Open. [21]

- In the Closed state, the circuit breaker allows all requests to pass through to the target service while monitoring for failures. If the failure rate exceeds a predefined threshold within a specific time window, the circuit breaker transitions to the Open state.

- In the Open state, the circuit breaker short-circuits all calls to the target service and instead directs requests to a predefined fallback mechanism. This helps to immediately relieve pressure on the failing service and avoids compounding system failures.
- After a configurable period of time, the circuit breaker enters the Half-Open state, during which it allows a limited number of test requests to pass through. If these requests are successful, the service is deemed healthy and the circuit breaker transitions back to the Closed state. If the test requests fail, the circuit returns to the Open state.

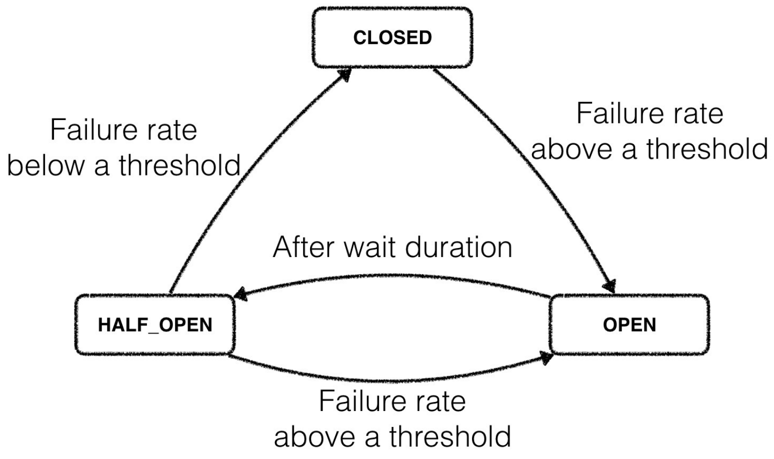
CLOSED
Failure rate below a threshold
Failure rate above a threshold
After wait duration
HALF_OPEN
OPEN
Failure rate above a threshold

Figure 2. Circuit Breaker mechanism.

This pattern improves system stability and user experience by preventing cascading failures, reducing latency during outages, and allowing degraded service operation through fallbacks. In this platform, the circuit breaker is configured using Resilience4j, with fallback URIs serving static or cached responses where appropriate.

Authentication and Authorization Filters

Security enforcement is handled via custom Gateway filters. Before redirecting a request to another service, these actions occur:

1. The requested endpoint is checked whether it requires authorization or not, if not, the request is simply forwarded without any changes.

2. If a token is missing, malformed, or expired, the request is rejected with 401 Unauthorized status.
3. If the path targets an administrative endpoint and the token lacks admin privileges, the request is rejected with 403 Forbidden status.
4. If valid, the filter extracts the user ID and roles from the token and attaches them as headers to the forwarded request.

This approach ensures that authorization is enforced at the edge, reducing the complexity of security logic in internal services and preventing unauthorized access early in the request lifecycle.

3.3.3 Clicker Service

The Clicker Service is a central component of the virtual currency system within the platform, responsible for enabling users to earn *VCoins* - a soft in-app currency that can later be converted into *VDollars*, a more valuable currency with broader usage across the system. The service gamifies user engagement by rewarding interaction, upgrades, task completions, and consistent usage streaks.

Earning VCoins through Interaction

At its core, the Clicker Service allows users to tap a virtual button, representing a simple click action. Each tap contributes a predefined amount of *VCoins* to the user's balance, with anti-abuse logic to prevent manipulation through fake timestamps or excessive tapping beyond recoverable limits.

Upgrades and Passive Income

Users can purchase upgrades using their earned *VCoins*. These upgrades improve passive earnings and unlock higher levels of engagement. Upgrade availability and progression are constrained by conditions such as previous upgrade levels, ensuring a structured progression system. The system calculates the cost and effects of upgrades and ensures users have enough balance before processing purchases.

Task Completion and Rewards

To further encourage engagement, the service provides tasks that users can complete, such as subscribing to a channel or watching a video. Task completion earns *VCoins* and contributes to net worth, enabling access to higher upgrade tiers and enhancing progression.

Daily Streak System

Consistency is rewarded by the daily streak system. Users receive bonuses for daily logins and activity. This system increases user retention and ensures regular passive accumulation of *VCoins*.

Currency Conversion

Users can convert their *VCoins* to *VDollars* at a fixed exchange rate (100,000 *VCoins* = 1 *VDollar*). Conversely, users can convert *VDollars* back to *VCoins* but with a 10% fee (1 VDollar = 90,000 VCoins).

Reliability and Financial Safety in Currency Conversion

In distributed systems where financial transactions are involved, it is critically important to guarantee consistency, fault tolerance, and reliability. It is fundamentally unsafe to rely on simple synchronous REST requests between services to process currency conversion. At any point during a distributed operation such as deducting a balance, requesting a conversion, or updating another service, errors can occur due to network instability, hardware failure, or temporary unavailability of downstream services. In real-world environments, external factors such as severe weather conditions or infrastructure outages can cause service interruptions. If a REST call fails midway through a transaction (e.g. the balance has been deducted but the currency conversion did not occur), it becomes nearly impossible to reconcile the system without risking data loss or financial inconsistency.

To avoid such issues, this system adopts the Outbox Pattern [22] as its core communication and transaction coordination mechanism. The Outbox Pattern ensures reliable event-driven communication between services without compromising transactional integrity.

While the Outbox Pattern effectively ensures data consistency and transactional integrity in distributed systems, its traditional form introduces a delay in user feedback. Because operations are processed asynchronously the user may receive no immediate confirmation of success, even though their request was accepted. This delay can negatively impact user experience in financial contexts, where users expect quick, real-time feedback on balance updates or currency conversions.

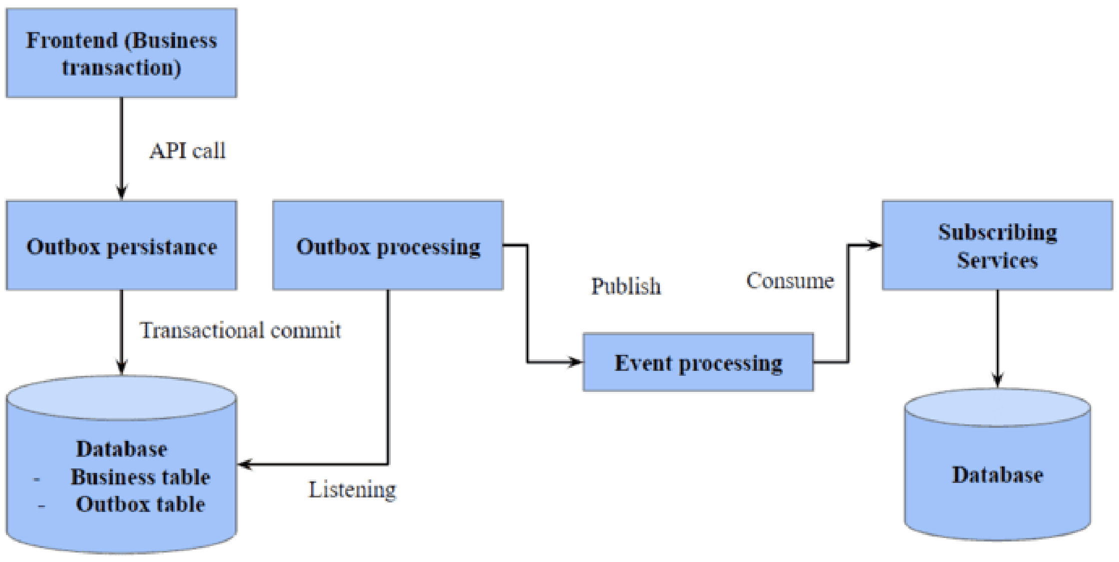

Figure 3. Outbox Pattern.

To address this, the author introduced a hybrid mechanism that maintains the safety guarantees of the Outbox Pattern while enabling near-instant feedback through tightly coordinated steps between the Clicker and Wallet services, combining synchronous execution with asynchronous confirmation.

In this implementation, the Clicker Service initiates the currency conversion by first making a synchronous REST call to the Wallet Service via a FeignClient. This request creates a new conversion event in the *outbox_events* table of the Wallet Service, marked with the status *PENDING_CONFIRMATION*, and returns an *eventID*. Only after this confirmation is received does the Clicker Service proceed to deduct the user's *VCoin* balance and store a matching local transaction record within a single database transaction. Upon successfully committing a transaction, a confirmation event is published to a Kafka topic. This signals the Wallet Service that it can immediately proceed with the conversion, minimizing latency.

Furthermore, to ensure resilience even in the event of a Kafka publishing failure, the Wallet Service includes a background reconciliation process. This process periodically scans for *PENDING_CONFIRMATION* events that have not been processed yet and contacts the original applicant service to confirm the status. If the applicant reports that the conversion was successfully initiated, the Wallet Service proceeds with finalizing the operation. This additional safeguard guarantees that no events are lost and all intended conversions are eventually completed, regardless of messaging system interruptions.

In conclusion, by building on the Outbox Pattern and extending it with performance-optimized feedback mechanisms, the system ensures that no money is lost or duplicated, even in the face of service failures, partial outages, or processing errors.

FeignClient Integration

To simplify inter-service communication, especially with the Wallet Service, the system uses Spring Cloud's declarative FeignClient. This client abstracts the underlying HTTP calls and integrates seamlessly with Spring Boot, allowing REST endpoints to be called as if they were local method calls.

However, the default HTTP client used by Feign may not always provide optimal performance in high-throughput environments. To improve connection management, timeout handling, and overall responsiveness, FeignClient's configuration has been enhanced to use OkHttp as the underlying HTTP client.

OkHttp offers features such as connection pooling and improved resource utilization, providing faster and more consistent response times and better reuse of connections between services. Additionally, it integrates well with resilience libraries such as Resilience4j, allowing circuit breakers to be triggered in the event of service delays or failures, thereby protecting the system from cascading errors.

3.3.4 Clicker Data Service

The Clicker Data Service was introduced as a dedicated microservice to separate and offload the responsibility of collecting and analyzing interaction data from the main Clicker Service. This architectural decision was driven by the author's expectation that Clicker Service would face the highest user load in the system. Integrating analytics functionality directly into Clicker Service would compromise its responsiveness and stability under such load. To improve scalability and maintain a clear separation of concerns, the author decided to develop a standalone service focused solely on data ingestion and analytics processing.

The Clicker Data Service is designed to receive click activity data from Clicker Service and store aggregated metrics for later analysis. The system uses an event-driven architecture where user click activities are published to a Kafka topic. These events are then consumed asynchronously by Clicker Data Service, which aggregates the data using Redis, an in-memory key-value store known for its high performance.

Real-Time Click Aggregation

After receiving a batch of click events via Kafka, the service processes them using a consumer component. Each event contains an accountId and a click count. These are written to a Redis Hash structure, where the total click count for each user is incremented accordingly. This low-latency, in-memory operation ensures that even under high loads, the system can efficiently accept click data without impacting application performance.

Periodic Synchronization with Persistent Storage

To ensure data durability and simplify analytics, the service includes a scheduled task that synchronizes aggregated data from Redis to a PostgreSQL database every 15 minutes. The synchronization process begins by renaming the active Redis key (for example, from *clicks* to *clicks_temp_YYYY-MM-DDTHH:MM*) before processing. This renaming serves two main purposes: it isolates current records from the synchronization process, thereby avoiding potential data leakage, and it enables retry mechanisms in the event of a failure to save data.

Once the Redis data has been safely isolated for each user, the system updates their total daily click count in PostgreSQL, using an upsert strategy to merge new data with existing records. At the same time, the service records the total click count accumulated across all users over a 15-minute interval in a separate table used to track system-wide activity.

Fault Tolerance and Distributed Execution

In case of application failure or system restart, the Clicker Data Service includes logic to automatically reprocess incomplete sync operations. It detects any Redis keys with a *_temp_* suffix (indicating unfinished migration tasks) and re-executes them upon startup.

To support horizontal scalability and prevent race conditions during synchronization, the service uses distributed locking with Redisson. Redisson is a Redis-based Java library that provides high-level distributed coordination tools, such as locks. It ensures that even when multiple instances of the Clicker Data Service are running, only one instance performs synchronization at a time, avoiding duplicate writes and ensuring consistency.

Design Evolution and Initial Failures

Initially, the author implemented a simple design for recording click activity: batches of click events received from Kafka were immediately written to a relational database. Although conceptually simple, this approach quickly proved suboptimal during performance testing. Each batch triggered a corresponding set of insert or update operations on the database, which resulted in significant I/O overhead and decreased system responsiveness.

More importantly, this implementation suffered from data loss. Under high-frequency workloads, the database could not cope with the volume of incoming events. This resulted in backpressure, missed writes, and incomplete persistence. In one test involving 100,000 simulated click events, only about 95% were successfully written. The rest of the data was lost due to transaction contention and resource saturation.

To address these shortcomings, the author restructured the service to aggregate click data in-memory using Redis. Using Redis Hash structures and atomic *HINCRBY* operations, the system achieved high-speed accumulation of click data without relying on the real-time database. This design significantly improved performance and eliminated the database as a bottleneck. However, despite this improvement, an additional problem arose: data loss during the synchronization process. While Redis records were being committed to the database, new events could arrive and overwrite the same Redis keys being processed, resulting in data loss. To address this issue, the system was modified to temporarily rename the Redis key before each synchronization cycle. This effectively isolates active records from the synchronization operation, ensures atomicity, and allows failed synchronizations to be retried without loss, thereby maintaining data integrity even under high load.

3.3.5 Wallet Service

The Wallet Service is the core financial component of the system, used primarily by other internal services to facilitate intra-service monetary transactions. Each financial transaction in the system is performed in two separate steps:

1. **Intention Stage:** the system expresses an intent to perform a transaction (e.g. deposit, withdrawal, or currency conversion), which results in the creation of a pending OutboxEvent.
2. **Confirmation Stage:** the transaction is confirmed or cancelled based on the

responses of the involved services.

This two-step approach ensures consistency and fault tolerance. If a transaction remains unconfirmed due to a service outage or network issues, the Wallet Service periodically queries the status of all pending events. Based on the results, it completes each transaction, marking it as *COMPLETED* or *CANCELLED*.

To prevent problems with concurrent changes, the Wallet entity uses Optimistic Locking. Since multiple threads may attempt to change the same wallet (e.g. due to heavy traffic or overlapping operations), there is a risk of data inconsistency or lost updates where changes from one operation inadvertently overwrite another. Optimistic Locking helps detect these conflicts at the database level using a version field. If a version mismatch occurs during an update, the transaction fails and can be safely retried, preserving data integrity.

In cases where Kafka or dependent services are temporarily unavailable, a backlog of unacknowledged events can accumulate. To efficiently process large volumes of these pending events, the Wallet service splits them into batches, distributing each batch across multiple threads for parallel processing. This scales event finalization throughput and minimizes transaction resolution latency, ensuring that the system remains responsive and consistent even under degraded conditions.

In addition, Wallet Service supports referral-based bonuses. When a new user registers via an invitation, their wallet is initialized and optionally credited with a predefined referral reward from the inviter's configuration.

3.3.6 Odds Service

The Odds Service was designed by the author as a core component to support real-time betting on esports events, with a primary focus on Counter-Strike 2 matches. The original idea was to provide users with the ability to place bets based on dynamically changing odds during live matches, a feature known as live betting.

In traditional betting platforms, especially those targeting esports, odds are commonly sourced from specialized third-party providers. These providers typically collaborate with official tournament organizers to access in-game data up to 30 seconds in advance, giving them an informational edge for accurate odds calculation. However, such services are commercial and often expensive. To avoid this dependency the author decided to simulate the functionality of these providers by building a self-contained odds calculation service.

Architectural Considerations

Before starting development, the author researched the architectural patterns of existing odds providers to ensure the solution would be both extensible and maintainable. As a result, the Odds Service was designed with modularity in mind, enabling the future integration or replacement of data sources with minimal code changes.

It is worth noting that the Odds Service is the only component in the system that does not support horizontal scaling. This design decision is due to the predictable and limited load that the service handles. It processes a fixed number of matches at a time and does not have to handle a large number of simultaneous user requests, unlike user services. Thus, a single instance is sufficient to reliably support the service's operations.

Functional Workflow

The Odds Service operates according to the following sequence:

1. **Receiving In-Game Events:** The service collects real-time game state data.
2. **Odds Calculation:** Based on current game conditions, probabilities are computed.
3. **Odds Distribution:** Updated odds are then propagated to downstream services via Kafka.

To obtain live in-game events, a third-party service was found that provides in-game events for Counter Strike 2 [**counter-strike**] with a 30-second delay. To obtain this information, the service uses Selenium, a browser automation tool commonly used for testing web applications. Selenium allows you to automate interactions with web pages, which allows the service to extract live match data directly from web interfaces where official APIs are not available.

Odds Calculation and Market Management

The system calculates the probability of each team winning, based on team strength and in-game factors. These probabilities are then used to derive odds for different betting markets. A market in this context refers to a specific type of bet, such as "Match Winner", "Map Winner", or "Total Round amount".

For example, if both teams are equally strong, the probability of either team winning is 50%. This gives fair odds of 2.00 for both teams (i.e. if a user bets 1 *VDollar*, if they win, they will receive 2 *VDollars*). However, like all betting services, the system includes a margin, a built-in percentage of profit for the operator. The author implemented a margin of 5%, adjusting the odds to 1.90/1.90.

In cases where the probability of one team winning becomes extremely skewed, the relevant market is automatically suspended, preventing users from placing new bets. If the dynamics of the game change again and the probabilities normalize, the market can be reopened. Once the match is completed or a certain market condition is resolved, the service publishes the final result for each market.

3.3.7 Bet Service

The Bet Service is the core component of the platform, responsible for managing the bet lifecycle, from creation to settlement. It provides users with a real-time betting interface and integrates with other platform components. The service is designed to provide a responsive user experience while maintaining consistency and resiliency in transaction processing.

Real-Time Odds Updates via WebSocket

Since the platform supports real-time betting, it is important to ensure that users receive the latest odds and match information with minimal latency. To achieve this, the system uses WebSocket, a communication protocol that enables low-latency bidirectional communication between the server and the client, making it well suited for real-time betting scenarios.

When the Odds Service issues an update via Kafka, only one Bet Service instance typically receives the event. However, since user sessions are distributed across multiple instances (e.g. 40% on instance 1 and 60% on instance 2), broadcasting the update to all connected clients is a challenge. While one solution was to set up Kafka consumers on all instances, this would require synchronizing Redis cache updates between them, which is used to reduce database load and provide faster responses to users.

Instead, the following hybrid approach was implemented:

1. The instance that receives the Kafka message updates the match data in Redis.
2. A DTO is prepared containing the updated market information.
3. This update is then published via Redis Pub/Sub.
4. All instances subscribed to the corresponding Redis channel receive the message and forward the update to their connected WebSocket clients.

This architecture allows for scalable real-time broadcasting without the overhead of multi-instance Kafka synchronization or race conditions in Redis updates.

Bet Placement and Processing

Users place bets by sending a request to the */bet/place* endpoint. When the request is received, the system introduces an artificial delay of 3 seconds. This delay simulates a real-world scenario where the market may close or the odds may change just before the bet is confirmed. The core logic is executed asynchronously using a dedicated thread pool.

During bet processing first, the system checks if the market is open and the result has not been calculated yet. It checks if the odds provided by the user match the current odds, if they differ and the user has not accepted all the odds changes, the request is rejected. If the bet is valid, the corresponding records are saved in the database and a withdrawal request is created in the Wallet Service, to ensure reliability, the same approach is used as with currency conversion. Finally, the system returns a successful response to the user.

Once the market outcome is known, the system automatically settles the results of all related bets. However, in cases where bets are not resolved due to service interruptions or data inconsistencies, the scheduler periodically scans for unresolved bets and attempts to resolve them.

Administrative Functionality

To manage events in the system, administrators manually add tournaments, matches, and teams through a dedicated admin panel. To improve the user experience, visual identifiers such as logos and images for tournaments and teams can be uploaded. These images are stored in Amazon S3, a scalable, highly available object storage that is typically used to store and retrieve any amount of data at any time.

For local development, the service uses LocalStack, a full-featured mockup of AWS cloud services running in Docker [23]. By substituting AWS credentials and endpoints into environment variables, developers can work with S3 locally without changing the application logic, thereby maintaining code portability between development and production environments.

Additionally, the admin panel includes manual intervention tools, allowing admins to change or adjust bets and correct incorrect markets if necessary. This ensures that the platform maintains data integrity and trust among users, even in exceptional circumstances.

3.3.8 Notification Service

The Notification Service is responsible for delivering real-time messages to individual users within the platform. It is implemented using WebSocket technology. Unlike the Bet Service, where WebSocket channels are public and accessible to all users interested in event updates, the Notification Service enforces strict authentication and user-specific session handling. This ensures that notifications are delivered solely to their intended recipients.

To facilitate inter-service communication and decouple the Notification Service from other components, the Redis Pub/Sub messaging system is used. This design allows any microservice in the system to publish a notification message to a specific Redis channel, and the Notification Service will handle delivery to the correct user.

For example, the Wallet Service can notify a user about balance update or the Bet Service can inform the user about the result of a bet. These services do not need to know whether the user is currently online or which instance of the Notification Service is responsible, they simply publish a message to a Redis channel and the Notification Service handles the rest.

Since a user's WebSocket session can be active on any Notification Service instance, each instance must independently determine whether it maintains an active session for the target user. Upon receiving a notification event from Redis, an instance checks its local session registry. If an active session for the user exists, the notification is immediately delivered over WebSocket. If no session exists, the instance discards the message. This mechanism is replicated across all instances, ensuring that the message reaches the user if they are currently connected. If the user has multiple application tabs open in the browser, the notification will arrive in all of them, since the system stores all active user sessions.

As a result, users receive a seamless, real-time experience, increasing engagement and overall responsiveness of the platform.

3.4 Observability and Reliability

In a distributed microservices architecture, observability plays a critical role in ensuring system reliability, performance monitoring, and proactive incident response. To meet these goals, the system integrates a comprehensive observability stack composed of Prometheus for metrics collection, Grafana for visualization and alerting, and Loki for centralized log aggregation. Together, these tools offer a unified and scalable solution for monitoring the platform's health, diagnosing issues, and maintaining operational excellence.

3.4.1 Prometheus: Metrics Collection

Prometheus is the primary tool used for collecting system and application metrics across all microservices. Each Spring Boot service exposes metrics via the */actuator/prometheus* endpoint using Micrometer. These metrics include JVM memory usage, request throughput, HTTP response codes, database query performance, etc.

The system's Prometheus instance is configured with Eureka-based service discovery using *eureka_sd_configs*. This allows Prometheus to automatically detect and scrape metrics from all registered services, without the need to manually update target IP addresses or ports, ensuring seamless scaling and dynamic discovery in Dockerized environments.

Prometheus also scrapes metrics from PostgreSQL via a dedicated *postgres-exporter* Docker container. This provides deep visibility into database-level performance indicators such as query execution time, cache hit ratios, and connection stats.

3.4.2 Grafana: Monitoring

Grafana provides a powerful visualization layer on top of Prometheus metrics. It is configured to automatically provision dashboards from mounted configuration files. Each dashboard offers clear insights into the health and performance of specific services, such as CPU utilization, garbage collection activity, response times, and custom configured metrics.

Grafana supports role-based access control and dashboard versioning, ensuring that the monitoring environment remains both secure and maintainable. Additionally, by enabling Prometheus as the default data source and Redis/Loki as secondary sources, Grafana provides a unified interface for visualizing both system metrics and application logs.

Crucially, Grafana also facilitates real-time alerting. System administrators can define alert thresholds based on Prometheus metrics such as high error rates or memory pressure and configure alert routing to external notification systems like email. This ensures rapid detection and escalation of operational issues, minimizing downtime and improving response time.

3.4.3 Loki: Centralized Log Aggregation

While Prometheus processes structured metrics, Loki complements it by offering scalable, centralized log aggregation. Each application logs messages using Logback, a Java framework for logging, configured with *loki-logback-appender*, which streams structured log data to a Loki instance. Logs are indexed with labels such as service name, instance, and log level, allowing for efficient querying.

Initially, the plan was to integrate the ELK stack for log management. However, since Grafana was already set up for metrics visualization, adopting Loki was a more pragmatic choice. Loki's native integration with Grafana enabled an effective logging solution without introducing additional infrastructure complexity.

This setup allows engineers to monitor application behavior, debug errors, and perform root cause analysis by correlating logs with metrics displayed in Grafana. For example, a spike in request latency visible in a dashboard can be immediately investigated in context by drilling down into the corresponding log entries in Grafana itself - thanks to its native integration with Loki.

3.5 Frontend

The frontend of the application is built using Angular and provides full functionality for both end users and administrators. Every feature available in the system has been implemented with a corresponding interface, enabling seamless interaction with the backend. The application emphasizes responsiveness, real-time updates, and a user-friendly experience across all devices.

3.5.1 User Interface

The user interface supports authentication flows, including email-verified registration, password-based login, and OAuth-based login through providers such as Google, Facebook, GitHub, and Discord. It also features an automatic token refresh system to maintain active sessions without manual intervention.

The core of the user experience revolves around earning and managing virtual currency. Users can interact with a dynamic clicker component to generate virtual coins, monitor their balance in real-time, and purchase upgrades that increase passive income. Tasks such as subscribing to channels or watching videos are also presented through the user interface, offering additional ways to earn rewards.

A dedicated wallet interface allows users to convert between two in-game currencies, initiate deposits, or withdraw. Conversion logic, including fee rules, is handled transparently in the interface.

The betting section offers a clean layout for browsing current and upcoming tournaments, viewing dynamic odds that are updated via WebSocket, and placing bets. Betting results and balance changes are communicated to the user via real-time notifications, ensuring that he is always aware of important events.

3.5.2 Admin Interface

The admin interface is designed to provide comprehensive oversight and management capabilities across all application services. It delivers detailed statistics, user-specific data, and tools to monitor and manage system operations efficiently.

User Management
Administrators can view key metrics, including the total number of registered users, daily, weekly, and monthly registration trends, and active users for the current day. Additional insights include users registered through OAuth, those invited by others, and accounts currently frozen. A search functionality enables admins to find specific users and access

detailed profiles, including activity and usage data, with options to take actions such as freezing accounts.

Clicker Service Management

Admins can monitor the performance of the clicker service with metrics like active users today, total clicks, clicks per user, streaks completed, total net worth, and upgrades purchased. Charts provide visualizations of trends, such as active users and clicks over varying timeframes (e.g. last week or last month). There are also tools for task management, allowing admins to add or modify tasks, along with user-specific controls to adjust *VCoin* balances or freeze accounts.

Wallet Management

The admin interface offers insights into the overall financial health of the system, including the total wallet balance, transaction volume, and currency conversions for both *VCoins* and *VDollars*. Admins can search for specific wallets to view detailed transaction histories and make adjustments, such as adding *VDollar* balances or setting referral bonuses.

Esports Management

For the esports betting system, administrators can view total bets, bets in specific time frames, total bet amounts, and win/loss distribution. The interface supports administrative tasks such as uploading images, adding tournaments, managing matches and participants, and overseeing markets for specific matches. In case of errors, administrators can adjust market data. User-specific betting data is also available for detailed examination.

3.6 Testing

3.6.1 Unit Testing

Unit testing focuses on verifying the correctness of individual components or methods in isolation from the rest of the system. In this project, unit tests were written for the core logic in services such as User Service, Clicker Service, Bet Service, Wallet Service.

The main goal of unit testing was to ensure the correct behavior of each method under both normal and edge conditions. For example, in Bet Service, unit tests were written to verify the correct behavior when the market is closed, the odds have changed, or the user has insufficient funds.

JUnit 5 was used as the main testing library along with Mockito for mocking dependencies.

Mocking allows tests to isolate the logic of the class under test by simulating the behavior of its collaborators, such as repositories or external services. This ensures that tests remain fast and deterministic, independent of infrastructure such as databases or networking.

3.6.2 Integration Testing

Integration testing verifies that the various components and services in a system interact correctly when combined. Unlike unit tests, which isolate specific methods or classes, integration tests simulate real-world interactions between multiple layers, such as the controller, service, repository, and external dependencies such as Redis, Kafka, and databases.

A dedicated integration testing repository was created for this purpose to clearly separate integration tests from production code and improve the scalability of testing strategies.

Custom test architecture: The base class provides reusable utilities for making HTTP requests, authentication, Redis and PostgreSQL operations, and configuration management. Services are initialized using a structured configuration file, making the tests environment-agnostic and easily configurable.

For example, one of the tests verifies user registration, ensuring that when a new user registers, the system correctly creates their account and generates an email verification token. It checks that the user's email address matches the one in the token, a retry token is issued, only one email is sent, and further retrying is allowed.

It also checks that the user is shown in the database with an inactive status and the corresponding role. It also confirms that the corresponding token record exists in the database with all required fields filled in.

By isolating integration testing in a separate repository and creating a flexible modular environment around it, the system provides high confidence in the behavior across all services.

3.6.3 Performance Testing

To evaluate the scalability and reliability of the application under heavy user traffic, the author conducted performance testing focusing on the most frequently used endpoint, */clicker/tap*. This part of the system handles the core user interaction of the platform and is expected to withstand the highest load during peak activity. The endpoint performs several internal operations per request, including retrieving and updating user data in the database, as well as performing arithmetic calculations related to rewards and click accumulation. These combined read-write and CPU-limited tasks make it a critical benchmark for end-to-end system performance.

Tool selection

For load generation, k6 [24] was chosen a modern, lightweight tool for HTTP API load testing. Its advantages include support for JavaScript scripts and excellent integration with monitoring systems. Compared to alternatives such as Apache JMeter or Gatling, k6 turned out to be more convenient and productive for the tasks at hand.

Testing Strategy

The test involved 3000 virtual users, each sending repeated requests every 2 seconds. This number was chosen because at this level of load the CPU reached 100% utilization, an important indicator for testing the limits of performance.

The system was subjected to peak testing, where the load rapidly increased to peak values, was held for a short time, and then rapidly decreased. This approach allowed to evaluate the system's resilience to sudden load surges and its ability to recover from a sharp drop in traffic.

Key Metrics

The following metrics were monitored during testing:

- Total number of requests
- Peak RPS
- Average and percentile response time (p90, p95)
- Volume of transferred data
- Presence or absence of failures

These metrics provided a holistic view of the system's behavior under load and helped pinpoint where improvements were needed.

Thresholds

Specific thresholds were set to determine the success of the tests:

- Request errors: less than 1% of the total. If exceeded, the test was stopped.
- Response time:
    1. p90 < 200 ms — 90% of requests should be faster than 200 ms.
    2. p95 < 300 ms — 95% of requests should be faster than 300 ms.
    3. p99 < 2000 ms — 99% of requests should not exceed 2 seconds.

These values reflect not only the survivability of the system, but also its ability to provide a fast and predictable response.

Observations and optimizations

Initial test showed that the system can handle up to 1330 RPS with an average latency of 187 ms, but there were failures indicating a lack of resources.

To eliminate the problems, the following optimizations were performed:

1. **Caching with Redis:** integrated as a caching layer to reduce the frequency of database queries. However, application profiling using JProfiler showed that the caching strategy resulted in unexpected overhead. Instead of reducing the load on the PostgreSQL database, it resulted in increased latency. This was primarily due to the fact that the system now had to wait for both Redis and database connections at the same time, effectively doubling the contention. As a result, the intended performance gains were not achieved — instead, response times worsened. Therefore, it was decided to abandon caching.
2. **Setting up a connection pool:** changing the parameters of the database connection pool gave a significant performance boost. The response time dropped to 54 ms, RPS reached 1500 and the errors disappeared.
3. **Reducing response size:** originally, the */clicker/tap* endpoint returned redundant information, including data that was not used on the client. Removing this data reduced the amount of information transferred from 1.7 GB to 79 MB, further improving performance and reducing network load.

Final results

After all the improvements, the system demonstrated stable operation under 1500 RPS load, with minimal latency and no failures. This confirms its readiness for operation in highly parallel access conditions. More detailed results are presented below in the Table 1.

Table 1. Performance Testing results.

Changes Applied	**Total Requests**	**RPS peak**	**Average Response Time**	**Data Transferred**	**Errors (Yes/No)**
Initial	189352	1330	187 ms	1.7 GB	Yes
Caching with Redis	59475	330	5000 ms	515 MB	Yes
Connection pool tuning	195025	1500	54 ms	1.7 GB	No
Reduction of received data	195390	1500	42 ms	79 MB	No

3.6.4 User Testing and Feedback

To test the functionality, usability and stability of the developed system, a round of user testing was conducted. The main goal was to identify any functional errors, evaluate the user experience and collect qualitative feedback on the overall design and performance of the application.

Test Group Composition

A diverse group of approximately 10–15 (depending on the day) users participated in the testing phase. The group was deliberately chosen to provide a wide range of perspectives:

- Some participants had IT or software development experience,
- Others were non-technical users,
- Several users were familiar with betting platforms,
- While others had no previous experience with betting or gambling systems.

This composition ensured that the feedback reflected both technical validation and real-world usability at varying levels of digital literacy and domain familiarity.

Hosting Setup and Deployment for Testing

To facilitate real-time testing and enable quick deployment of fixes, the application was hosted directly on a personal computer. The deployment involved several configuration steps to ensure reliability, security, and public accessibility:

- **Nginx Configuration:** Nginx [25], a high-performance HTTP server and reverse proxy, was configured to route incoming HTTPS traffic to the appropriate services running on a personal machine. Its primary role in this setup was to handle domain-based routing and act as a TLS termination point, enabling secure connections.
- **Domain Purchase:** A domain name was acquired to allow users to access the application via a human-readable address rather than an IP address. Domains are essential for improving accessibility and are often required by third-party services like OAuth providers.
- **OAuth Reconfiguration:** OAuth provider settings were updated to reflect the new domain name, as these platforms typically require a predefined list of redirect URIs for authentication to work securely.
- **TLS Configuration:** TLS was set up to secure data transmission between the client and server. This was achieved by generating and installing a valid SSL certificate, ensuring encrypted and trusted communication.
- **Public Network Exposure:** The local hosting environment was exposed to the public internet through network configuration and port forwarding, making the application accessible to testers externally.

Once deployment was complete, each user was sent a unique referral link. Upon registration through this link, users received 100 VDollars, providing them with immediate access to the full range of application features, including betting functionality and wallet interactions.

Testing Process and Outcomes

The testing period lasted for approximately three days, coinciding with the IEM Melbourne 2025 Counter-Strike tournament which was held from 20-04-2025 to 27-04-2025, to simulate real-world usage during the event. Feedback was collected through informal interviews and messaging throughout the testing period.

- **Day 1:** A few non-critical issues were found, mostly related to UI layout inconsistencies. Additionally, a backend issue was identified where newly added functionality had not been fully tested - for example, match score data inconsistencies due to Redis not updating properly, resulting in outdated information being displayed.
- **Day 2-3:** After fixing the identified issues, the application remained stable. No further errors were observed and the system successfully processed and displayed

user interactions.

Feedback and Conclusions

The user testing phase proved to be a valuable component of the system validation process. Conducted over several days coinciding with the IEM Melbourne 2025 Counter-Strike tournament, the testing environment successfully simulated real-world usage patterns. The period also created a time-sensitive context in which users interacted with the application during live matches, revealing how the system behaved under realistic conditions.

User feedback highlighted several key strengths of the platform. In particular, participants, both technical and non-technical, expressed particular enthusiasm for the live betting experience, which they found immersive and engaging.

From an architectural perspective, the testing period confirmed that the system should scale horizontally based on demand. For example, the betting service, which is heavily used during match windows, becomes inactive after the events have finished.

3.7 Infrastructure and Deployment

After completing the main development phase and implementing all major features, the system was prepared for deployment to a cloud environment. The main goal at this stage was to validate whether the application is suitable for scalable and reliable operation in the cloud.

The deployment was targeted at Amazon Web Services using Elastic Kubernetes Service [26], a managed Kubernetes platform provided by AWS. EKS simplifies the process of running Kubernetes [27] on AWS by automating infrastructure provisioning, cluster management, and scalability. It offers the benefits of Kubernetes such as container orchestration and resource optimization, while reducing the operational burden of manually managing clusters.

One of the main benefits of using Kubernetes is the ability to deploy updates without downtime. When a new version of a service is available (for example, an updated Docker image), updating a deployment is as simple as changing the image tag in the YAML configuration. After configuration is applied, Kubernetes gradually replaces old pods with new ones, ensuring that a minimum number of pods remain available at all time and if something goes wrong during the update, Kubernetes can automatically roll back to the previous stable version.

Since the author had no previous experience with Kubernetes, the deployment process began locally. This allowed debugging and validation of configuration files and container images in a more controlled environment. After a successful local deployment, an AWS account was set up, which was used to deploy the application to the EKS cluster.

The system entry point was configured using Ingress [28], a Kubernetes component that manages external access to services in the cluster. Ingress provides HTTP and HTTPS routes from outside the cluster to internal services and provides hostname or path-based routing. To integrate deployment with a custom domain name, the DNS settings were updated for the previously purchased domain to point to the AWS-provided load balancer.

Separate subdomains were configured for different parts of the application, including the admin panel and the Grafana dashboard. This ensured a clear separation of concerns and improved the maintainability of the web interface. The deployment tasks were performed through the AWS web console and via the command line using tools such as *kubectl*, *aws*, and *eksctl*. All Kubernetes components were defined using YAML configuration files, allowing for consistent configuration of the infrastructure under version control.

An example configuration file for the betting service included definitions for:

- **Deployment:** defines how the betting service should be run, including the Docker image, environment variables, and startup conditions.
- **Service:** exposing the application internally within the Kubernetes cluster.
- **Horizontal Pod Autoscaler (HPA):** enables automatic scaling of the application based on resource usage. For example, HPA monitors the CPU usage of the betting service and if the threshold goes over 50%, HPA adds more replicas. When the load decreases, it reduces their number.

As a result of these efforts, the entire system was successfully deployed and made fully operational on AWS, confirming that the architecture, containerization, and orchestration strategies are compatible with modern cloud environments. The deployment validated all core services, frontend interfaces, and monitoring components in a real-world cloud infrastructure, completing the transition from local development to scalable production readiness.

4 Results

4.1 Functional Completeness

The development process resulted in a fully functional system consisting of eight Spring Boot-based microservices and two Angular applications, one for end users and one for administrators. Each backend service was interfaced with a frontend interface, ensuring availability, usability, and integration of all functions.

The platform supports key operations such as secure registration including OAuth, user-facing gaming, virtual currency management with conversion logic, live esports betting, real-time notifications, and comprehensive administrative oversight. All components work together as a holistic and modular architecture.

4.2 Testing and Reliability

Testing was an integral part of the development, covering both functional correctness and system performance.

- **Unit Testing:** Core business logic in each service was tested using automated unit tests to verify correctness in isolation.
- **Integration Testing:** Interactions between services, such as communication over Kafka or shared database operations, were verified through integration tests.
- **Performance Testing:** k6 was used to simulate high user load on critical endpoints. This helped identify and resolve bottlenecks, and confirm system behavior under peak stress.
- **User Testing:** During the IEM Melbourne 2025 tournament, 10–15 users tested the platform in a live environment with full functionality. Minor issues were quickly resolved, and the system remained stable. Feedback confirmed strong usability and real-world readiness, especially for the live betting experience.

4.3 System Performance

Performance testing confirmed the platform's ability to support high concurrency. After targeted optimizations, the system achieved a 15% increase in throughput, an 80% reduction in average response time, and zero errors during load tests, all while remaining within established performance thresholds.

These improvements translate directly into business value. Fast and reliable response times ensure a smooth user experience during peak activity, which is critical to user retention. Furthermore, by identifying and eliminating performance bottlenecks, the platform reduces unnecessary resource consumption, contributing to operational cost efficiencies. Finally, maintaining consistently low latency gives the platform a competitive advantage over slower or less stable alternatives, improving its ability to attract and retain users in a performance-sensitive market.

4.4 Scalability and Cloud Deployment

The platform is deployed on Amazon Elastic Kubernetes Service, using a managed Kubernetes environment for easy orchestration and scaling of microservices. Horizontal Pod Autoscaling automatically adjusts the number of service instances based on real-time resource usage, ensuring efficient handling of traffic spikes during live events such as esport tournaments.

This setup allows the system to scale up during peak loads and scale down when demand decreases, optimizing resource utilization and costs. Rolling Kubernetes updates and health checks ensure minimal downtime during deployments, maintaining service availability.

4.5 Fault Tolerance and Resilience

The system's microservices architecture provides fault isolation, preventing failures from propagating across services. Retry mechanisms and circuit breakers handle transient errors, and Kafka provides reliable asynchronous communication. Kubernetes health checks provide automatic recovery by restarting or rescheduling failed services. Critical data, such as monetary transactions, is protected from loss by applying fault-tolerant design patterns.

4.6 Extensibility and Maintainability

The system was designed with extensibility and maintainability in mind, using a modular microservices architecture that allows for easy addition or modification of features without impacting the entire platform. This approach promotes flexibility, allowing new features or updates to be integrated with minimal disruption. Multiple types of testing further enhance maintainability, ensuring that the platform can evolve efficiently while maintaining high quality and stability.

5 Summary

The goal of this thesis was to develop a production-ready, open-source implementation of a scalable cloud-native entertainment platform. Built using microservices and Spring Cloud technologies, the system was designed to support high user loads, deliver real-time features, and ensure fault tolerance, while remaining modular, maintainable, and extensible for future growth.

To achieve this, the system was designed around eight independent microservices, each responsible for a distinct functional area. The frontend, built with Angular, provides separate interfaces for users and administrators, enabling seamless interaction with backend services.

Key architectural strategies included the use of circuit breakers, distributed locking, the Outbox pattern for transaction integrity, and a full observability stack for real-time monitoring and diagnostics. The system was thoroughly validated through unit and integration testing, high-load performance testing, and real-world user testing during a live esports event. The platform was successfully deployed to Amazon Web Services using Elastic Kubernetes Service, confirming its cloud-native scalability and operational resilience.

In summary, this thesis demonstrates how Spring Cloud and microservice architecture can be effectively applied to build a production-grade digital entertainment platform. The work offers a practical reference for building scalable, fault-tolerant, and maintainable systems, and provides a strong foundation for future enhancements or commercial deployment.

References

[1] Anna Rud. *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience*. [Accessed: 19-05-2025]. URL: https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/.

[2] *Spring Cloud*. [Accessed: 19-05-2025]. URL: https://spring.io/projects/spring-cloud.

[3] Jetinder Singh. *The What, Why, and How of a Microservices Architecture*. [Accessed: 19-05-2025]. URL: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9.

[4] Yury Izrailevsky; Stevan Vlaovic; Ruslan Meshenberg. *Completing the Netflix Cloud Migration*. [Accessed: 19-05-2025]. URL: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration.

[5] *Spring Boot*. [Accessed: 19-05-2025]. URL: https://spring.io/projects/spring-boot.

[6] *Sping Cloud Netflix*. [Accessed: 19-05-2025]. URL: https://cloud.spring.io/spring-cloud-netflix/reference/html/.

[7] *Spring Cloud Gateway*. [Accessed: 19-05-2025]. URL: https://spring.io/projects/spring-cloud-gateway.

[8] *Spring Cloud OpenFeign*. [Accessed: 19-05-2025]. URL: https://spring.io/projects/spring-cloud-openfeign.

[9] *Apache Kafka*. [Accessed: 19-05-2025]. URL: https://kafka.apache.org/.

[10] Raphael De Lio. *Understanding Pub/Sub in Redis*. [Accessed: 19-05-2025]. URL: https://medium.com/redis-with-raphael-de-lio/understanding-pub-sub-in-redis-18278440c2a9.

[11] Aissatou Bella. *10 Best practices Mastering Fault Tolerance in Spring Boot Microservices*. [Accessed: 19-05-2025]. URL: https://medium.com/@aissatoub4228/building-resilient-microservices-with-spring-boot-and-spring-cloud-mastering-fault-tolerance-65e138a759b0.

[12] Sujatha R. *Horizontal scaling vs vertical scaling: Choosing your strategy*. [Accessed: 19-05-2025]. URL: https://www.digitalocean.com/resources/articles/horizontal-scaling-vs-vertical-scaling.

[13] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. [Accessed: 19-05-2025]. URL: https://www.postgresql.org/.

[14] Ravidu Perera. *Asynchronous Patterns for Microservice Communication*. [Accessed: 19-05-2025]. URL: https://blog.bitsrc.io/asynchronous-patterns-for-microservice-communication-edd6722a7323.

[15] Hajar Ram. *Monitor a Spring Boot App Using Prometheus*. [Accessed: 19-05-2025]. URL: https://www.baeldung.com/spring-boot-prometheus.

[16] Suraj Mishra. *Logging in Spring Boot With Loki*. [Accessed: 19-05-2025]. URL: https://www.baeldung.com/spring-boot-loki-grafana-logging.

[17] Grafana Labs. *Dashboard anything. Observe everything*. [Accessed: 19-05-2025]. URL: https://grafana.com/grafana/.

[18] IBM. *What is Redis?* [Accessed: 19-05-2025]. URL: https://www.ibm.com/think/topics/redis.

[19] Tom Collins. *What is Maven in Java? (Framework and Uses)*. [Accessed: 19-05-2025]. URL: https://www.browserstack.com/guide/what-is-maven-in-java.

[20] *Spring Cloud Config*. [Accessed: 19-05-2025]. URL: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/.

[21] Bohdan Storozhuk. *Circuit Breaker Implementation in Resilience4j*. [Accessed: 19-05-2025]. URL: https://dzone.com/articles/circuit-breaker-implementation-in-resilience4j.

[22] Safa Bouguezzi. *Outbox Pattern in Microservices*. [Accessed: 19-05-2025]. URL: https://www.baeldung.com/cs/outbox-pattern-microservices.

[23] *What is Docker?* [Accessed: 19-05-2025]. URL: https://docs.docker.com/get-started/docker-overview/.

[24] Anshul Bansal. *How to Execute Load Tests Using the k6 Framework*. [Accessed: 19-05-2025]. URL: https://www.baeldung.com/k6-framework-load-testing.

[25] *Nginx*. [Accessed: 19-05-2025]. URL: https://nginx.org/.

[26] Amazon. *Amazon Elastic Kubernetes Service*. [Accessed: 19-05-2025]. URL: https://aws.amazon.com/eks/.

[27] *Production-Grade Container Orchestration*. [Accessed: 19-05-2025]. URL: https://kubernetes.io/.

[28] Amazon. *Route application and HTTP traffic with Application Load Balancers*. [Accessed: 19-05-2025]. URL: https://docs.aws.amazon.com/eks/latest/userguide/alb-ingress.html.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Vadim Filonov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Developing a Scalable Entertainment Platform with Spring Cloud: A Full Cycle from Design to Implementation”, supervised by Ali Ghasempour
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

04.06.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – GitHub Repositories

The full source code of the developed platform is available on GitHub and is divided into three repositories:

- Backend (microservices and infrastructure): https://github.com/vadof/vplay-backend
- Frontend (user and admin interfaces): https://github.com/vadof/vplay-frontend
- Testing (integration and end-to-end tests): https://github.com/vadof/vplay-test