Project Report for Yushan

Practice Module for Graduate Certificate in Architecting Scalable Systems

Team 18

Members:

1. Ahan Jaiswal
2. Nguyen Phu Truong
3. Yang Shuang
4. Zhang Yan
5. Zhu Yuhui

CONTENTS

1. Introduction

1.1 Background

Yushan - Home of Stories is a gamified web novel reading platform developed as part of the Practice Module for Certificate in Architecting Scalable Systems (SWE5001). The project addresses the fragmentation in the current web novel ecosystem, where readers and writers are scattered across multiple platforms for reading, tracking, discussion, and collaboration.

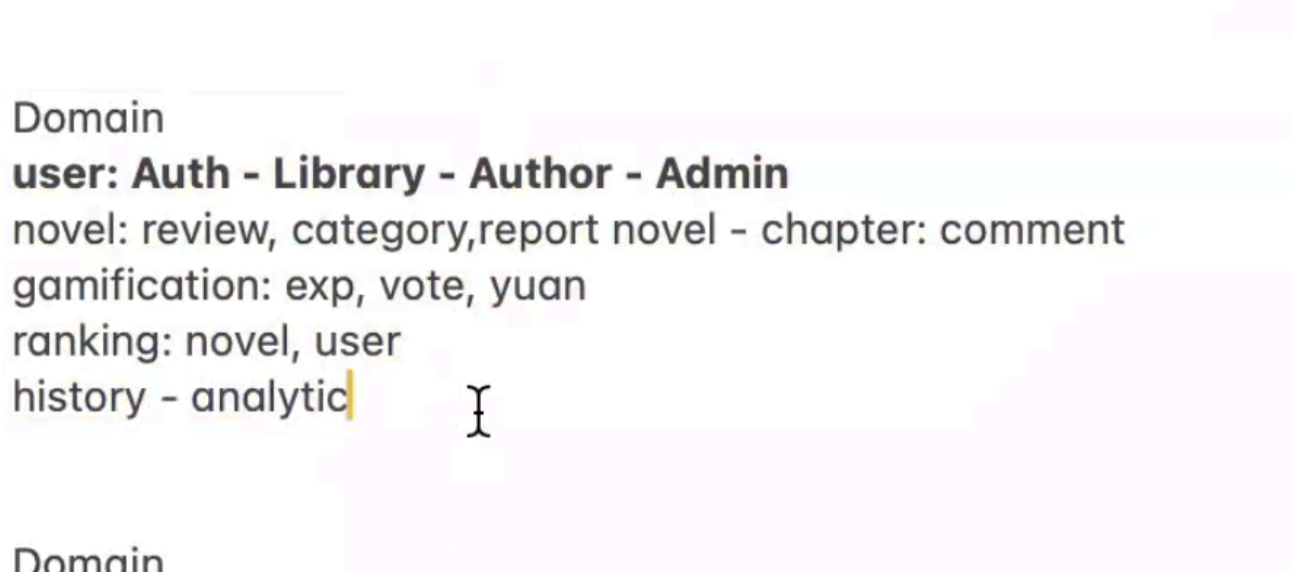
Following the successful completion of the monolithic Yushan – Home of Stories platform under the Designing Modern Software Systems (DMSS) module, the next phase of the project was undertaken as part of this Architecting Scalable Systems (ASS) Graduate Certificate.

While the monolithic application offered a fully functional web-novel reading ecosystem with integrated gamification, author tooling, administrative management, and community features, its growing complexity highlighted several architectural limitations that would restrict further expansion.

To address these limitations, the team embarked on **re-architecting the monolith into a distributed microservices-based system**, leveraging domain-driven design (DDD), event-driven communication patterns, and modern DevOps practices.

This transformation aimed to improve scalability, modifiability, and deployment independence while aligning with real-world cloud-native architecture principles.

Through a series of group discussions, domain modelling workshops, and service-boundary identification exercises, the team defined **five core microservices**, each operating in its own repository and equipped with independent CI/CD pipelines, infrastructure provisioning, and observability tooling.



Rough Division of Domains based on our Initial Backend Refinements

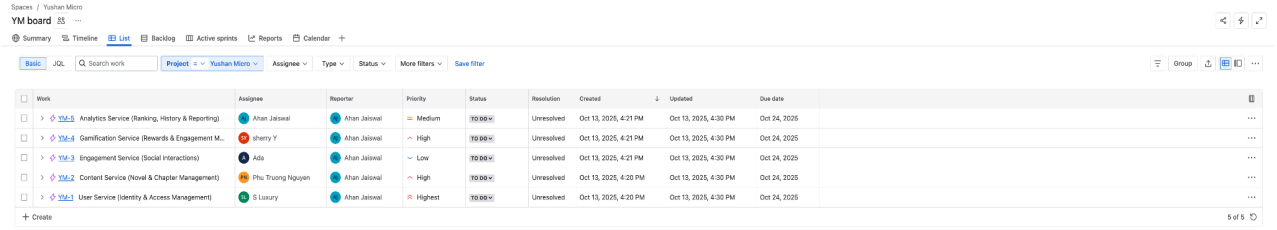
The microservices migration builds upon the solid functional foundation of the monolith while providing a future-proof architecture capable of supporting higher traffic, evolving domain logic, and modular feature growth.

1.2 Business Needs

Transitioning Yushan from a monolithic architecture into a distributed microservices-based cloud native solution addresses several key business and technical drivers:

- **Scalability Demands -** As the platform grows (in readers, authors, novels, chapters, analytics, and community interactions), the monolith's shared resources and tightly coupled logic limit horizontal scaling and isolated workload optimization.
- **Independent Feature Evolution -** Future enhancements—such as advanced analytics, real-time notifications, richer gamification, and author collaboration—require the ability to evolve specific components without impacting the rest of the system.
- **Deployment Flexibility -** The monolith's single deployment unit introduces risk and slows down delivery. Microservices allow independent deployments, faster iteration, and more predictable rollback mechanisms.
- **Domain Separation & Maintainability -** The monolithic codebase mixes concerns across user management, novels, chapters, ranking, gamification, and history/analytics. Decomposing these domains into bounded contexts reduces cognitive load and supports cleaner development workflows.
- **Operational Resilience -** A failure in one module (e.g., ranking or gamification) should not bring down the reading or authentication experience. Microservices improve availability through isolation.

By reconstructing Yushan as a set of self-contained services, the platform aligns with long-term business goals of **growth, modularity, reliability, and sustainable evolution**.



1.3 Stakeholders

Key Business Stakeholders

- **Course Instructors / Professors** – Primary evaluators and mentors.
- **Readers** – Depend on high availability, fast loading, and scalable reading experiences.
- **Authors** – Rely on stable content creation tools, analytics, and engagement features.
- **Administrators** – Require reliable moderation and administrative interfaces.

Technical Stakeholders

- **Ahan Jaiswal** – Product Owner / Scrum Master, Microservices Engineer, CI/CD & DevOps, QA Automation, Infrastructure, Observability & Monitoring
- **Nguyen Phu Truong** – Backend Developer, Microservices Engineer, CI/CD & DevOps, Infrastructure, Observability & Monitoring
- **Yang Shuang** – Frontend Developer, API Integration, QA Automation
- **Zhang Yan** – Backend Engineer, Microservices Implementation
- **Zhu Yuhui** – Frontend Developer, Integrations, QA Automation

All stakeholders collaborate to ensure the microservices redesign meets functional, technical, and all architectural objectives.

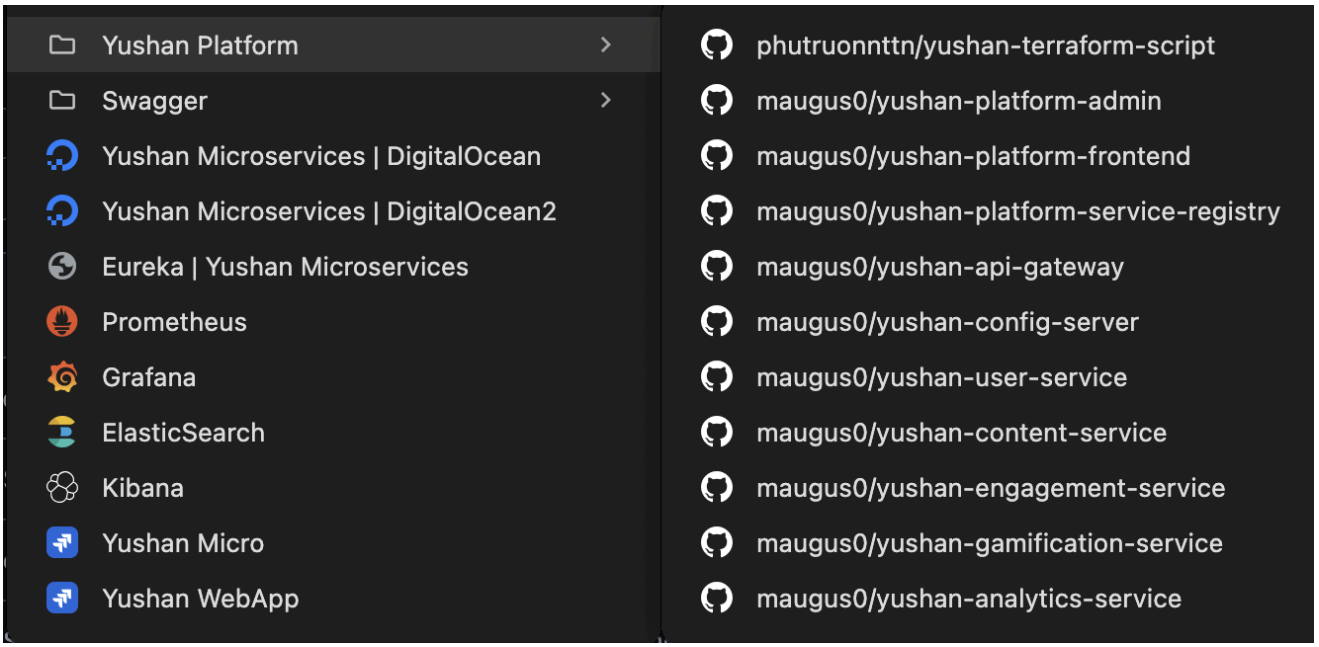
1.4 Project Scope

The scope of the ASS module focuses on **re-architecting the Yushan monolithic application into a scalable, cloud-ready microservices ecosystem**, while preserving core platform functionality.

This includes:

- Identifying bounded contexts and domain-driven service boundaries
- Designing 5 independent microservices based on agreed domain models
- Implementing multi-repo microservices with independent CI/CD
- Setting up asynchronous communication where required
- Implementing API gateways, service discovery, and cross-service contracts
- Ensuring observability (logging, tracing, metrics) across all services
- Deploying services into containerized or cloud environments

While the original monolithic platform remains fully functional, this project focuses on **architecture, design, and implementation of core microservices**, not full replication of every monolithic feature but enhancement in the core functionality of the platform like gamification elements.



1.4.1 Functionality in scope

The following domains and their services are included:

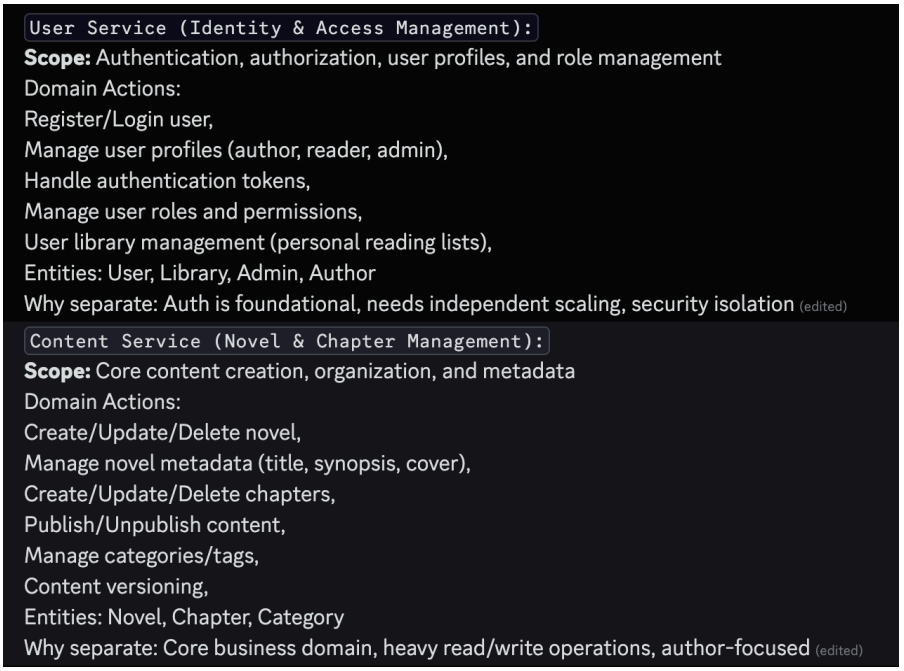
1. User Service

- Authentication & Authorization
- Profile management
- Role management (Reader → Author transitions and User → Admin transitions)

2. Content Service

- Novel / Chapters CRUD
- Novel metadata
- Category management

- Author-owned content management

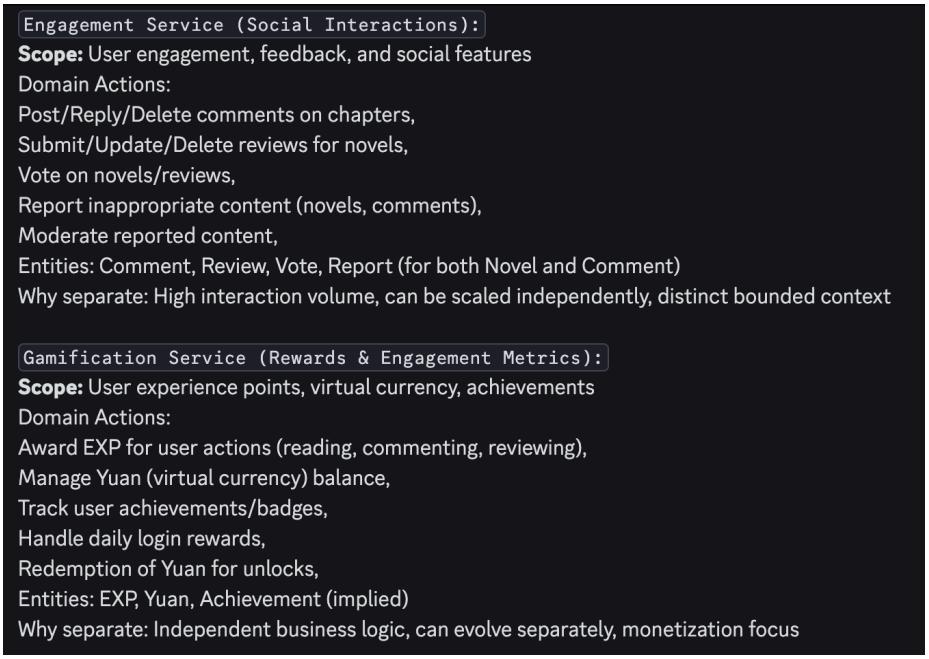


3. Gamification Service

- EXP, Level, Yuan
- Daily login rewards
- Leaderboard integration

4. Engagement Service

- Comments / Reviews CRUD
- Reports CRUD and Admin Moderation
- Votes



5. Analytics Service

- Reading history
- Engagement analytics
- Writer dashboard metrics
- Ranking (novel, author, user)
- Aggregation of stats from other services

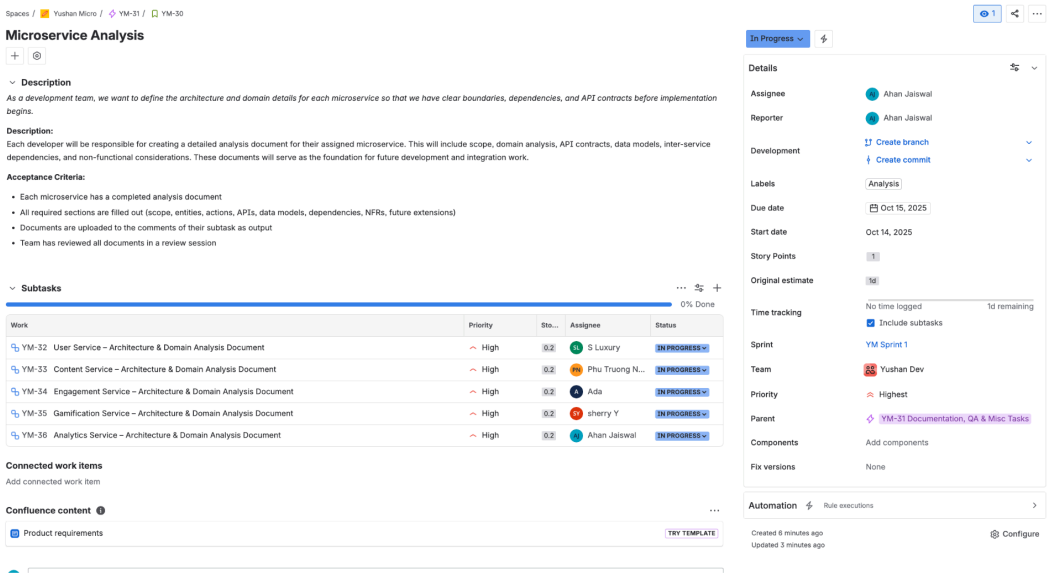
`Analytics Service (Ranking, History & Reporting):`
Scope: Data aggregation, rankings, reading history, and analytics
Domain Actions:
Track reading history,
Calculate novel rankings (trending, popular, new),
Calculate user rankings (top readers, authors),
Generate analytics reports,
Track novel views/engagement metrics,
Generate author dashboard data,
Entities: History, Ranking (for novels and users), Analytics
Why separate: Read-heavy, can use separate data stores (read replicas/OLAP), asynchronous processing

1.4.2 Functionality out of scope

The following are intentionally excluded for this phase:

- Complex multi-service transactions requiring Saga orchestration
- Real-time push notifications
- Mobile app support (but web app is adjust for mobile, and it's a PWA)
- AI-based analytics or recommendations
- Multi-language support
- External payment integration
- In-app messaging or chat

The microservices solution focuses on **core functional domains**, clean architecture, and scalability patterns, and full feature reproduction + enhancement to a platform level application.



1.4.3 Quality Attributes

The microservices architecture emphasizes:

Scalability

- Horizontal scaling of individual services
- Stateless containers for lightweight deployment
- Caching for high-traffic endpoints

Performance

- Database read-write separation (where applicable)
- Optimized inter-service communication
- Asynchronous messaging for high-volume updates

Reliability

- Service isolation preventing cascading failures
- Circuit breakers, retries, and fallback mechanisms
- Health checks and readiness/liveness probes

Security

- Centralized authentication (User Service)
- Fine-grained authorization through API gateway
- Secure inter-service communication
- Input validation and OWASP-aligned hardening

Maintainability

- Clean domain boundaries
- Independent codebases (multi-repo)
- Versioned API contracts and shared DTO libraries

Testability

- Unit, integration, and contract testing
- Test containers for environment parity
- Independent service CI pipelines

Observability

- Centralized logs
- Distributed tracing (trace IDs passed across services)
- Metrics dashboards
- Alerts and health monitoring

2. Project Conduct

2.1 Project Plan

The project followed a **one-sprint, time-boxed development plan** spanning from **13 Oct to 24 Oct 2025** (10 working days). A high-level **Work Breakdown Structure (WBS)** was created to organise work across the microservices, with each EPIC representing a single domain-bounded microservice.

Work Breakdown Structure (WBS)

The project was decomposed into the following major workstreams:

EPIC 1: User Service (Identity & Access Management) – Assignee: *Zhang Yan (S Luxury)*

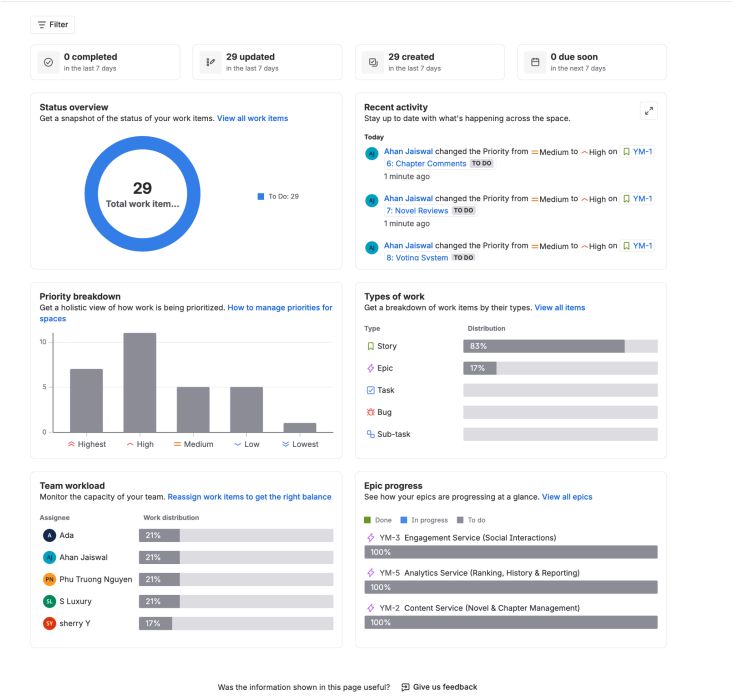
Estimated effort: 10 story points

- US-002 User Registration
- US-003 User Authentication
- US-001 User Profile Management
- US-004 Role-Based Access Control
- US-005 User Library Management

EPIC 2: Content Service (Novel & Chapter Management) – Assignee: *Phu Truong Nguyen*

Estimated effort: 10 story points

- US-006 Novel Creation & Management
- US-007 Chapter Management
- US-008 Content Publishing Workflow
- US-009 Category & Tag Management
- US-010 Discovery & Search (Basic)



EPIC 3: Engagement Service (Social Interactions) – Assignee: *Ada*

Estimated effort: 10 story points

- US-011 Comments
- US-012 Reviews
- US-013 Voting
- US-014 Content Reporting
- US-015 Moderation Tools

EPIC 4: Gamification Service (Rewards & EXP System) – Assignee: *sherry Y*

Estimated effort: 10 story points

- US-016 Daily Login Rewards
- US-017 Achievement System
- US-018 Virtual Currency (Yuan)
- US-019 Experience Points (EXP) System *(Two of these stories were sized at 3 points to balance workload, as this EPIC had only four stories.)*

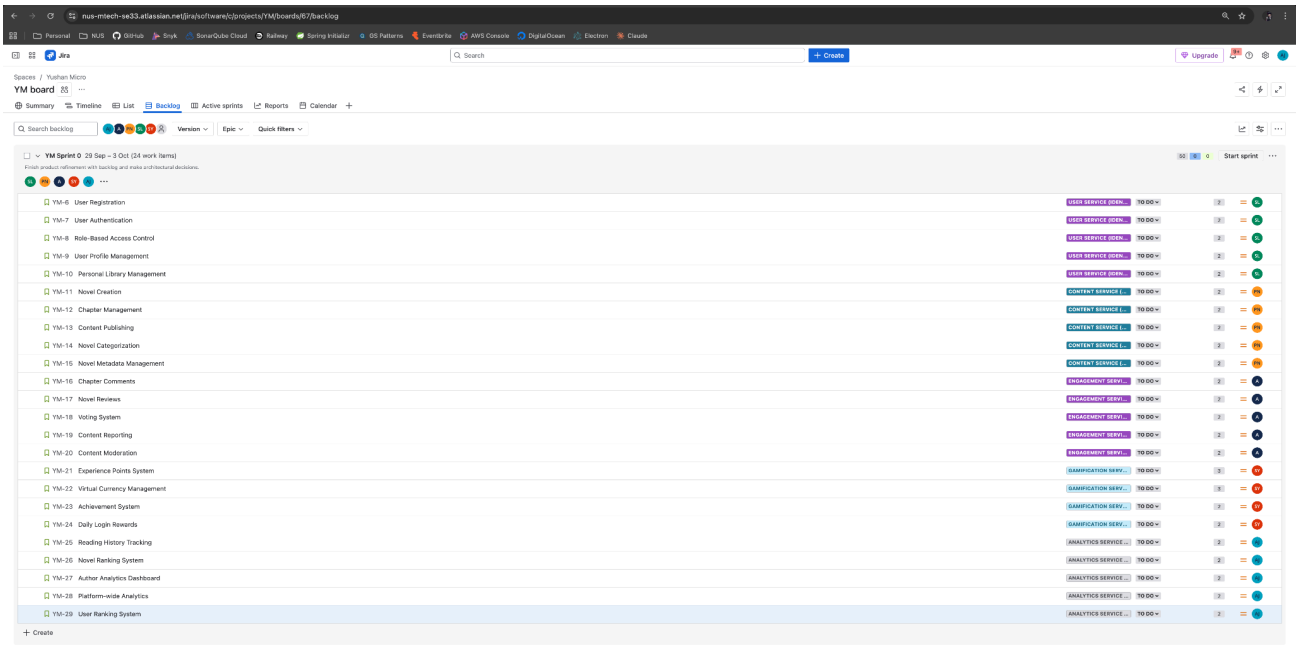
EPIC 5: Analytics Service (Ranking, History & Reporting) – Assignee: *Ahan Jaiswal*

Estimated effort: 10 story points

- US-020 Reading History Tracking
- US-021 Novel Ranking
- US-022 Author Analytics Dashboard
- US-023 Platform-wide Analytics
- US-024 Advanced Analytics Processing

EPIC 6: Documentation, QA & Misc Tasks – Assigned to all

- YM-30 Microservice Analysis (5 subtasks × 0.2 points per member)
- Documentation of service architecture, endpoints, ERDs, DDD models
- QA review and consistency checks



Implementation Priority & Sprint Plan

To ensure timely delivery, work was split into **two priority waves**:

Week 1 – Critical Path Deliverables

These services are foundational to system operation and cross-service integration:

- **US-002, US-003** – Authentication layer
- **US-006, US-007, US-008** – Core Content Creation & Publishing
- **US-020** – Basic Analytics (Reading History)

All Week 1 items were targeted for mandatory completion.

Week 2 – Feature Completion

Remaining features across domains:

- User Management (US-001, US-004, US-005)
- Content Metadata (US-009, US-010)
- Engagement Features (US-011 → US-015)
- Gamification (US-016 → US-019)
- Extended Analytics (US-021 → US-024)

All Week 2 items were also targeted for confirmed completion.

2.2 Project Status

As of the close of Sprint 1 (24 October 2025):

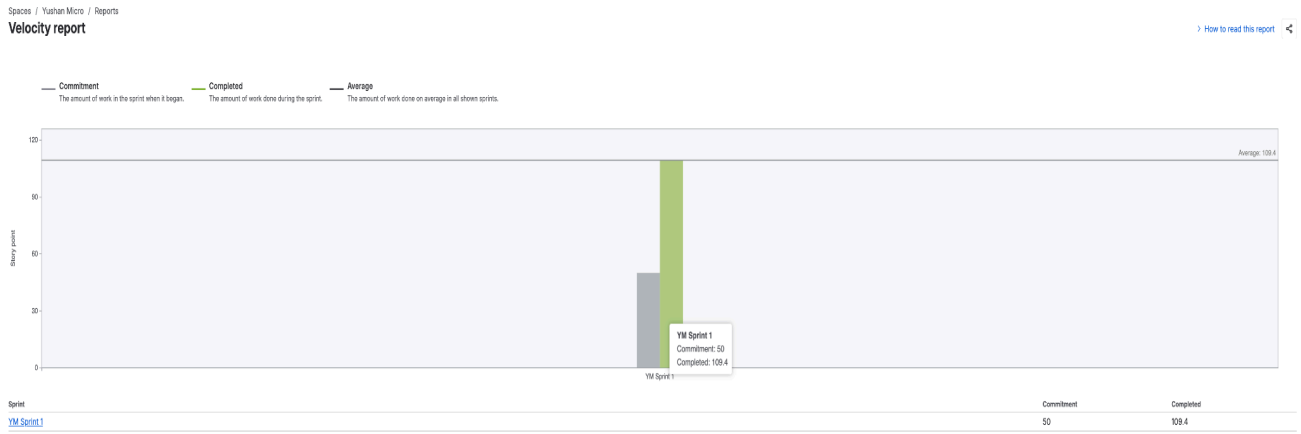
- All EPICs are completed (DONE).
- All 24 User Stories were delivered successfully.
- The project exceeded its planned velocity, completing more than 2× the committed work.

The Jira board shows:

- Committed: 50 story points
- Delivered: 109.4 story points

This reflects a highly productive sprint, with each team member completing beyond their scope.

Outstanding Issues: There are no outstanding open issues. All stories and subtasks across all six EPICs have been implemented, reviewed, and marked as Done.



2.3 Project Metrics

Milestone	Status	Date
Sprint Planning & EPIC Finalisation	Completed	13 Oct 2025
Microservice Domain Modelling	Completed	14 Oct 2025
Week 1 Critical Path Completion	Completed	18 Oct 2025
Week 2 Feature Completion	Completed	24 Oct 2025
All EPICs Delivered	Completed	24 Oct 2025
Sprint Review & Final Documentation	Completed	24 Oct 2025

YM Sprint 1 Performance:

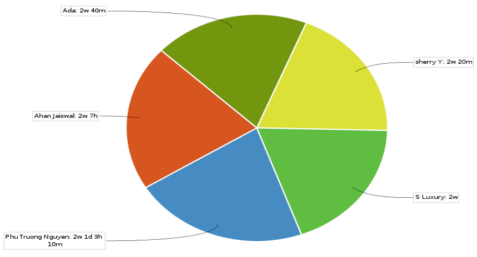
- Planned velocity: 50 points
- Actual delivered velocity: 109.4 points (218% of planned) All stories across User, Content, Engagement, Gamification, and Analytics services were completed.

Team Member	Time	Percentage
Ahan Jaiswal	**2w 7h**	**20%**
Ada	**2w 40m**	**19%**
sherry Y	**2w 20m**	**19%**
Zhang Yan (S Luxury)	**2w**	**19%**
Phu Truong Nguyen	**2w 1d 3h 10m**	**21%**

Workload Pie Chart Report

Project: Yushan Micro (Time Spent) by Assignee

Chart



3. Solution Overview

3.1 Logical Architecture & Design

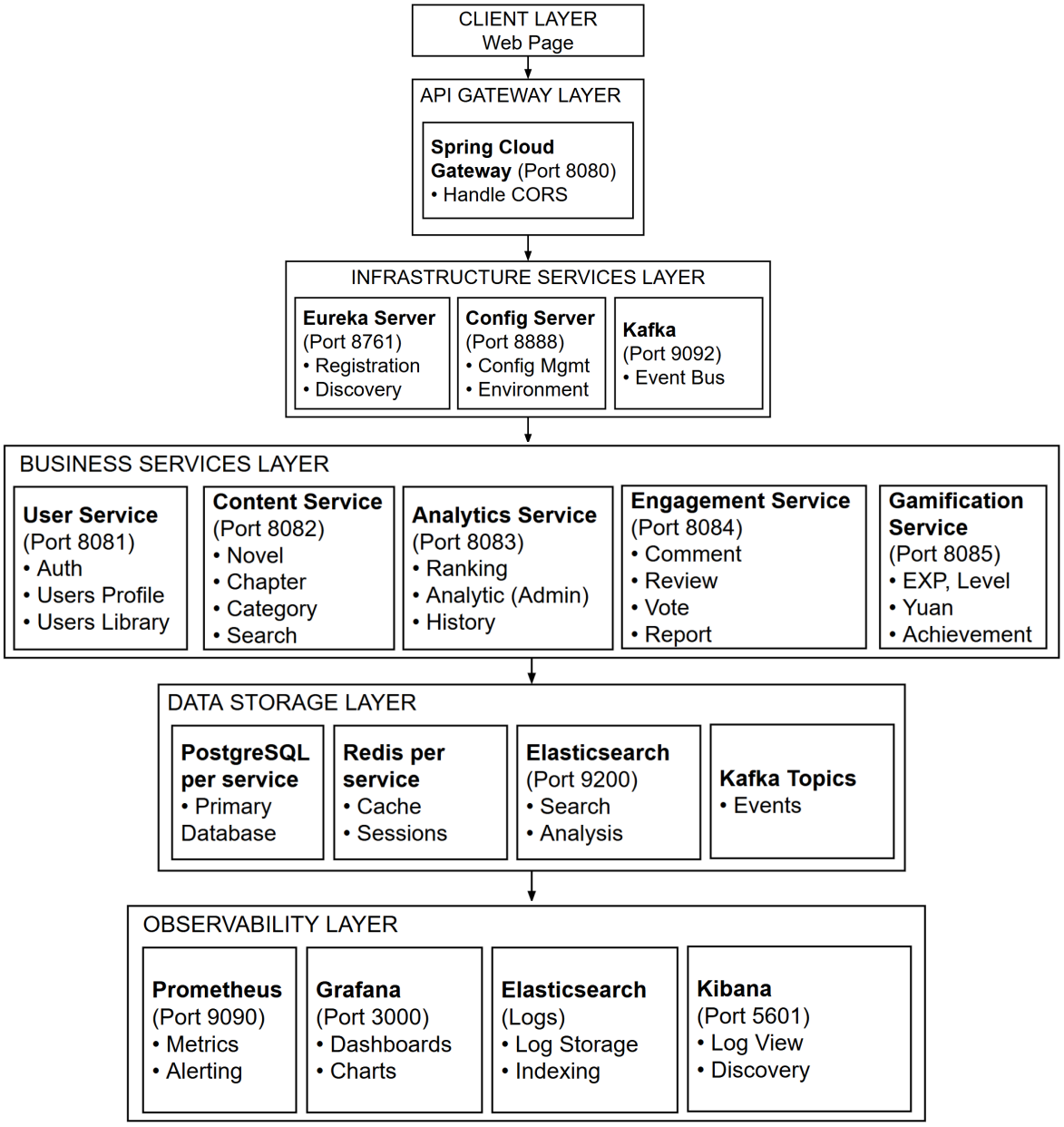
3.1.1 Key Architectural Decisions

The key architectural decisions taken are as follows:

Identifier	Description
AD-01	**Microservice Architecture** Rationale: Chosen over a Monolithic architecture to achieve superior scalability, maintainability, and team agility. This logical pattern allows services to be developed, deployed, and scaled independently. Trade-off: We accepted significantly higher operational complexity (managing multiple services, CI/CD pipelines, and inter-service communication) in exchange for long-term scalability and development velocity.
AD-02	**Event-Driven Architecture (EDA)** Services communicate primarily asynchronously via a central message bus. This decouples services, ensuring high resilience and extensibility (as detailed in 3.1.4). Rationale: Chosen over a purely synchronous, request-response model to achieve maximum service decoupling and system resilience. If one service is down, the other relative services can still work without interruption. Trade-off: We accepted the complexity of managing a message bus (Kafka) and the challenge of eventual consistency (data is not instantly replicated across all services) in exchange for this high availability and decoupling.
AD-03	**Database-per-Service** Each microservice manages its own database. This grants full autonomy, prevents data-level coupling, and allows each service to scale its persistence layer independently. Rationale: This pattern enforces true service autonomy and bounded context, preventing data-level coupling. It allows each service to select the best persistence technology for its needs and scale its data layer independently. Trade-off: We chose this over a simpler shared-database model. We fully accepted the major trade-off of this decision: complex cross-service data aggregation. As detailed in 3.4, queries that join data from multiple services now require complex API Composition.
AD-04	**API Gateway Pattern**

	A single entry point for all client requests. This decision centralizes cross-cutting concerns like security, routing, and load balancing, abstracting the internal service topology from clients. Rationale: Provides a single, unified, and secure entry point for all client applications, abstracting the internal service topology. It centralizes cross-cutting concerns, primarily routing and JWT-based security validation. Trade-off: This pattern introduces a new component that is mission-critical and a potential single point of failure. We accepted the responsibility of ensuring the Gateway is highly available in exchange for simplifying all downstream services and securing the platform's perimeter.

3.1.2 Tiers and Layers



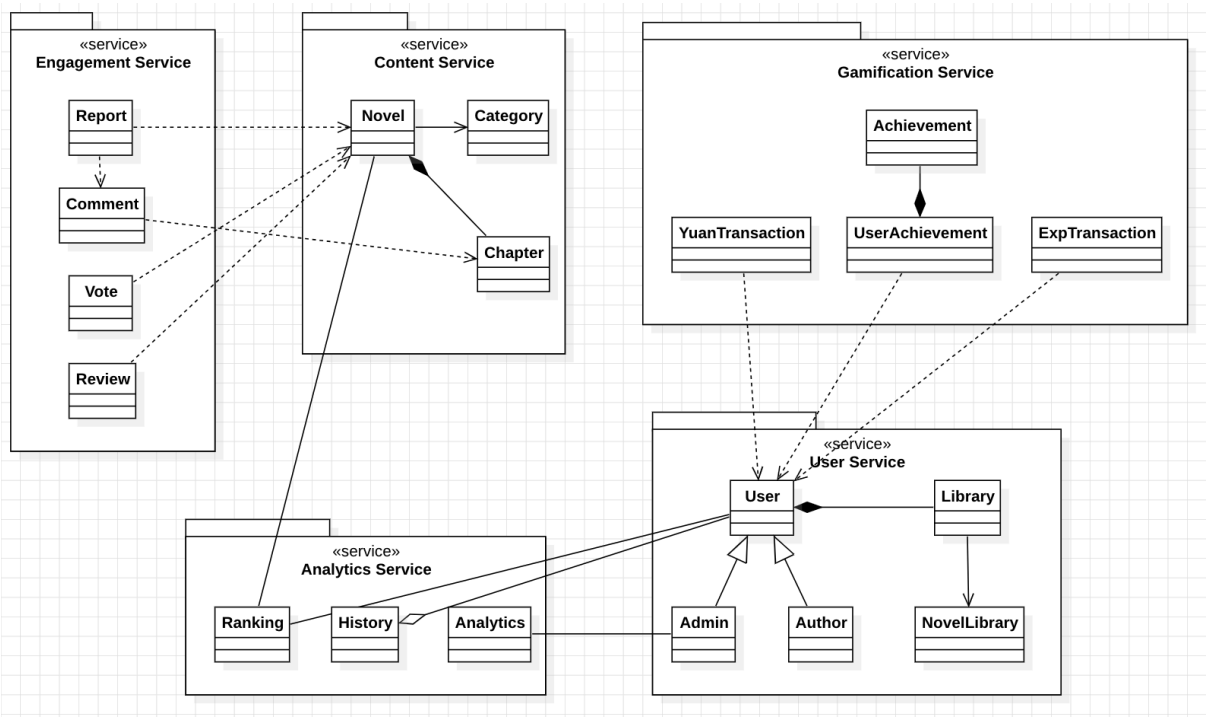
Tier	Layer(s)	Description & Responsibilities
Client Tier	Presentation	This tier is responsible for all user interaction and presentation logic. It consists of two independent Single Page Applications: the Yushan Webapp (for readers and authors) and the Yushan Admin Dashboard (for administrators).
Infrastructure Tier	Cross-Cutting	This foundational layer represents the shared infrastructure services that enable the entire platform. It includes the Service Discovery (Eureka) for service registration, the Message Bus (Kafka) for asynchronous communication, and the Container Orchestration platform that runs all services.
Application Tier	Gateway & Business Logic	This is the core of the system where all business logic resides. It contains two primary sub-layers: • API Gateway: The single entry point that handles all incoming client requests, performing routing, security (JWT validation), and load balancing. • Microservices: A collection of decoupled services (User, Content, Engagement, etc.) that encapsulate specific business domains.
Data Tier	Persistence	This tier is responsible for the persistence and storage of all application data. It consists of multiple independent databases (PostgreSQL) for transactional data, as well as shared Redis instances for caching.
Observability Layer	Cross-Cutting	This logical layer provides the monitoring, logging, and tracing capabilities essential for a distributed system. It aggregates logs, metrics (e.g., request latency, error rates), and distributed traces from all other tiers to provide a unified view of system health and performance.

3.1.3 Nodes and Subsystems

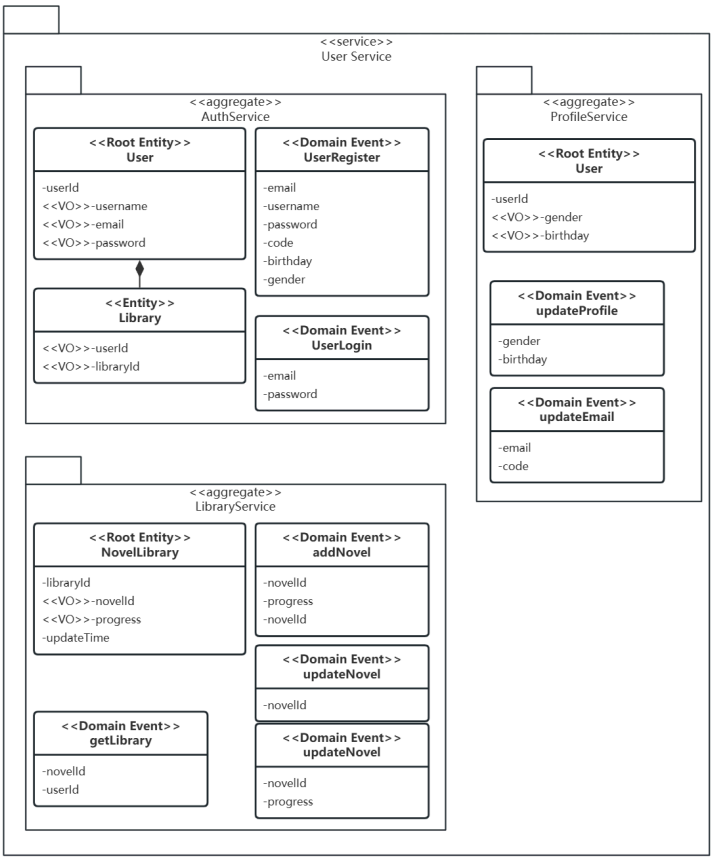
Node / Subsystem	Responsibilities
User Service	The core Identity & Access Management service. It handles user registration, login, and JWT generation. It also manages domains strongly coupled to the user, such as their profile, roles (reader, author, admin), and personal library.
Content Service	Manages the full lifecycle of literary content. Authors use this service to create, update, and publish novels and chapters. Readers use it to discover, query, and read content.
Engagement Service	Responsible for all interaction features with the novels and chapters. This includes managing comments (on chapters), reviews (on novels), votes novels and reports.
Gamification Service	Implements the platform's core incentive and reward mechanisms. It manages user experience points (EXP), virtual currency (Yuan), and achievement badges, reacting to activity events from other services.
Analytics Service	Handles data aggregation, processing, and reporting. It tracks user reading history, generates novel rankings (e.g., popular, new, trending), and provides authors with detailed engagement metrics for their work.

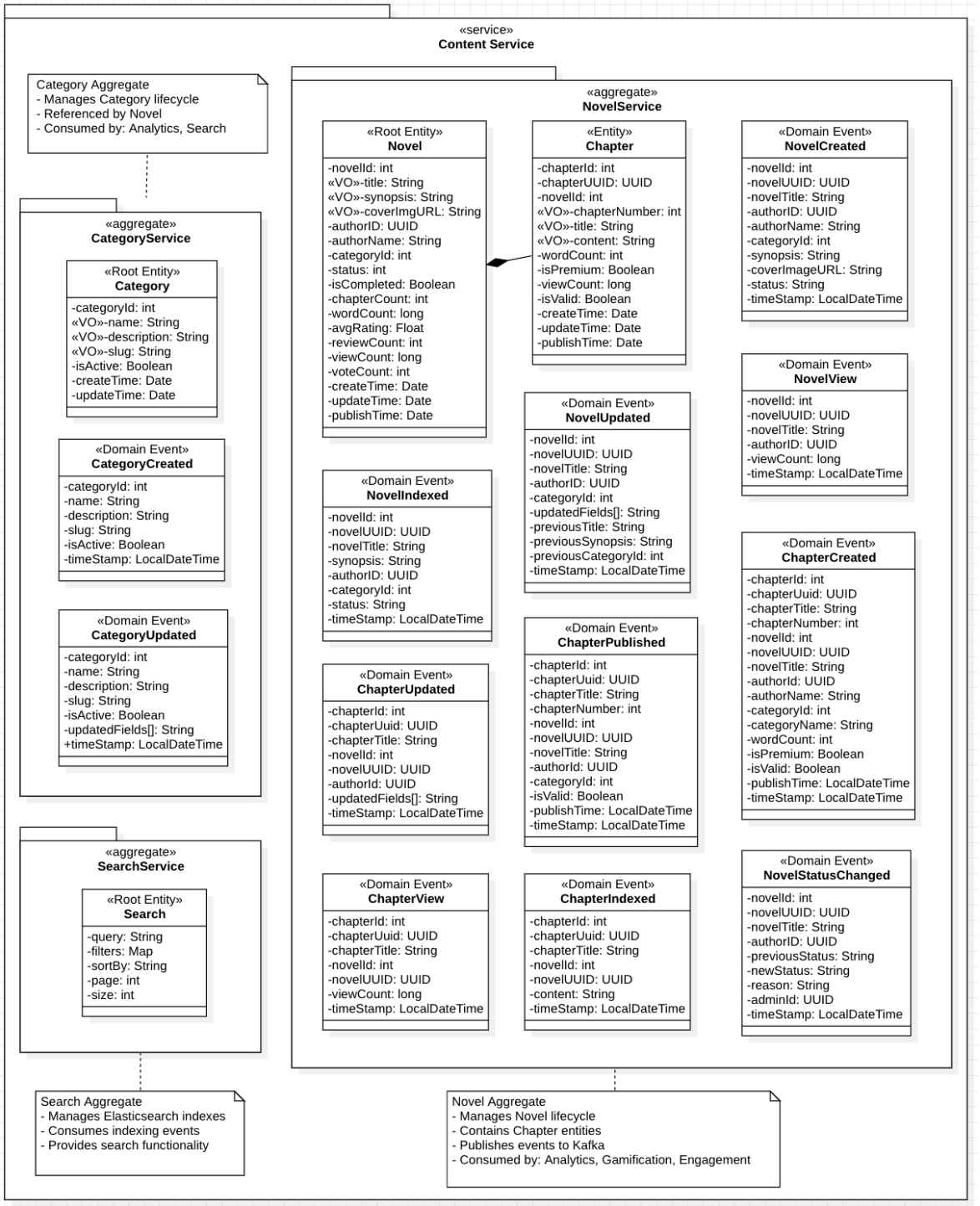
3.1.4 Platform Design

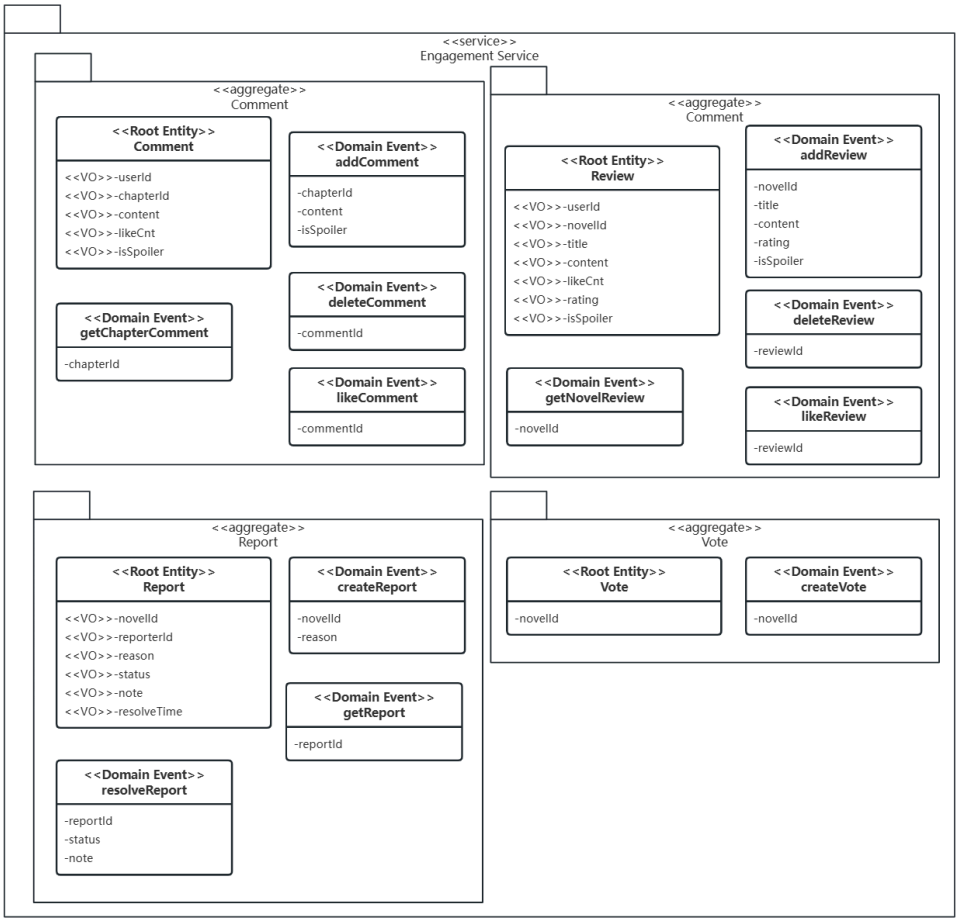
1. Bounded Contexts

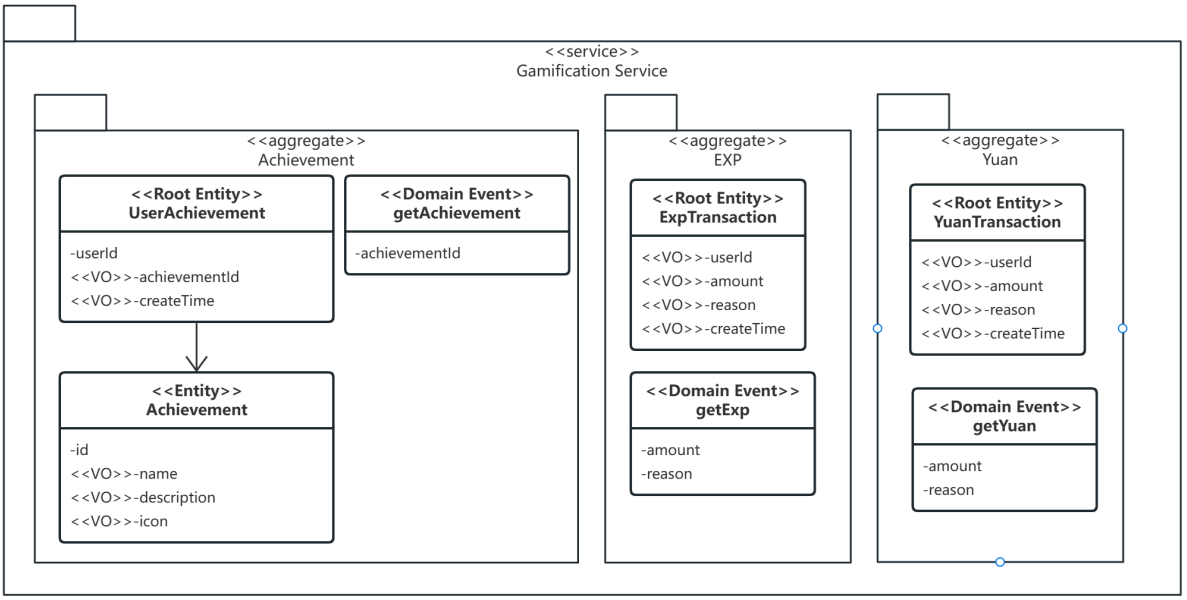


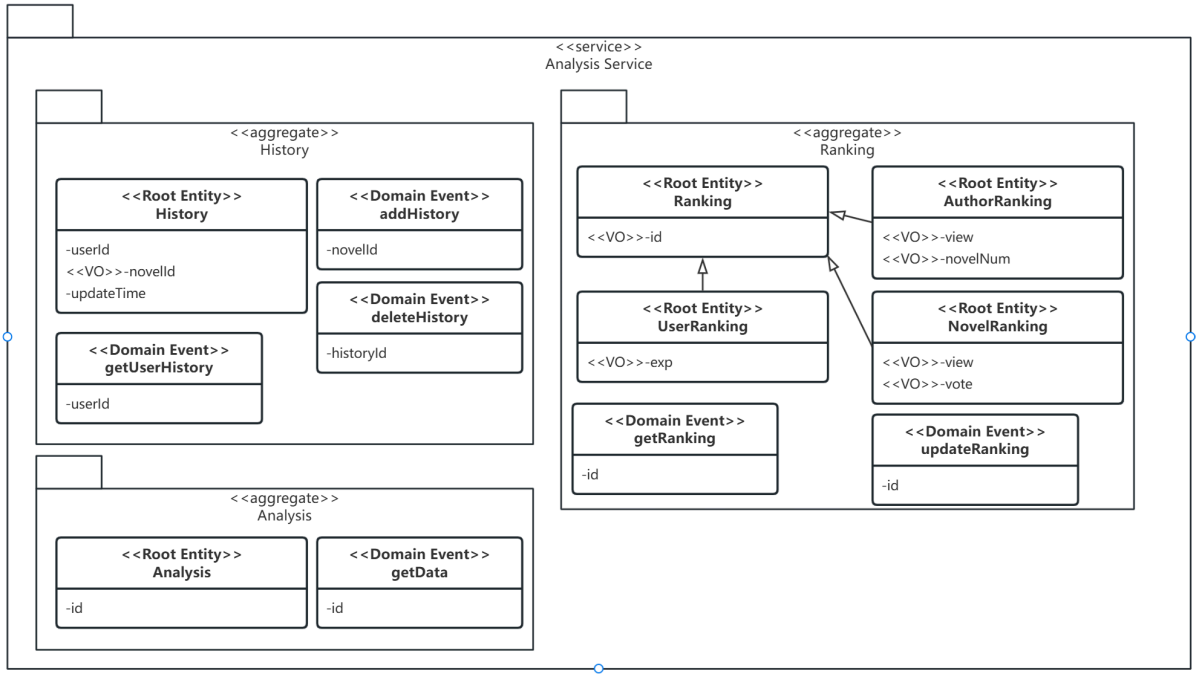
2. Design Aggregates and Domain Events




«service»
Content Service
Category Aggregate
- Manages Category lifecycle
- Referenced by Novel
- Consumed by: Analytics, Search
«aggregate»
CategoryService
«Root Entity»
Category
-categoryId: int
«VO»-name: String
«VO»-description: String
«VO»-slug: String
-isActive: Boolean
-createTime: Date
-updateTime: Date
«Domain Event»
CategoryCreated
-categoryId: int
-name: String
-description: String
-slug: String
-isActive: Boolean
-timeStamp: LocalDateTime
«Domain Event»
CategoryUpdated
-categoryId: int
-name: String
-description: String
-slug: String
-isActive: Boolean
-updatedFields[]: String
+timeStamp: LocalDateTime
«aggregate»
SearchService
«Root Entity»
Search
-query: String
-filters: Map
-sortBy: String
-page: int
-size: int
Search Aggregate
- Manages Elasticsearch indexes
- Consumes indexing events
- Provides search functionality
«aggregate»
NovelService
«Root Entity»
Novel
-novelId: int
«VO»-title: String
«VO»-synopsis: String
«VO»-coverImgURL: String
-authorID: UUID
-authorName: String
-categoryId: int
-status: int
-isCompleted: Boolean
-chapterCount: int
-wordCount: long
-avgRating: Float
-reviewCount: int
-viewCount: long
-voteCount: int
-createTime: Date
-updateTime: Date
-publishTime: Date
«Entity»
Chapter
-chapterId: int
-chapterUUID: UUID
-novelId: int
«VO»-chapterNumber: int
«VO»-title: String
«VO»-content: String
-wordCount: int
-isPremium: Boolean
-viewCount: long
-isValid: Boolean
-createTime: Date
-updateTime: Date
-publishTime: Date
«Domain Event»
NovelCreated
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorID: UUID
-authorName: String
-categoryId: int
-synopsis: String
-coverImageURL: String
-status: String
-timeStamp: LocalDateTime
«Domain Event»
NovelView
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorID: UUID
-viewCount: long
-timeStamp: LocalDateTime
«Domain Event»
NovelUpdated
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorID: UUID
-categoryId: int
-updatedFields[]: String
-previousTitle: String
-previousSynopsis: String
-previousCategoryId: int
-timeStamp: LocalDateTime
«Domain Event»
NovelIndexed
-novelId: int
-novelUUID: UUID
-novelTitle: String
-synopsis: String
-authorID: UUID
-categoryId: int
-status: String
-timeStamp: LocalDateTime
«Domain Event»
ChapterCreated
-chapterId: int
-chapterUuid: UUID
-chapterTitle: String
-chapterNumber: int
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorId: UUID
-authorName: String
-categoryId: int
-categoryName: String
-wordCount: int
-isPremium: Boolean
-isValid: Boolean
-publishTime: LocalDateTime
-timeStamp: LocalDateTime
«Domain Event»
ChapterPublished
-chapterId: int
-chapterUuid: UUID
-chapterTitle: String
-chapterNumber: int
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorId: UUID
-categoryId: int
-isValid: Boolean
-publishTime: LocalDateTime
-timeStamp: LocalDateTime
«Domain Event»
ChapterUpdated
-chapterId: int
-chapterUuid: UUID
-chapterTitle: String
-novelId: int
-novelUUID: UUID
-authorId: UUID
-updatedFields[]: String
-timeStamp: LocalDateTime
«Domain Event»
NovelStatusChanged
-novelId: int
-novelUUID: UUID
-novelTitle: String
-authorID: UUID
-previousStatus: String
-newStatus: String
-reason: String
-adminId: UUID
-timeStamp: LocalDateTime
«Domain Event»
ChapterView
-chapterId: int
-chapterUuid: UUID
-chapterTitle: String
-novelId: int
-novelUUID: UUID
-viewCount: long
-timeStamp: LocalDateTime
«Domain Event»
ChapterIndexed
-chapterId: int
-chapterUuid: UUID
-chapterTitle: String
-novelId: int
-novelUUID: UUID
-content: String
-timeStamp: LocalDateTime
Novel Aggregate
- Manages Novel lifecycle
- Contains Chapter entities
- Publishes events to Kafka
- Consumed by: Analytics, Gamification, Engagement


<<service>>
Engagement Service
<<aggregate>>
Comment
<<Root Entity>>
Comment
<<VO>>-userId
<<VO>>-chapterId
<<VO>>-content
<<VO>>-likeCnt
<<VO>>-isSpoiler
<<Domain Event>>
addComment
-chapterId
-content
-isSpoiler
<<Domain Event>>
deleteComment
-commentId
<<Domain Event>>
likeComment
-commentId
<<Domain Event>>
getChapterComment
-chapterId
<<aggregate>>
Comment
<<Root Entity>>
Review
<<VO>>-userId
<<VO>>-novelId
<<VO>>-title
<<VO>>-content
<<VO>>-likeCnt
<<VO>>-rating
<<VO>>-isSpoiler
<<Domain Event>>
addReview
-novelId
-title
-content
-rating
-isSpoiler
<<Domain Event>>
deleteReview
-reviewId
<<Domain Event>>
likeReview
-reviewId
<<Domain Event>>
getNovelReview
-novelId
<<aggregate>>
Report
<<Root Entity>>
Report
<<VO>>-novelId
<<VO>>-reporterId
<<VO>>-reason
<<VO>>-status
<<VO>>-note
<<VO>>-resolveTime
<<Domain Event>>
createReport
-novelId
-reason
<<Domain Event>>
getReport
-reportId
<<Domain Event>>
resolveReport
-reportId
-status
-note
<<aggregate>>
Vote
<<Root Entity>>
Vote
-novelId
<<Domain Event>>
createVote
-novelId


<<service>>
Gamification Service
<<aggregate>>
Achievement
<<Root Entity>>
UserAchievement
-userId
<<VO>>-achievementId
<<VO>>-createTime
<<Domain Event>>
getAchievement
-achievementId
<<Entity>>
Achievement
-id
<<VO>>-name
<<VO>>-description
<<VO>>-icon
<<aggregate>>
EXP
<<Root Entity>>
ExpTransaction
<<VO>>-userId
<<VO>>-amount
<<VO>>-reason
<<VO>>-createTime
<<Domain Event>>
getExp
-amount
-reason
<<aggregate>>
Yuan
<<Root Entity>>
YuanTransaction
<<VO>>-userId
<<VO>>-amount
<<VO>>-reason
<<VO>>-createTime
<<Domain Event>>
getYuan
-amount
-reason



3. Interactions

Our platform is designed as an extensible ecosystem rather than a fixed set of features, with an Event-Driven Architecture (EDA) at its core. This design choice provides maximum decoupling and system resilience.

Services communicate asynchronously via a shared Apache Kafka instance. When a service performs a business action, it publishes an event to a central topic, without knowing or caring which other services are listening. Other services subscribe to these topics and react independently.

Example Event Flow: New User Registration

A user submits their details to the API Gateway, which routes the request to the User Service.

The User Service validates the data, creates a new user in its PostgreSQL database, and generates a JWT.

Upon successful creation, the User Service publishes a UserRegisteredEvent message to the user.events Kafka topic.

Two different services are subscribed to this topic:

The Gamification Service consumes the event and grants the new user a "Welcome" badge and bonus EXP.

The Analytics Service consumes the event and increments its "New Users Today" counter.

This loose coupling ensures that the registration process is fast (it doesn't wait for the badge to be granted) and resilient (if the Gamification Service is down, registration is unaffected). This pattern allows for future extensibility; a new "Email Welcome" service could be added to subscribe to the same event without requiring any changes to the User Service.

These 2 tables below are our entire interactions between services asynchronously or synchronously.

Producer-Consumer Interactions - Asynchronous

Producer	Kafka Topic	Consumer

User Service • When register • When log in	user-events	Gamification Service • Reward
Engagement Service • When user comment	comment-events	Gamification Service • Reward
Engagement Service • When user review	review-events	Gamification Service • Reward
Engagement Service • When user vote	vote-events	Gamification Service • Reward
Every Service • When use trigger APIs	active	User Service • Update last active

Producer-Consumer Interactions - Synchronous

Producer	Consumer
User Service • GET /api/v1/users/{userId} • POST /api/v1/users/batch/get	Engagement Service Analytics Service
Content Service • GET /api/v1/novels/{novelId} • POST /api/v1/novels/batch/get • GET /api/v1/chapters/{chapterId} • POST /api/v1/chapters/batch/get	User Service Engagement Service Analytics Service
Content Service • GET /api/v1/novels/{novelId}/vote-count • POST /api/v1/novels/{novelId}/vote • PUT /api/v1/novels/{novelId}/rating • GET /api/v1/chapters/novel/{novelId}	Engagement Service
Gamification Service • GET /api/v1/gamification/votes/check	Engagement Service
Gamification Service • GET /api/v1/gamification/rank	Analytics Service

3.2 Physical Architecture & Design

3.2.1 Key Architectural Decisions

The key architectural decisions taken are as follows:

Identifier	Description
AD-05	*Domain-aligned microservices (vs. monolith or modular monolith)* ● *Context* *The platform spans distinct domains: user, content, engagement, gamification, analytics.* ● *Options* ○ *1) Monolith 2) Modular monolith 3) Domain-split microservices*

	● *Decision* *Adopt domain-split microservices: user, content, engagement, gamification, analytics.* ● *Rationale* ○ *Independent deployability and scaling per domain* ○ *Team ownership aligns with bounded contexts* ○ *Enables different data models and storage choices per service* ● *Trade-offs* ○ *Distributed systems complexity (network, retries, tracing)* ○ *Data consistency shifts to cross-service patterns (events/Saga)* ● *Consequences* ○ *Introduce API Gateway and service discovery* ○ *DB-per-service pattern and asynchronous eventing between services*
AD-06	*Dockerized Java backends with Spring Boot (container deployment units)* ● *Context* *All backend services are written in Java and need consistent packaging/running across environments.* ● *Options* ○ *1) Native processes (JARs on VMs) 2) Containers with Java/Spring Boot 3) Alternative runtimes (Go/Node)* ● *Decision* *Package each service as a Docker container using Spring Boot.* ● *Rationale* ○ *Spring ecosystem (Actuator, Spring Cloud) and team expertise* ○ *Uniform health probes (/actuator/health) and config model* ○ *Portable, immutable deployment units* ● *Trade-offs* ○ *Higher memory footprint and startup time vs. some alternatives* ○ *Image and base JDK hardening required* ● *Consequences* ○ *Standardize Dockerfiles, base images, and vulnerability scanning* ○ *Define resource limits and readiness/liveness probes*
AD-07	*Infrastructure on DigitalOcean, provisioned via Terraform (centralized runtime on Droplets)* ● *Context* *Early-stage cost and operational simplicity favored VM-based infrastructure.* ● *Options* ○ *1) Managed Kubernetes / App Platform 2) Self-managed Kubernetes 3) DO Droplets + Terraform (VMs)* ● *Decision* *Use DigitalOcean Droplets provisioned and managed by Terraform; centralize infra runtime.* ● *Rationale* ○ *Lower cognitive/ops overhead than Kubernetes at current scale* ○ *Full control with predictable costs; reproducible IaC provisioning* ● *Trade-offs* ○ *Fewer built-in autoscaling/rollout features vs. Kubernetes/App Platform* ○ *Capacity planning and failover require operational playbooks* ● *Consequences* ○ *One or more “infrastructure” Droplets host Nginx, API Gateway, Config Server, Eureka*

	○ *Services run on dedicated Droplets; DO Load Balancer fronts the public entry*
AD-08	*Centralized configuration and service discovery (Spring Cloud Config + Eureka)* ● *Context* *Multiple services require runtime configuration and discovery without hard-coded endpoints.* ● *Options* ○ *1) Static configs and hostnames 2) Consul + external config 3) Spring Cloud Config + Eureka* ● *Decision* *Use Spring Cloud Config Server for configuration and Eureka for service discovery.* ● *Rationale* ○ *Native Spring integration; consistent bootstrap across services* ○ *Enables dynamic service registration and runtime config refresh* ● *Trade-offs* ○ *Additional moving parts; need HA and backup strategy for Config/Eureka* ○ *Bootstrapping order and secured access must be carefully designed* ● *Consequences* ○ *Host Config/Eureka on infra Droplets; restrict access via firewall rules* ○ *Store configs in Git; protect secrets via environment/secret managers*
AD-09	*Single public API Gateway entry at https://yushan.duckdns.org/api/v1* ● *Context* *Public traffic must be terminated and routed through a single controllable ingress.* ● *Options* ○ *1) Nginx only 2) Kong/Traefik 3) Spring Cloud Gateway behind Nginx/DO LB* ● *Decision* *Expose a single API entry via Nginx → Spring Cloud Gateway at https://yushan.duckdns.org/api/v1.* ● *Rationale* ○ *Central place for routing, auth, rate limits, cross-cutting filters* ○ *Leverages Spring filters and unified request tracing* ● *Trade-offs* ○ *Gateway becomes a potential bottleneck and single point of failure* ○ *Additional hop may add small latency* ● *Consequences* ○ *Maintain routing rules and health checks; automate TLS issuance/renewal* ○ *Plan blue/green or canary at gateway level for safer releases*
AD-10	*Database per service using PostgreSQL (vs. shared schema/instance)* ● *Context* *Each domain has different consistency and query needs.* ● *Options* ○ *1) Shared DB/schema 2) Schema-per-service 3) DB-per-service (PostgreSQL)* ● *Decision* *Adopt DB-per-service with PostgreSQL instances per domain.* ● *Rationale* ○ *Strong ownership and autonomy; avoids schema coupling* ○ *PostgreSQL offers ACID, JSONB, robust indexing and extensions*

	• *Trade-offs*
	  ◦ *No cross-service joins; embrace API composition/queries per service*
	  ◦ *Cross-service transactions require Saga/Outbox patterns*
	• *Consequences*
	  ◦ *Use Flyway for versioned migrations per service*
	  ◦ *Backups, monitoring, and capacity handled per database*

AD-11	*Asynchronous events for decoupling (Kafka-based)*
	• *Context*
	Engagement, gamification, and analytics require fan-out and replayable streams.
	• *Options*
	  ◦ *1) Pure synchronous REST 2) RabbitMQ 3) Kafka topics for domain events*
	• *Decision*
	Use Kafka topics to publish/consume domain events for decoupled workflows.
	• *Rationale*
	  ◦ *High throughput, partitioning, durable retention, replay for analytics*
	  ◦ *Reduces coupling between write paths and downstream processors*
	• *Trade-offs*
	  ◦ *Eventual consistency; message ordering/duplication concerns*
	  ◦ *Operational overhead (brokers, retention, DLQs)*
	• *Consequences*
	  ◦ *Enforce idempotency keys; implement retries and DLQ (events.dead-letter)*
	  ◦ *Document topic taxonomy and retention windows; provide replay tooling*

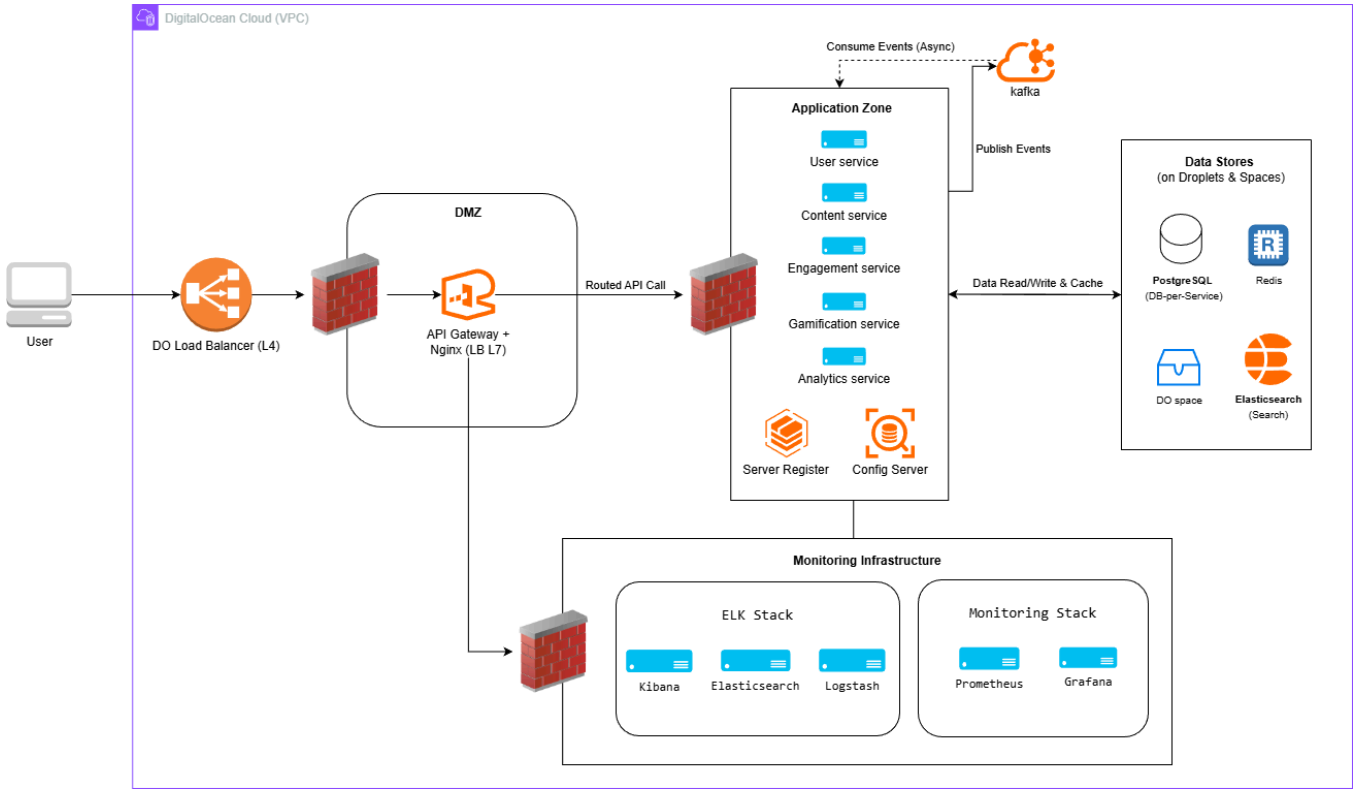
AD-12	*Redis for real-time counters and OTP (vs. database-backed counters)*
	• *Context*
	Low-latency counters/rankings and one-time codes are performance sensitive.
	• *Options*
	  ◦ *1) SQL counters 2) Kafka Streams 3) Redis atomic ops with TTL*
	• *Decision*
	Use Redis for atomic increments, leaderboards, and OTP with TTL semantics.
	• *Rationale*
	  ◦ *Sub-millisecond operations; simple primitives (INCR, sorted sets)*
	  ◦ *Natural TTL support for ephemeral data (OTP)*
	• *Trade-offs*
	  ◦ *Cache-to-source drift; potential data loss if not persisted/durable*
	  ◦ *Requires periodic reconciliation with authoritative DB*
	• *Consequences*
	  ◦ *Nightly/periodic backfill to Postgres; define eviction and TTL policies*
	  ◦ *Rate-limit and abuse-prevention counters live in Redis*

AD-13	*CI/CD with GitHub Actions; IaC with Terraform*
	• *Context*
	Need repeatable build/test/deploy pipelines and infrastructure automation.
	• *Options*
	  ◦ *1) Manual scripts 2) Jenkins/GitLab CI 3) GitHub Actions + Terraform*
	• *Decision*
	Use GitHub Actions for CI/CD pipelines and Terraform for infrastructure provisioning.
	CI reference: https://github.com/maugus0/yushan-platform-frontend/actions
	• *Rationale*
	  ◦ *Tight GitHub integration, PR gating, reusable workflows*

	○ *Declarative, reviewable infrastructure changes via Terraform* ● *Trade-offs* ○ *Runner concurrency/quotas; vendor coupling to GitHub* ○ *Terraform state management and change coordination required* ● *Consequences* ○ *Standardize build/test stages, image tagging, and deploy jobs per service* ○ *Versioned Terraform modules; plan/apply gates; policy checks*

3.2.2 Technology and Services

The physical diagram:



Incoming traffic from users first reaches the DigitalOcean Load Balancer (Layer 4), which distributes requests to the API Gateway + Nginx component hosted inside the DMZ. The gateway acts as a Layer 7 reverse proxy and load balancer, routing API calls securely to services in the Application Zone.

The Application Zone contains all backend microservices (User, Content, Engagement, Gamification, and Analytics), along with supporting components such as the Config Server and Service Registry. Services can publish and consume asynchronous events through Kafka, and interact with the Data Stores layer, which includes per-service PostgreSQL databases, Redis, Elasticsearch, and object storage (DO Spaces).

All logs and metrics generated by the system flow into the Monitoring Infrastructure, consisting of the ELK Stack (Kibana, Elasticsearch, Logstash) for centralized logging, and Prometheus + Grafana for metrics and service monitoring. This architecture separates public-facing, internal, data, and monitoring zones to improve security, clarity, and operational reliability.

Main technology and services:

Category	Selection	Purpose	Key Rationale	Key Trade-offs
Runtime	Java + Spring Boot (Docker)	Service implementation	Mature ecosystem, Actuator, Spring Cloud; team skills	Higher memory/startup vs Go/Node; base image hardening
Packaging	Docker containers	Immutable deploy units	Consistent across environments; health probes	Image supply-chain mgmt required
Cloud	DigitalOcean Droplets + VPC + LB	Compute, networking, ingress	Cost-effective, simple at current scale; full control	Fewer managed features vs K8s/App Platform
IaC	Terraform	Provision infrastructure	Repeatable, reviewable changes; drift control	State management; planning discipline
Config/Discovery	Spring Cloud Config + Eureka	Externalized config & service registry	Native Spring integration; simple bootstrap	Add'l components to maintain; HA considerations
API Ingress	Nginx → Spring Cloud Gateway	Single public entry	Centralized auth/routing/observability	Potential bottleneck; needs HA and rate-limit
Databases	PostgreSQL (DB-per-service)	Transactional data	ACID, JSONB, strong indexing; autonomy per domain	No cross-service joins; Saga/Outbox complexity
Caching	Redis	Realtime counters, OTP, leaderboard	Atomic ops, TTL; sub-ms latency	Cache-source drift; persistence planning

Messaging	Apache Kafka	Async decoupling	High throughput, replay, partitions	Operability (brokers, retention, DLQ); eventual consistency
Search	Elasticsearch	Full-text & aggregations	Rich query/analysis, scalable	Resource-intensive; index sync/ops cost
Object storage	DigitalOcean Spaces	Large content bodies/media	S3-compatible; cost-effective	Read latency vs DB; signed URL lifecycle
Monitoring	Prometheus + Grafana	Metrics/SLA dashboards	Open standards (OTel exporters); flexible dashboards	Self-managed alerts; scraping footprint
Logging	ELK (Elasticsearch, Logstash, Kibana)	Centralized logs	Powerful search & visualization	Storage/ingest cost; pipeline complexity
CI/CD	GitHub Actions	Build/test/deploy	Tight GitHub integration; reusable workflows	Runner quotas; secret hygiene
DNS/TLS	duckdns.org + Let's Encrypt	Public routing and TLS	Automated cert renewal; zero cost	Let's Encrypt rate limits; cert ops reliability

3.2.3 Persistence Design

1. Data management

- Pattern: Database-per-service (DB-per-service) for domain autonomy and strong consistency within each service boundary.
- Primary transactional store: PostgreSQL per service (user, content, engagement, gamification, analytics). Managed as separate instances on DigitalOcean Droplets inside the platform VPC.
- Large bodies & media: DigitalOcean Spaces, private buckets in the same region; databases store only object keys or presigned URLs.
- Full-text search: Elasticsearch cluster inside the VPC; content service builds/maintains indices.
- Ephemeral/low-latency state: Redis (rankings, OTP, session caches).
- Asynchronous messaging: Kafka topics for domain events, with DLQs for poison messages.

- Configuration & discovery: Spring Cloud Config + Eureka control plane; services consume stores via VPC-only addresses.

2. Special considerations (transactions, concurrency, reliability)

- Intra-service transactions stay within a single PostgreSQL instance (ACID). Isolation default: READ COMMITTED; elevate to `SELECT … FOR UPDATE` for monetary updates (gamification).
- Cross-service workflows avoid 2PC; use Outbox pattern (DB transaction + async publish to Kafka) and Saga compensation for multi-step flows (purchase, rewards).
- Redis is authoritative only for ephemeral data; periodic reconciliation jobs backfill rankings into PostgreSQL to prevent drift.
- Elasticsearch indices are treated as non-authoritative; rebuild/reindex tooling and snapshots are required to recover from index loss.
- Idempotency across event consumers enforced via event IDs and processed-event tables; Kafka DLQ retains failures for replay.

3. Data persistence and retention

Data Type	Persist beyond session	Retention
Users/profile	Yes	Until deletion
Sessions / OTP (Redis)	No	5 minutes via TTL
Content metadata	Yes	Until removal
Chapter bodies/media	Yes	Lifecycle policy (archive/delete on takedown)
Ledger transactions (Yuan)	Yes (immutable)	>=7 years
Search indices	Non-authoritative	Snapshot 30 days; rebuildable
Analytics raw	Yes	30 days; aggregates ≥12 months
rankings	Partially	Redis ephemeral; nightly reconciliation to DB

4. Information architecture by data group

1) Users & Auth

- Data format: relational rows (UUID keys), authentication artifacts; short-lived tokens/OTP in Redis.
- Proposed storage technology: PostgreSQL (user-db); Redis for sessions/OTP.
- Security requirements & controls: TLS in transit; bcrypt/argon2 password hashing; secrets via secret manager; least-privilege DB accounts; audit logging for auth/admin actions; IP allowlists and rate limits at gateway.
- Trade-off consideration: strong consistency vs. higher operational rigor (key rotation, audits).
- Persistence/retention: user/profile until deletion; verification/reset tokens 24h; sessions/OTP 5 minutes (Redis TTL).

2) Content metadata & index

- Data format: relational rows for canonical metadata (novel, chapter metadata); JSON documents in search index.
- Proposed storage technology: PostgreSQL (content-db) + Elasticsearch.
- Security requirements & controls: VPC-only endpoints; ES auth/TLS; index snapshots to Spaces; signed URLs used when joining with object storage.
- Trade-off: fast search/aggregation vs. index maintenance and consistency overhead.
- Persistence/retention: metadata until content removal; search indices follow metadata lifecycle.

3) Chapter body & media

- Data format: flat objects (text blobs, images); DB holds object keys/presigned URLs.
- Proposed storage technology: DigitalOcean Spaces.
- Security requirements & controls: private buckets, presigned short-lived URLs, server-side encryption, access logs.
- Trade-off: scalable and cost-effective vs. the need for lifecycle/restore planning.
- Persistence/retention: persistent; archive or delete per takedown policy.

4) Full-text search

- Data format: JSON documents.
- Proposed storage technology: Elasticsearch with scheduled snapshots to Spaces.
- Security requirements & controls: TLS, credentials, network segmentation.
- Trade-off: high performance vs. operational cost; not a system of record.
- Persistence/retention: snapshots retained 30 days; indices rebuilt as needed.

5) Ranking, counters & OTP

- Data format: key/value, sorted sets (Redis).
- Proposed storage technology: Redis (AOF on critical counters; pure cache for OTP).
- Security requirements & controls: TLS, ACLs, isolated instances for rate-limit/OTP, TTLs.
- Trade-off: sub-ms latency vs. persistence risk; requires scheduled reconciliation.
- Persistence/retention: ephemeral; OTP minutes; counters reconciled nightly.

6) Event stream

- Data format: JSON/Avro messages (eventId, timestamps, payload).
- Proposed storage technology: Kafka (topics per domain).
- Security requirements & controls: TLS, SASL/ACLs, Schema Registry, DLQ, consumer lag monitoring.
- Trade-off: decoupling/replay vs. eventual consistency and broker ops.
- Persistence/retention: 7 days typical for domain topics; DLQ 90+ days.

7) Engagement (comments/reviews/likes)

- Data format: relational rows; Redis cache for hot reads.
- Proposed storage technology: PostgreSQL (engagement-db) + Redis (cache).
- Security requirements & controls: input sanitization; moderation flags; rate limiting; TLS; backups.
- Trade-off: RDBMS integrity; cache helps read scalability.
- Persistence/retention: persistent; moderation/audit logs per policy.

8) Gamification & transactions (Yuan ledger)

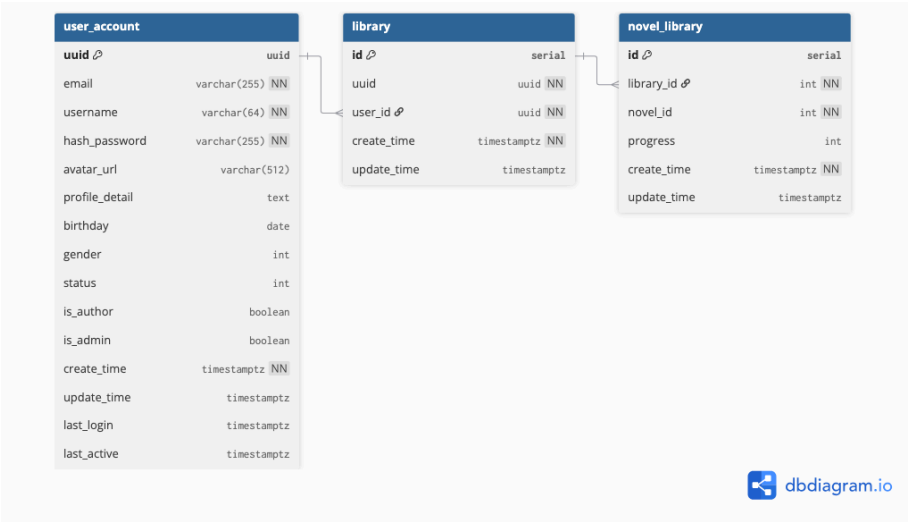
- Data format: immutable ledger entries (balances, transactions).
- Proposed storage technology: PostgreSQL (gamification-db).
- Security requirements & controls: strict audit trails; encryption at rest; RBAC; anomaly detection.
- Trade-off: requires strict ACID and careful locking; Outbox/Saga for cross-service steps.
- Persistence/retention: ≥7 years (regulatory/audit).

9) Analytics & history

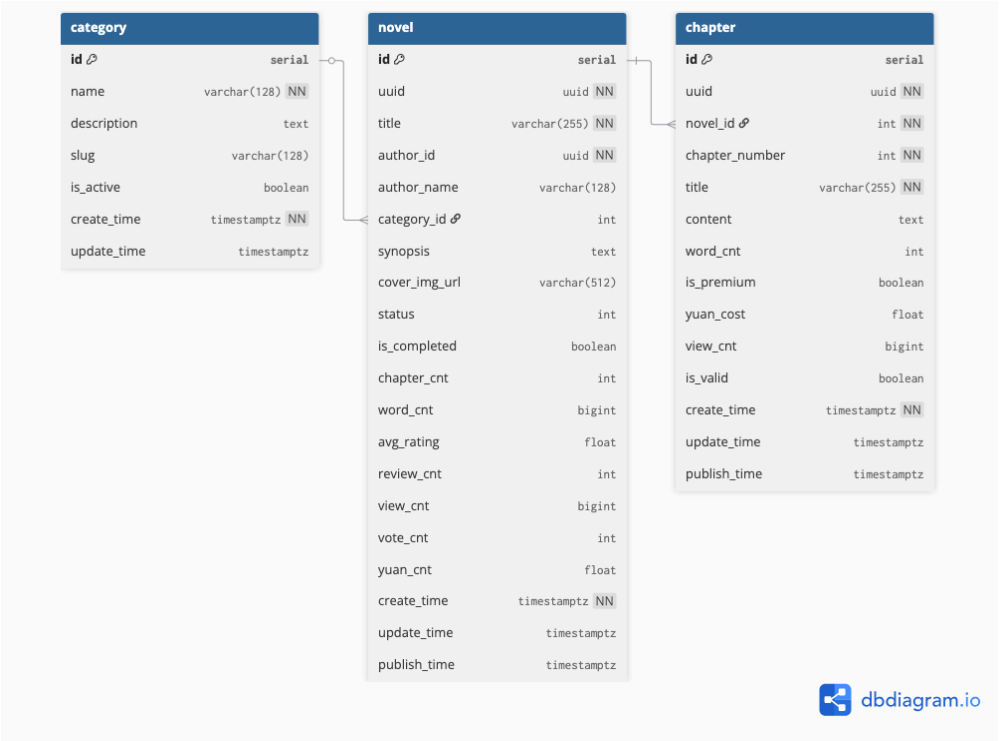
- Data format: history rows and aggregates; JSON payloads; event ingestion via Kafka.
- Proposed storage technology: PostgreSQL (analytics-db).
- Security requirements & controls: PII minimization/anonymization; dashboard access control; retention policies.
- Trade-off: storage growth; aggregation and TTL jobs required.
- Persistence/retention: raw events 30 days; aggregates ≥12 months.

5. Logical Data Model

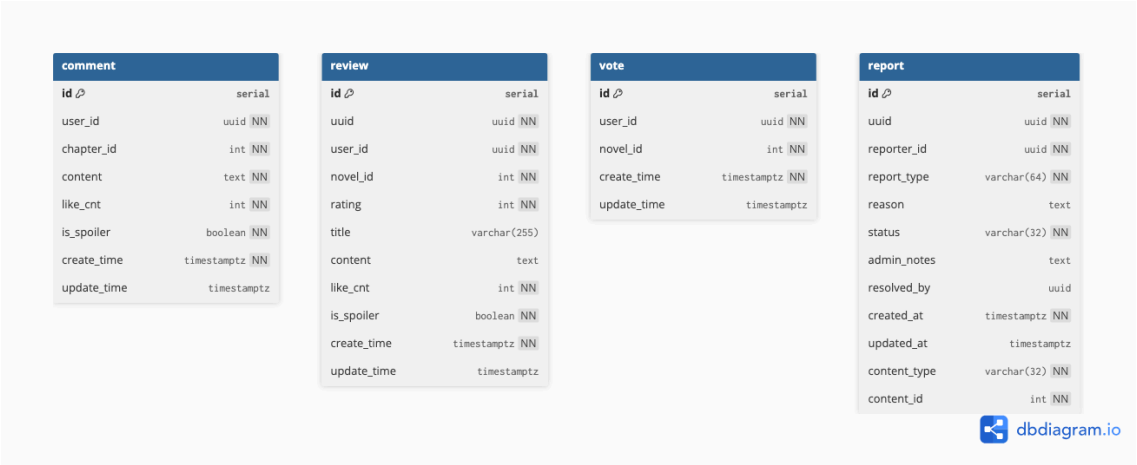
1) User Service (user-db)



2) Content Service (content-db)

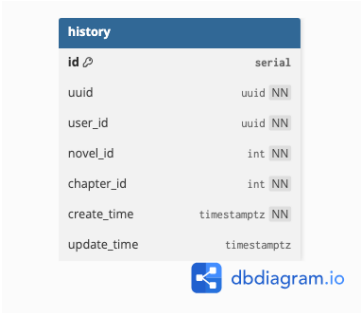


3) Engagement Service (engagement-db)



4) Gamification Service (gamification-db)

5) Analytics Service (analytics-db)



6. Representative NoSQL/document schemas

1）Elasticsearch ChapterDocument (search index)

```
{
 "chapterId": 123,
 "uuid": "d6a1e9f2-xxxx-xxxx-xxxx-abcdef123456",
 "novelId": 45,
 "chapterNumber": 1,
 "title": "Chapter One",
```

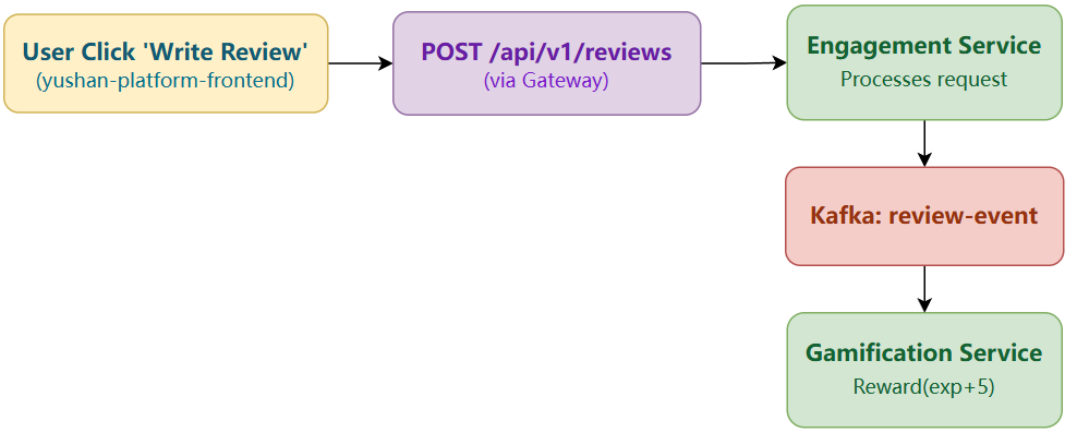
```
 "content": "Text may be truncated for indexing",
 "wordCount": 2045,
 "isPremium": false,
 "viewCnt": 12345,
 "publishTime": "2025-10-01T12:00:00Z",
 "tags": ["fantasy","adventure"],
 "createdAt": "2025-09-01T09:00:00Z",
 "updatedAt": "2025-09-10T11:00:00Z"
}
```

2）Kafka domain event (example payload envelope)

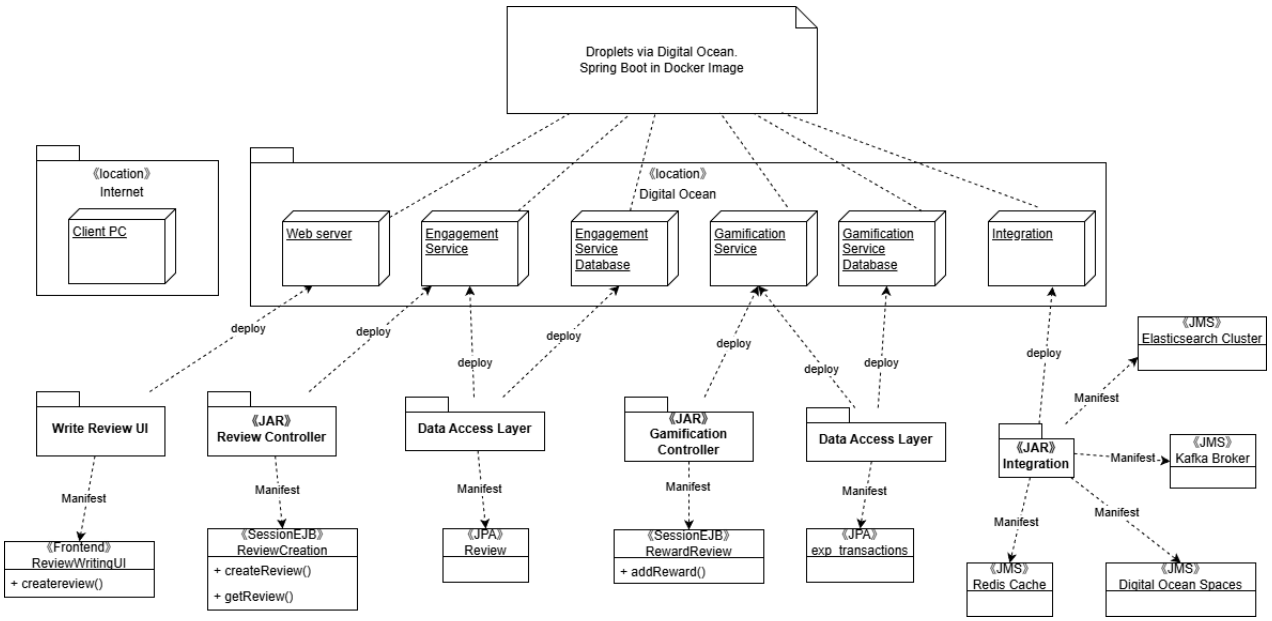
```
{
 "eventId": "evt-2025-11-13-0001",
 "eventType": "ChapterCreated",
 "occurredAt": "2025-11-13T07:00:00Z",
 "sourceService": "content-service",
 "payload": {
  "chapterUuid": "d6a1e9f2-xxxx-xxxx-xxxx-abcdef123456",
  "novelUuid": "a1b2c3-xxxx-xxxx-xxxx-zzzzzz999999",
  "authorUuid": "550e8400-e29b-41d4-a716-446655440001",
  "title": "Chapter One",
  "publishTime": "2025-10-01T12:00:00Z"
 }
}
```

3.2.4 Detailed Design

3.2.4.1 Reader writes reviews



Detailed Deployment Elements



Intent

Allow a reader (authenticated USER/AUTHOR/ADMIN) to create a single review per novel, update novel rating metrics, and trigger an experience reward in the gamification domain.

Actors / Services

- Frontend (yushan-platform-frontend) via API Gateway.
- Engagement Service (review lifecycle).
- Content Service (novel existence / rating updates).
- Gamification Service (EXP reward).
- Kafka (domain event: review-event).
- Redis (optional counters / caching).
- PostgreSQL (engagement-db.review, content-db.novel, gamification-db.exp_transactions).

Preconditions

- User authenticated (JWT validated at Gateway).
- Target novel exists in Content Service.
- User has not already reviewed the novel.

Postconditions

- New row inserted into review table.
- Novel's average rating and review count recalculated and persisted.
- EXP transaction inserted (exp_transactions).
- Optional level-up event published.
- Optional review count counter incremented in Redis.

Main Sequence

1. Frontend: POST /api/v1/reviews.
2. API Gateway routes to ReviewController.createReview().
3. ReviewService.createReview():
    - ContentServiceClient.getNovelById(novelId) (validate novel exists).

    - ReviewMapper.selectByUserAndNovel(userId, novelId) (ensure uniqueness).
    - ReviewMapper.insertSelective(review) (persist review).
    - updateNovelRatingAndCount(novelId) (aggregates all reviews).
    - contentServiceClient.updateNovelRatingAndCount(novelId, avg, count) (apply stats).
    - KafkaEventProducerService.publishReviewCreatedEvent(...).
4. (Synchronous reward path or asynchronous consumer) Gamification:
    - GamificationService.rewardReview(userId, reviewId)
    - ExpTransactionMapper.insert(expTransaction)
    - GamificationService.checkLevelUpAndPublishEvent(userId, earnedExp)
5. Response returned to client with ReviewResponseDTO.

Core Classes & Key Methods

Layer	Class	Representative Methods / Notes
Controller	ReviewController	createReview(), getReview(), updateReview(), likeReview(), unlikeReview()
Service	ReviewService	createReview(), updateReview(), deleteReview(), toggleLike(), updateNovelRatingAndCount()
Data Access	ReviewMapper (MyBatis) + ReviewMapper.xml	insertSelective(), selectByUserAndNovel(), selectByNovelId(), updateLikeCount()
Event Producer	KafkaEventProducerService	publishReviewCreatedEvent()
External Clients	ContentServiceClient, UserServiceClient	getNovelById(), updateNovelRatingAndCount(); getUsernameById()
Gamification	GamificationService	rewardReview(), checkLevelUpAndPublishEvent()
Gamification Persistence	ExpTransactionMapper + ExpTransactionMapper.xml	insert(), sumAmountByUserId()

Data Model Touchpoints

- engagement-db.review: (id, uuid, user_id, novel_id, rating, title, content, like_cnt, is_spoiler, create_time, update_time).
- content-db.novel: updates avg_rating, review_cnt, update_time.
- gamification-db.exp_transactions: (id, user_id, amount, reason, created_at).
- Kafka topic review-event payload fields (example): { reviewId, reviewUuid, userId, novelId, rating, title, isSpoiler, createdAt }.

Transaction Boundary

- Review insertion + rating recalculation + novel stats update are logically separated (Engagement DB transaction for review + rating calculation; Content update is a remote call—no distributed transaction).
- Event publication is currently “fire-and-forget” after DB commit.
- Gamification EXP insertion runs in its own transaction when reward endpoint or event consumer executes.

Concurrency & Idempotency

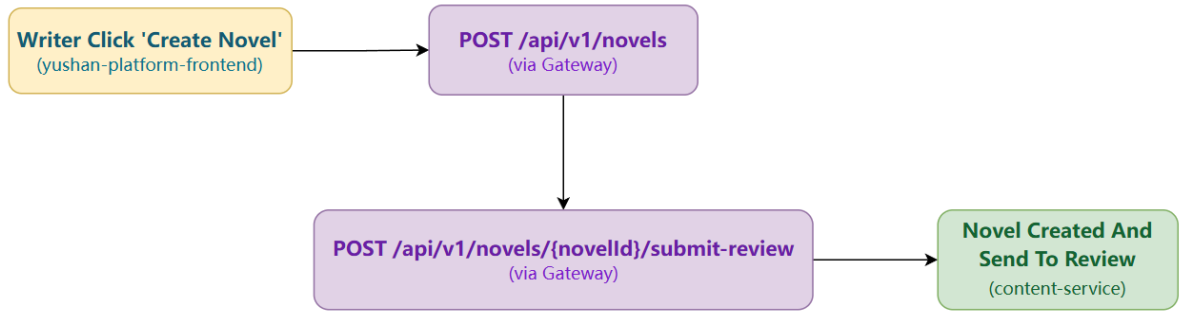
- Uniqueness enforced by selectByUserAndNovel() check (recommended: DB unique index on (user_id, novel_id) for stronger guarantee).

Error Handling

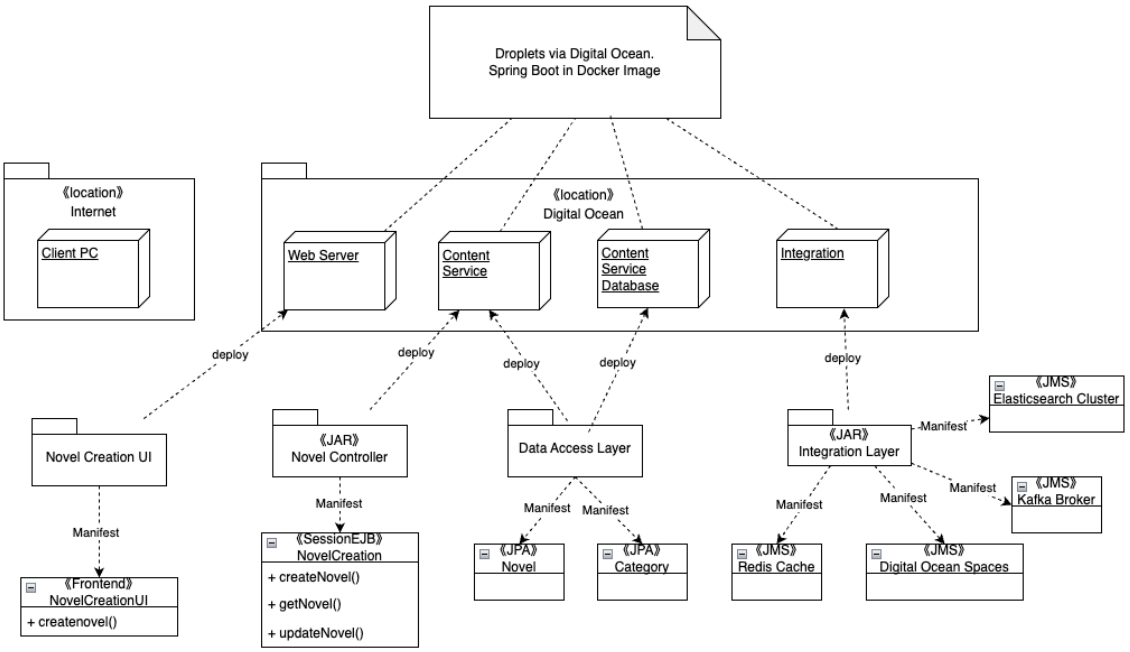
Condition	Exception
Novel missing	ResourceNotFoundException
Duplicate review	IllegalArgumentException
Remote client failure	Propagated / wrapped

3.2.4.2 Writer create a novel

Main Sequence



Detailed Deployment Elements



Intent

Allow an authenticated AUTHOR to create a novel record, initialize counters/flags, optionally auto-index to Elasticsearch, and publish a domain event for downstream services (engagement, analytics, recommendations).

Actors / Services

- Frontend via API Gateway.
- Content Service (novel lifecycle).
- Elasticsearch (indexing Chapter/Novel docs).
- Kafka (domain event: novel-event).
- Redis (optional: author stats, rate-limits).
- PostgreSQL (content-db.novel).

Preconditions

- JWT validated; role contains AUTHOR (Content service SecurityConfig enforces role-based access in endpoints/tests via JwtTestUtil token).
- Category (if provided) exists or is null/validated.
- Title uniqueness may be advisory (business rule dependent).

Postconditions

- New row in `content-db.novel` with default counters (chapter_cnt=0, view_cnt=0, review_cnt=0, avg_rating nullable/0).
- Optional index entry in Elasticsearch (author/title/synopsis for search).
- Kafka NovelCreated event published for analytics/recommendations.

Core Classes & Key Methods

Layer	Class	Representative Methods / Notes

Controller	NovelController	createNovel(), getNovel(), updateNovel(), deleteNovel()
Service	NovelService	createNovel(authorId, title, synopsis, categoryId, coverImgUrl)
Data Access	NovelMapper	insertSelective()

Data Model Touchpoints

- `Novel` entity fields (id, uuid, title, authorId, authorName, categoryId, synopsis, coverImgUrl, status, isCompleted, chapterCnt, wordCnt, avgRating, reviewCnt, viewCnt, voteCnt, yuanCnt, timestamps).

Transaction Boundary

- enforce (authorId, title) uniqueness to prevent accidental duplicates

Concurrency & Idempotency

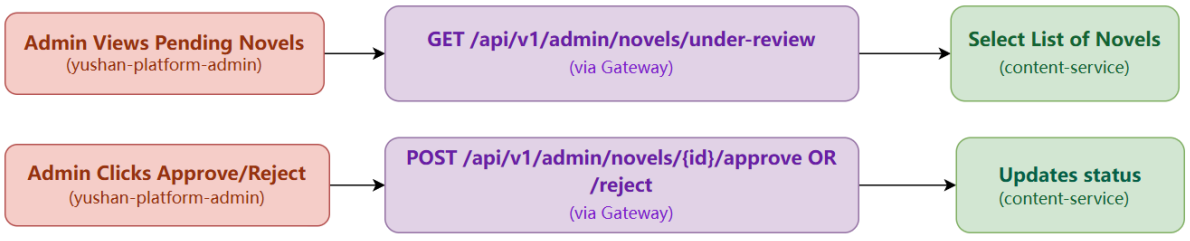
- Uniqueness enforced by selectByUserAndNovel() check (recommended: DB unique index on (user_id, novel_id) for stronger guarantee).

Error Handling

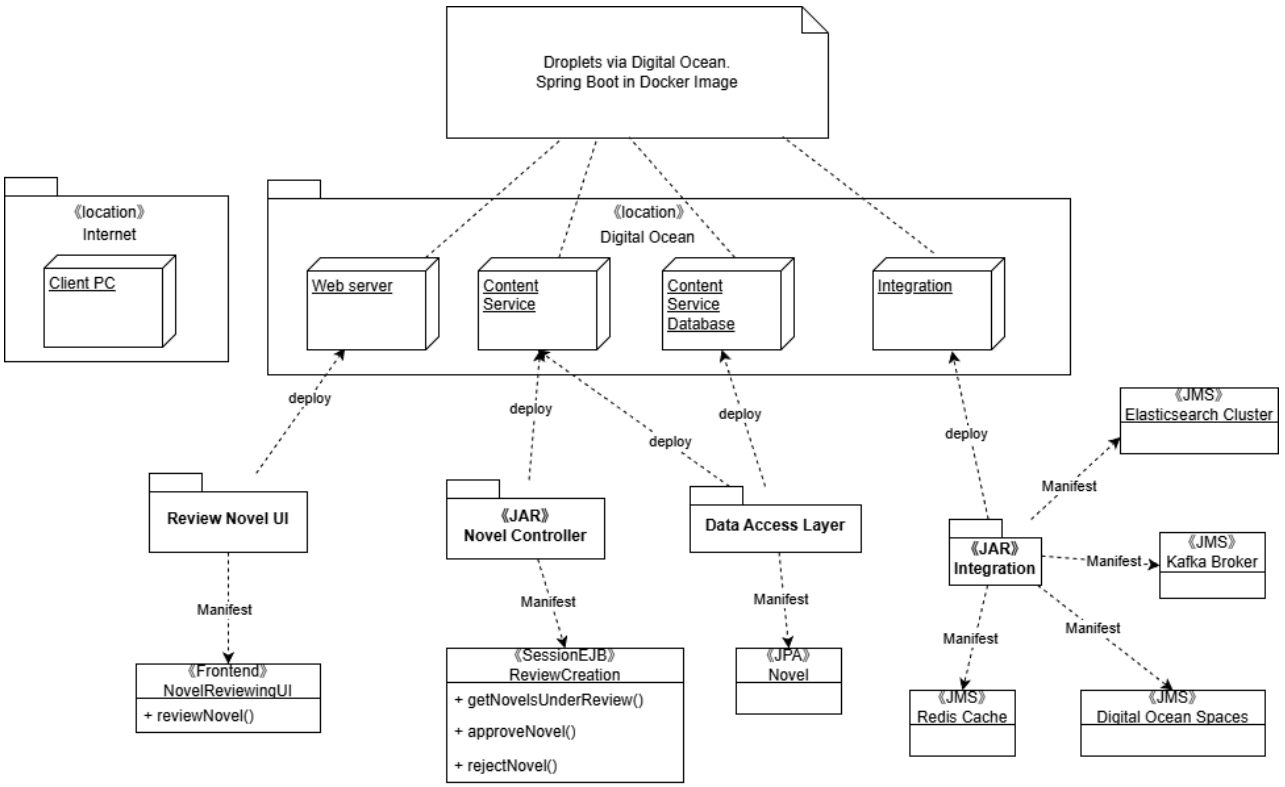
- 400: invalid payload (title empty, category invalid).
- 401/403: not authenticated/authorized.
- 500/502: ES/Kafka failures — novel stays persisted; indexer/replay handles recovery.

3.2.4.3 Admin review the under_review novels

Main Sequence



Detailed Deployment Elements



Intent

Allow admin to review and moderate novels awaiting approval (e.g., status = UNDER_REVIEW), including publishing/withholding, and optionally batch actions on chapters (publish flags).

Actors / Services

- Frontend (admin panel) via API Gateway.
- Content Service (novel status transitions; chapter publish flags).
- Engagement Service (optional: associated reports context).
- Elasticsearch (refresh/remove index on state change).
- Kafka (moderation event for audit/analytics).
- PostgreSQL (content-db.novel, content-db.chapter; engagement-db.report optional).

Preconditions

- ADMIN JWT validated at Gateway.
- Target novels exist and are currently in `UNDER_REVIEW` (or equivalent status code in `Novel.status`).

Postconditions

- Novel status updated to `APPROVED/PUBLISHED` or `REJECTED`; timestamps updated.
- For approvals, optionally bulk flip `Chapter.is_valid=true` (publish) or scheduled release windows.
- ES index updated (create or remove) to reflect visibility.
- Moderation event published for audit trail.

Core Classes & Key Methods

Layer	Class	Representative Methods / Notes
Controller	AdminNovelController	moderateNovel(id, action); batchModerateNovels(ids, action)
Service	NovelModerationService	approveNovel(id); rejectNovel(id)
Data Access	NovelMapper + ChapterMapper	updateStatus; selectByPrimaryKey; updatePublishStatusByIds; updatePublishStatusByNovelId

Data Model Touchpoints

- `Novel.status`, `Novel.updateTime`
- `Chapter.isValid`, `Chapter.publishTime`

Transaction Boundary

- Single DB transaction per moderation action (update novel + batch chapter status). ES/Kafka as post-commit actions (or Outbox).

Concurrency & Idempotency

- Idempotent moderation:
  - Ignore no-op transitions (APPROVED→APPROVED).
  - Use `updated_at` or versioning to prevent lost updates in concurrent admin actions.

Error Handling

- 404: novel not found.
- 409: invalid transition (e.g., REJECT on already PUBLISHED unless force flag).
- 502: ES failure — moderation stands; index repair job will reconcile.

3.3 Other Architectural Decisions

The key architectural decisions taken are as follows:

Identifier	Description
AD-14	**Modern Spring Cloud Stack Selection** Decision: We chose to avoid the legacy Netflix OSS stack (Zuul, Hystrix, Ribbon), which is in maintenance mode. We selected the modern, actively developed Spring Cloud stack (Gateway, Resilience4j, LoadBalancer).

	Rationale: This decision was driven by the Performance (4.1) and Maintainability (4.4) quality attributes. The non-blocking, reactive architecture of Spring Cloud Gateway provides superior throughput to Zuul 1. Using an actively developed stack like Resilience4j ensures long-term support and access to more flexible fault-tolerance patterns, directly impacting system resilience.
AD-15	**Integrated DevSecOps Pipeline** Decision: Security was not treated as an add-on but was integrated directly into the CI/CD pipeline, implementing a full DevSecOps (SAST, SCA, DAST) workflow. Rationale: This was a foundational decision to satisfy the Security (4.3) quality attribute. By automating security scanning (Checkstyle, OWASP Dependency-Check, Trivy, OWASP ZAP) on every code commit, we enforce security and quality gates before deployment, rather than discovering vulnerabilities in production.
AD-16	**Containerization & Multi-Stage Docker Builds** Decision: All microservices are containerized using Docker and built using optimized multi-stage Dockerfiles. Rationale: This directly supports Performance (4.1) by reducing image size (~70% reduction to 150-200MB) for faster deployments, and Maintainability (4.4) by ensuring a consistent, reproducible runtime environment across development, staging, and production.

3.4 Architectural Limitation

The key outstanding architectural issues in the solution are as follows:

Identifier	Issue and impact	Description	Resolution	Owner	Status
AISS–01	Complexity of Cross-Service Data Querying	The strict Database-per-Service pattern (3.1.1) makes data aggregation complex. Queries requiring data	Mitigation (Current): We use API Composition, where the client or Gateway makes multiple requests	Everyone	Closed

		from multiple services (e.g., User, Content, Engagement) must be handled at the application layer. Impact**:** High. Can lead to performance bottlenecks and complex application-layer code as the platform scales.	and stitches the data together.		
AISS –02	Runtime Coupling from Synchronous Calls	While our architecture is event-driven (3.1.4), core features (like fetching novel details for a user's library) still rely on synchronous HTTP (Feign) calls, creating runtime coupling. Impact**:** Medium. A failure in a downstream service can impact the availability of an upstream feature.	Mitigation (Current): This risk is actively managed. All Feign clients are wrapped with Resilience4j Circuit Breakers. If the Content Service fails, the User Service's library feature will fail gracefully (e.g., return cached data or an error) instead of cascading the failure. This is sufficient for the current scale.	Everyone	Closed
AISS -03	Non-Cloud-Native Service Orchestration	We selected **Netflix Eureka** for service discovery due to its simplicity.	Roadmap (Future)**:** A migration from our current Docker-based	Everyone	Open

		This is a standalone component in maintenance mode and only handles discovery. We lack a unified, cloud-native platform for container orchestration, auto-scaling, and self-healing. Impact**:** Low (Current), High (Long-term). Poses a future maintenance risk and limits scaling capabilities.	deployment to Kubernetes (K8s). K8s would natively handle service discovery (replacing Eureka), container management, and auto-scaling, aligning the platform with industry-standard cloud-native orchestration.		
AISS -04	Missing Distributed Tracing	The current observability layer (3.1.2) lacks distributed tracing. We cannot trace a single request's end-to-end journey (e.g., from Gateway -> User Service -> Content Service). This makes it extremely difficult to pinpoint the source of latency or errors in a complex call chain.	Roadmap (Future): Implement a full distributed tracing stack. This involves instrumenting all services with OpenTelemetry libraries to propagate trace context, and deploying a collector and backend (like Jaeger or Zipkin) to visualize the end-to-end traces.	Everyone	Open

		Impact: High. Severely impacts debuggability and performance-bottleneck analysis in a distributed environment.			

4. Quality Attributes

4.1 Performance

Our architecture applies several performance tactics to ensure the system stays fast and stable under increasing user activity. The design focuses on reducing response time, increasing throughput, and avoiding bottlenecks—aligned with the performance principles taught in Software Architecting for Scalable Systems (load distribution, caching, asynchronous processing, reducing service demand, and controlling arrival rate).

1. Service-Level Performance Strategies

Each microservice is optimized based on its workload pattern:

User Service (authentication & profile)

- Uses indexed queries and selective data retrieval to reduce service time.
- Redis caches short-lived values such as OTPs and user activity state, preventing repeated DB calls.
- These techniques lower both response time and DB utilization.

Content Service (novels & chapters – read-heavy)

- Search and browsing rely on Elasticsearch so that full-text queries avoid scanning relational tables.
- Page size is capped to control arrival rate and prevent oversized queries.
- Frequently requested metadata (e.g., featured novels, chapter lists) is cached.
- Image-related operations use lightweight async processing to avoid blocking requests.

Engagement Service (likes, comments, views)

- Write-heavy paths record events asynchronously via Kafka instead of processing everything in-request.
- Atomic DB updates avoid row-lock conflicts.
- Query endpoints use filtered and paginated access to keep service time predictable.

Gamification Service (points, rewards, leaderboards)

- Batch writes and idempotent processing ensure updates remain fast even under spikes.
- Leaderboards rely on Redis sorted sets for O(log n) updates and O(k) reads.

- Frequently accessed gamification summaries are memoized.

Analytics Service (history & ranking)

- Ingestion endpoints are intentionally lightweight to keep response time small.
- Rankings are precomputed periodically and stored in Redis, so user-facing ranking APIs return in constant time.

Benefit: By keeping each microservice optimized for its workload characteristics, the system avoids single-service bottlenecks and increases overall throughput.

2. Load Balancing & Traffic Distribution

- The API Gateway sits behind a cloud load balancer, distributing traffic across multiple gateway instances.
- This improves throughput by adding more workers (parallel queues), and improves response time by avoiding overloaded nodes (as per Queuing Theory).
- Gateway-level circuit breakers and retry policies protect downstream services from cascading failures.

3. Multi-Layer Caching

Caching is one of the strongest performance tactics applied in the platform:

Backend caching

- Redis is used across services to store:
    - Novel/chapter metadata
    - User library state
    - Trending lists
    - Engagement counters
    - Leaderboard slices
    - Precomputed analytics snapshots
- This reduces service demand (Si) on the databases, thereby lowering utilization (U) and improving response time (R = S / (1-U)).

Frontend caching

- The web app caches reading progress, ranking snapshots and frequently accessed lists to reduce round-trip latency for end users.

Static asset caching

- Assets served via Nginx use compression and long-lived cache headers.

Future enhancement: integrate CDN edge caching to reduce transmission time (TTFB, TTLB).

4. Asynchronous & Event-Driven Processing

To control arrival rate and reduce queueing:

- Heavy or non-urgent tasks (views, comments, achievements, analytics) are pushed to Kafka topics.

- User-facing endpoints return immediately while background consumers handle the expensive work.
- Batch jobs consolidate analytics and rankings on intervals rather than executing complex queries on-demand.

This follows the “reduce load” and “pre-compute” tactics from the course, improving both throughput and perceived performance.

5. Query, Storage & Search Optimization

- Read-intensive operations use Elasticsearch, which is designed for high throughput and low-latency search workloads.
- SQL queries use filtering and pagination to avoid O(n) scans.
- Precomputation for daily/weekly rankings reduces the worst-case service time for analytics endpoints.

Future enhancement: enable search synonyms or vector search to improve accuracy without increasing latency.

6. Rate Limiting & Timeouts

- The front-end applies client-side throttling to prevent rapid repeated calls.
- Server-side rate limiting is planned at the API Gateway using Redis-based rate limiters.
- These techniques control arrival rate λ, preventing the system from exceeding its maximum throughput μ and hitting the steep response-time curve shown in the lectures.

7. Static Asset & Delivery Optimization

- The React frontend is optimized with code splitting, tree shaking, compression, and hashed bundles.
- Nginx applies gzip/brotli compression, reducing transmission time and improving TTFB.

Future improvement: Auto-generate WebP/AVIF thumbnails to reduce bandwidth for users on slow networks.

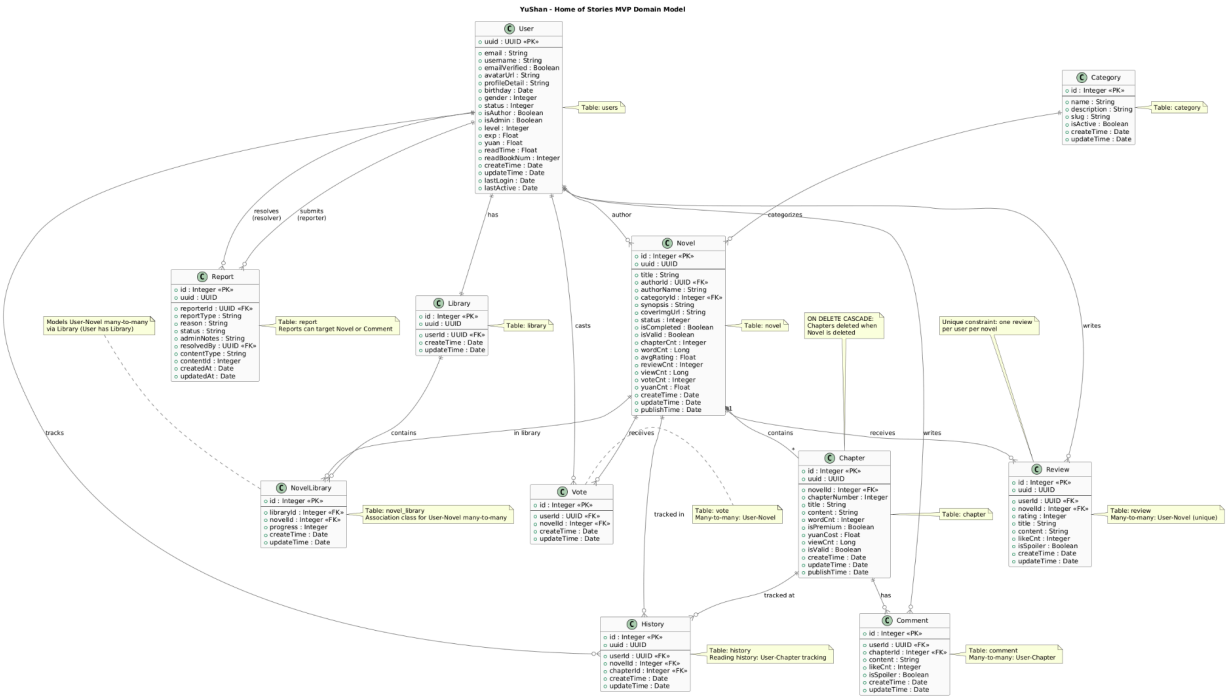
8. Benefits & Expected Outcomes

- Lower Response Time (R): caching, precomputation, and async processing reduce service time and queue time.
- Higher Throughput (X): load balancing and event-driven processing allow horizontal scaling.
- Stable Under Load: by reducing demand on bottlenecks (DBs, heavy queries), utilization stays in the “healthy” <70% zone shown in the course.
- Predictable Performance: precomputed rankings, Redis leaderboards, and Elasticsearch ensure that even high-traffic pages return consistently fast responses.

9. Future Enhancements

- Autoscaling policies (HPA) when migrating to Kubernetes.
- Centralized distributed tracing (OpenTelemetry) to pinpoint slow components.
- More fine-grained application-level caching layers.
- Distributed rate limiting and gateway-level flow control.

- Performance benchmarking and load testing to tune thread pools, DB connection pools, and Kafka partitions.



4.2 Availability

The platform incorporates multiple availability tactics to ensure the system remains accessible even when individual components fail. The design focuses on four key areas: fault detection, fault isolation, graceful degradation, and fast recovery. Together, these approaches reduce unplanned downtime and keep overall system availability high.

1. Health Checks, Fault Detection & Self-Healing

- All microservices expose liveness and readiness endpoints. These are continuously probed by the service registry and load balancer to detect unhealthy instances early. When an instance becomes slow or unresponsive, it is automatically removed from the routing pool so that traffic flows only to healthy nodes.
- Service metrics (CPU, memory, latency, error rate) are collected and monitored to trigger alerts and automated restarts when degradation is detected. This supports self-healing and reduces Mean Time to Recovery (MTTR).
- **Effect**: faults are detected quickly and isolated before they propagate.

2. Resilience Patterns: Circuit Breakers, Timeouts & Retries

- All inter-service calls are protected by timeouts, retry rules, and circuit breakers. When a downstream service becomes unstable, the circuit breaker opens to stop traffic from piling up and causing cascading failures. During this period, fallbacks return cached or degraded responses so that user-facing features still operate in partial mode.
- **Effect**: prevents chain reactions, limits the blast radius of failures, and maintains partial service instead of total outage.

3. Service Discovery & Externalized Configuration

- The platform uses dynamic service discovery so that services can register and deregister automatically. Failed nodes are evicted quickly, and newly added instances appear

immediately to the gateway. Configuration is externalized so changes (thresholds, connection URLs, tuning parameters) can be applied without requiring service restarts.
- **Effect**: enables rolling changes, quick failover, and supports horizontal scaling without downtime.

4. Redundant, IaC-Managed Infrastructure

- Infrastructure is provisioned using Infrastructure-as-Code to ensure reproducibility and fast recovery. Each microservice runs on separate nodes/VMs so failures do not overlap. A cloud load balancer distributes incoming traffic across multiple instances and offers automatic failover if one node becomes unreachable.
- Monitoring, logging, and alerting infrastructure is provisioned as a dedicated module, increasing visibility and reducing diagnostic time during incidents.
- **Effect**: minimizes single points of failure and improves reliability of the hosting environment.

5. Stateless Services & Rolling Deployments

- All application services are stateless; stateful components such as databases, Redis, object storage, and Elasticsearch operate independently. Statelessness allows instances to be replaced or scaled up without affecting in-flight user sessions. Deployments are done using rolling updates to replace instances one by one, keeping the system online during releases.
- **Effect**: no planned downtime during deployments and easy horizontal scaling.

6. Data Durability, Snapshots & Disaster Recovery

- Managed PostgreSQL, Redis, and search storage support periodic backups and snapshots. Backup strategies cover both full and incremental approaches depending on the datastore. These backups allow restoration in case of corruption or accidental data loss.
- A disaster recovery plan is defined for restoring critical data and recreating infrastructure from code.
- **Effect**: preserves data availability and reduces recovery time during critical failures.

7. Load Shedding, Rate Limiting & Backpressure

- To protect the system during traffic spikes, client-side throttling is applied, and the gateway can enforce server-side rate limits. If the platform becomes overloaded, requests can be slowed or rejected to prevent total collapse. These tactics prevent overload scenarios such as retry storms, noisy neighbours, and resource exhaustion.
- **Effect**: protects critical services under extreme load and stabilizes overall system behaviour.

8. Benefits & Future Enhancements

Benefits:

- Multiple recovery and detection mechanisms ensure faults are found quickly and isolated early.
- Redundancy in compute layers, monitoring, and configuration management decreases unplanned downtime.
- Graceful degradation patterns allow partial functionality instead of full outages.
- IaC and stateless microservices significantly reduce MTTR.

Future enhancements:

- Multi-region failover and geo-redundant databases.
- Blue/green deployments with automated smoke tests.
- Chaos engineering experiments to verify resilience under fault injection.
- Expanding alerting with availability SLOs and incident response runbooks.

4.3 Security

The system incorporates multiple layers of security controls to protect confidentiality, integrity, and availability of data. The design follows core security principles such as defence-in-depth, least privilege, proper access control (IAM), encryption, network segregation, and secure CI/CD hygiene.

1. Identity & Access Management (IAM)

- The platform uses JWT-based authentication with short-lived access tokens and refresh tokens. Passwords are hashed using BCrypt, preventing recovery even if the hash is leaked.
- Email verification and OTP flows ensure authenticity before allowing account-level operations.
- Role-based access control (RBAC) is implemented across services (admin, author, reader). Only authorized users can manage novels, chapters, comments, and admin dashboards.
- All services validate tokens consistently through shared security configurations, ensuring complete mediation—every request is checked.
- **Benefit**: Prevents spoofing, elevation of privilege, and unauthorized access.
- **Future improvement**: Introduce fine-grained permissions or claims, support MFA/WebAuthn, and refresh-token rotation.

2. Transport Security & Encryption

- All public traffic goes through HTTPS with TLS 1.2+, enforced by Nginx and the load balancer. This protects confidentiality and integrity of data in transit.
- Internal services communicate inside a private VPC to reduce exposure. Sensitive configuration (JWT secrets, DB credentials) is externalized and loaded over secure channels.
- Hashing, UUID-based IDs, and sanitized events on Kafka ensure no sensitive data is exposed across trust boundaries.
- **Benefit**: Protects against eavesdropping, tampering, and replay attacks.
- **Future improvement**: Mutual TLS between microservices or service mesh adoption.

3. Secure CI/CD Pipeline (Preventive & Detective Controls)

- Each backend pipeline includes security gates:
    - Unit and integration testing to prevent logic flaws.
    - Static application security testing (SAST) via SonarCloud, SpotBugs, Checkstyle.
    - Software composition analysis (SCA) through OWASP Dependency-Check, OSS Index, and Snyk.
    - Container image scanning with Trivy before deployment.
    - Strict workflow permissions and concurrency rules ensure no untrusted code or dependency can enter the system.
- **Benefit**: Many vulnerabilities are detected early before reaching production.
- **Future improvement**: Add Terraform security scanning, secret scanning, and signed container images (cosign) to ensure supply-chain integrity.

4. Access Control Enforcement & Least Privilege

- Backend services run as stateless processes with the minimum required permissions to DB, Redis, S3, and Elasticsearch.
- Admin frontend performs client-side permission checks (e.g., only admin can access management pages).
- Kafka and object storage access keys are restricted, scoped tightly per microservice.
- **Benefit**: Limits blast radius if a key or component is compromised.

5. Data Protection & Privacy

- Sensitive data (emails, passwords, profile details) is stored only in appropriate services. Cross-service messages use minimal, non-PII payloads.
- Redis and PostgreSQL handle state and are configured through central configuration, avoiding accidental exposure.

- Backup strategies (snapshots for PostgreSQL, Redis, and Elasticsearch) support recovery with minimal data loss.
- **Benefit**: Reduces risk of data leakage and supports business continuity.

6. Network Segregation & Isolation

- Services are deployed inside a private network, reachable only through the API Gateway. Direct access from the Internet is blocked by cloud firewalls.
- A clear separation exists between:
    - public zone (gateway, Nginx)
    - application services (user/content/engagement/gamification/analytics)
    - data layer (PostgreSQL, Redis, Elasticsearch)
    - monitoring stack (Grafana, Prometheus, Filebeat, Logstash)
- **Benefit**: A compromised frontend cannot directly reach internal components. Reduces lateral movement in attacks.
- **Future improvement:** Add an out-of-band network for logs or sensitive administrative access.

7. Monitoring, Detection & Incident Handling

- Prometheus, Grafana, Alertmanager, Filebeat, and Logstash provide continuous monitoring of logs, metrics, and anomalies.
- Alert rules detect spikes in failures, latency, or resource usage—early indicators of potential attack patterns (e.g., brute force, DDoS-like spikes).
- Circuit breakers and retries also act as resiliency-focused security controls, preventing cascading failures.
- Future improvement: Add SIEM integration, anomaly detection, and IDS/IPS to detect intrusion attempts at the network and host levels.

8. Disaster Recovery & Backups

- Automated snapshots for PostgreSQL, Elasticsearch, and Redis provide protection against data corruption or tampering.
- Terraform-managed infrastructure enables fast recreation of environments from versioned definitions.
- Golden images and reproducible container builds allow quick deployment during an incident.
- **Future improvement**: Define clear RPO/RTO targets and automate disaster-recovery testing.

9. Benefits & Future Enhancements

Benefits:

- Multi-layered controls protect CIA requirements while keeping the system usable.
- Secure-by-design principles such as least privilege, complete mediation, and defence-in-depth reduce risks.
- CI/CD gates, encrypted communications, network segregation, and active monitoring provide strong preventive and detective capabilities.

Future enhancements:

- Introduce zero-trust networking for internal services.
- Automate key/secret rotation.
- Add IDS/IPS, SIEM, and periodic penetration testing.
- Strengthen end-to-end encryption for sensitive operations.

4.4 Extensibility and Maintainability

The platform is designed to evolve easily as business requirements expand. The architecture separates concerns clearly, reduces coupling, and applies practices that simplify changes at both

code and infrastructure level. This ensures the system remains maintainable and can grow without major redesign.

1. Modular Microservice Architecture

- The solution is divided into five functional microservices (user, content, engagement, gamification, analytics). Each owns its domain logic, data, and API contract, while supporting platform services (gateway, config service, service registry) provide routing, configuration, and discovery.
- Each microservice exposes its own Swagger/OpenAPI documentation endpoint, so its APIs are self-describing and easily discoverable by other teams or new services.
- **Benefit**: Changes in one domain do not affect others, enabling independent evolution and deployment, and the Swagger pages make API understanding and extension straightforward.
- **Future improvement:** Aggregate these Swagger definitions into a central API portal with explicit versioning and deprecation policies to better support external consumers and long-term evolution.

2. Shared Platform & Standardized Patterns

- Common platform components handle infrastructure concerns:
    - Config service centralizes environment- and domain-specific parameters, reducing duplicated configuration across codebases.
    - API gateway enables adding new services through simple routing rules instead of touching multiple clients.
    - Service registry automatically registers and updates service instances when scaling up or down.
- These elements reduce configuration drift and help maintain uniform behavior across all services.
- **Future improvement**: Provide standardized service templates and scaffolding to bootstrap new microservices with consistent structure and best practices built in.

3. Event-Driven Extensibility

- The ecosystem supports adding new features without modifying existing services:
    - Services emit domain events asynchronously to Kafka.
    - New services (e.g., recommendation engine, notification service) can subscribe to existing topics and react independently.
    - A unified event envelope standardizes message format, improving compatibility and reducing coupling.
- **Benefit**: New capabilities can be plugged in without changing producers, making the system highly extensible.
- **Future improvement:** Introduce a schema registry to enforce backward-compatible event evolution and provide a documented event catalog.

4. Maintainable Codebase

- **Backend**
    - Code is layered (controller → service → repository/mapper) to contain change and increase cohesion.
    - SQL is isolated behind dedicated mappers, reducing impact when schemas evolve.
    - Automated tests (unit and integration) provide safety nets, enabling refactoring and feature additions with confidence.
    - Static analysis, style checks, and coverage thresholds detect code smells and regressions early.
- **Frontend**
    - Feature-based folder structure keeps related components, hooks, and services together.
    - Shared utilities and typed API clients improve reuse and reduce duplication.

    - Component and service tests help catch regressions in UI and client logic.
- **Benefit**: Clear structure and uniform coding practices reduce onboarding time, make changes safer, and lower the risk of side effects.
- **Future improvement**: Add architectural decision records (ADRs) and generate API clients directly from OpenAPI definitions to keep frontends and backends in sync.

5. CI/CD & Release Management

- Automation pipelines enforce consistent build, test, and release processes:
    - Standard workflows build, test, scan, and package every service the same way.
    - Artifacts (Docker images, digests, reports) provide traceability and reproducibility across environments.
    - Concurrency controls avoid overlapping runs and inconsistent deployments.
- **Benefit**: Changes are delivered in a repeatable way, making it easier to maintain and evolve the platform over time.
- **Future improvement:** Introduce automated semantic versioning, changelog generation, and deployment strategies (blue–green, canary) controlled through the pipeline with manual approval gates.

6. Infrastructure as Code & Observability

- Infrastructure is defined declaratively via Terraform:
    - Compute, networking, load balancing, databases, caches, and monitoring components are all described as code.
    - Changes to infrastructure are versioned, reviewed, and repeatable across environments.
    - Observability (metrics, logs, dashboards) supports deep visibility and quick diagnosis during incidents.
- **Benefit**: Environment drift is minimized, and infrastructure changes are easier to reason about and roll back if needed.
- **Future improvement:** Integrate secret management (e.g., Vault/Secrets Manager), add drift detection, and manage more platform components (e.g., scheduled jobs, alert rules) as code.

7. Documentation & Knowledge Sharing

- A central index document lists all services, their responsibilities, and integration points.
- Architecture and design documents provide context for new contributors.
- Swagger/OpenAPI documentation for each microservice acts as live API documentation.
- **Benefit**: New team members can understand, debug, and extend the system faster, reducing dependency on tacit knowledge.
- **Future improvement**: Schedule periodic documentation reviews, link ADRs to design decisions, and adopt architecture validation tools (e.g., fitness functions, architecture tests) to ensure implementation remains aligned with the intended architecture.

8. Benefits & Future Enhancements

Benefits:

- Modular services, clear boundaries, and event-driven patterns make the system easy to extend.
- Consistent coding style, tests, and CI/CD pipelines reduce maintenance effort and risk.
- Infrastructure as Code and strong observability accelerate debugging and ensure reliable changes.
- Per-service Swagger documentation improves discoverability and reduces friction when integrating new features or clients.

Future enhancements:

- Expand automation (service scaffolding, documentation generation, deployment workflows).

- Introduce schema registry, central API portal, and ADRs for clearer evolution tracking.
- Add SLO-based dashboards to proactively detect areas needing refactoring or scaling.

5. DevOps and Development Lifecycle

5.1 Source Control Strategy

Our source control strategy ensures strong traceability, high development velocity, and secure collaboration across more than ten repositories that make up the Yushan microservice platform. The strategy covers repository layout, branching conventions, pull request workflow, artifact management, access control, and integration with CI/CD pipelines.

1. Repository Structure

We adopt a **multi-repository architecture**, where each microservice, platform service, frontend application, and infrastructure module is stored in its own dedicated GitHub repository. This aligns strictly with domain boundaries and supports independent development and deployment cycles.

Backend Microservices

- **yushan-user-service**
- **yushan-content-service**
- **yushan-engagement-service**
- **yushan-gamification-service**
- **yushan-analytics-service**

Platform Services

- **yushan-service-registry (Eureka)**
- **yushan-config-server**
- **yushan-api-gateway**

Frontend Applications

- **yushan-platform-frontend** (Reader UI, deployed to GitHub Pages)
- **yushan-platform-admin** (Admin Console, deployed to GitHub Pages)

Infrastructure & DevOps

- Terraform modules for provisioning DigitalOcean droplets and firewalls
- Monitoring stack (Prometheus, Grafana, ELK/Logstash + Filebeat)
- Deployment scripts and observability configuration

This structure provides:

- Clean separation of concerns
- Technology isolation between services
- Easier debugging and troubleshooting
- Independent ownership and contributor access
- Clear CI/CD pipelines per repository

It also matches standard microservice best practices, avoiding the scaling limitations of monolithic repositories.

Design Rationale: Why We Chose a Multi-Repository Architecture (Instead of a Mono-Repo)

Although a mono-repository could have been used to host all microservices under a single version-controlled workspace, we deliberately chose a multi-repository model for the Yushan platform. The decision is based on several practical and architectural considerations:

- **Independent evolution of services**

  Each service has its own lifecycle, backlog, and release cadence. A multi-repo setup ensures one service can be refactored or upgraded (e.g., migrate to a new Spring Boot version) without forcing coordinated changes across unrelated modules.

- **Clear team ownership & contributor boundaries**

  Different team members are responsible for different microservices. A multi-repo structure enforces clean ownership, reduces accidental changes in unrelated services, and simplifies access control (GitHub permissions per repo).

- **Service-specific CI/CD pipelines**

  Each repository contains its own GitHub Actions pipeline optimized for that technology stack:

  - Spring Boot microservices → Maven, JaCoCo, SonarCloud, OWASP, Trivy
  - React frontends → Jest, ESLint, Snyk, ZAP, GitHub Pages deploy

  A mono-repo pipeline would be significantly more complex, slower, and harder to maintain.

- **Reduced coupling and faster feedback loops**

  Developers only clone and run the service they are working on. CI runs only for changed repos instead of the entire system, improving development speed and lowering infrastructure costs.

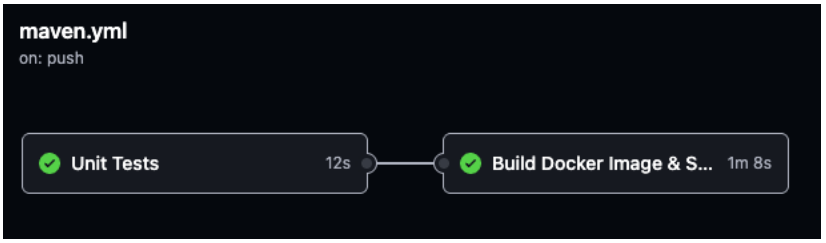
- **Avoids cross-service interference**

  Microservices are isolated by design. Separate repositories prevent:

  - unintended dependency sharing
  - uncoordinated code modifications
  - merge conflicts across teams
- **Better scalability for future services**

  New microservices (e.g., recommendation engine, reporting service) can be added as new repositories without impacting existing folder structure or CI pipelines.

Overall, while a mono-repo could simplify code visibility, the multi-repo approach provides superior modularity, maintainability, scalability, and clearer separation of responsibilities, which aligns better with the architectural and DevOps goals of a microservice platform.



Server Register CI/CD Pipeline

Content Service CI/CD Pipeline

2. Branching Model

We follow a **JIRA-driven, task-based branching strategy**, ensuring full traceability from requirement to code change.

Branch Naming Rule

Every development task in JIRA generates a corresponding branch:

YM-150-create-novel-status-bug

YM-221-add-leaderboard-endpoint

YM-310-refactor-kafka-event-envelope

Format: <JIRA-ID>-<short-description>

Branching Guidelines

- Work is always branched from the **main** branch.
- **No direct commits to main**.
- Every change must be merged through Pull Request (PR).
- Branches are deleted after the PR is merged.

Why this model?

- Perfect mapping between business requirements → code → deployment.
- Zero ambiguity when auditing changes.
- Reduced conflict and better parallel development.
- Improved alignment with Sprint workflow.

3. Pull Request & Code Review Workflow

Every contribution must go through a PR and pass strict automated checks.

Pull Request Rules

- PR title **must begin with the JIRA key**
  Example: *[YM-203] Add daily ranking API*
- Description must include:
  - Link to JIRA task
  - Summary of changes
- At least **one reviewer** must approve before merge.
- All GitHub Actions pipelines must pass:
  - Unit tests

    - Integration tests
    - Static analysis (SpotBugs, Checkstyle, ESLint)
    - SAST/SCA scans (SonarCloud, Snyk, Dependency-Check)
    - Code style & formatting
    - Docker image build
    - Trivy scan
    - ZAP DAST baseline (frontend / content service)

Merge Strategy

- "Squash and merge" to keep history clean and readable.
- Commit message for squash includes JIRA key automatically.

This enforces quality and ensures every commit on main is production-ready.

```yaml
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
    types: [opened, synchronize, reopened]
```

4. Commit Standards

To maintain a clean audit trail:

- Every commit starts with the JIRA key: *YM-118: Refactor ChapterDTO mapper*
- Commit content must be atomic (single purpose).
- No sensitive data (secrets, credentials, tokens) is ever committed.

This matches enterprise-grade governance practices.

5. Artifact Management

Each repository produces multiple artifacts during CI:

a) Docker Images – GitHub Container Registry

All microservices build Docker images and publish them to:

ghcr.io/maugus0/yushan-<service>

Images are:

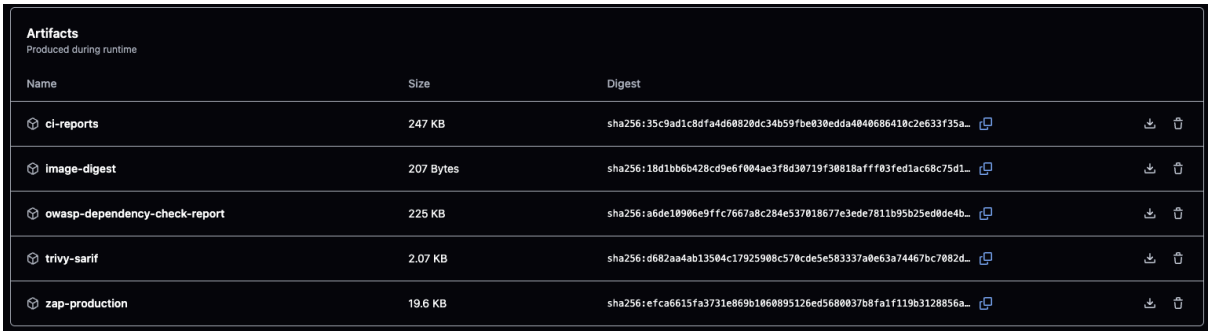
- Multi-platform (linux/amd64, linux/arm64)
- Tagging scheme:
    - <sha>
    - <branch>
    - latest
    - staging
    - release tags (v0.3.1)

b) CI/CD Artifact Storage

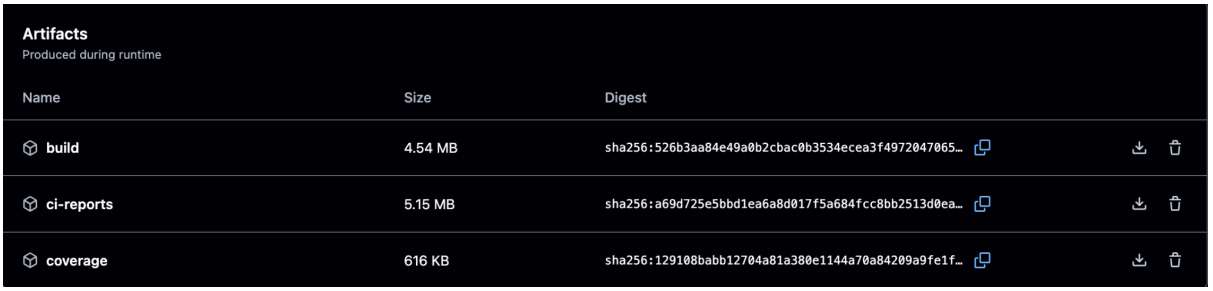
CI runs generate the following artifacts for auditing:

- SpotBugs SARIF
- Checkstyle Report
- JaCoCo Coverage
- SonarCloud Quality Gate summary
- OWASP Dependency-Check HTML + XML + SARIF
- Snyk SARIF
- Trivy container vulnerability SARIF
- ZAP DAST reports (HTML, JSON, MD)
- Build artifact bundles (frontend)
- Docker image digest files

These artifacts ensure traceability and compliance with DevSecOps requirements.



Backend CI/CD artifacts



Frontend CI/CD artifacts

6. CI/CD Integration

Every repository contains a dedicated GitHub Actions pipeline.

Backend pipeline includes:

- Unit tests
- Integration tests
- Lint & static analysis
- Security gates (SCA, SAST)
- Build Docker image
- Trivy container scan
- Deploy to DigitalOcean (only on main)

- ZAP DAST baseline
- Report aggregation & artifacts generation

Frontend pipeline includes:

- ESLint, Prettier, SonarCloud
- Snyk OSS
- Build verification
- Docker build
- Trivy scan
- Deploy to GitHub Pages
- Playwright smoke test
- Auto rollback on failure

This ensures all services are always safe, tested, secure, and reproducible.

7. Access Control & Security

Security is enforced at multiple levels:

Developer Authentication

- GitHub SSO
- Mandatory 2FA for all team members

Role-Based Permissions

- Maintainers → write/admin
- Service owners → maintain per-service repo
- Contributors → read + PR
- DevOps → full access for infra repos

Secret Management

- All credentials stored in GitHub Encrypted Secrets:
    - DB password
    - JWT secret
    - DigitalOcean SSH key
    - Sonar token
    - Snyk token
    - NVD key
    - OSS Index token
    - Logstash endpoint
- No secrets are stored in source code

This satisfies DevSecOps governance requirements.

8. Summary

The source control strategy supports:

- **Independent service ownership** through multi-repo isolation
- **High development velocity** via JIRA-based branching
- **Strict code quality & security** enforced through CI/CD
- **Full traceability** from requirement → code → artifact → production

- **Strong access control & secret management**
- **Clean release cycles and reproducible builds**

Together, these practices create a **scalable, secure, and maintainable DevOps foundation** for the Yushan microservice platform.

5.2 Continuous Integration

Our project uses GitHub Actions as the central CI platform for all repositories (backend microservices, platform services, and frontends). The goal is to automatically build, test, and scan every change before it reaches the main branch or production, following the DevSecOps practices taught in the course.

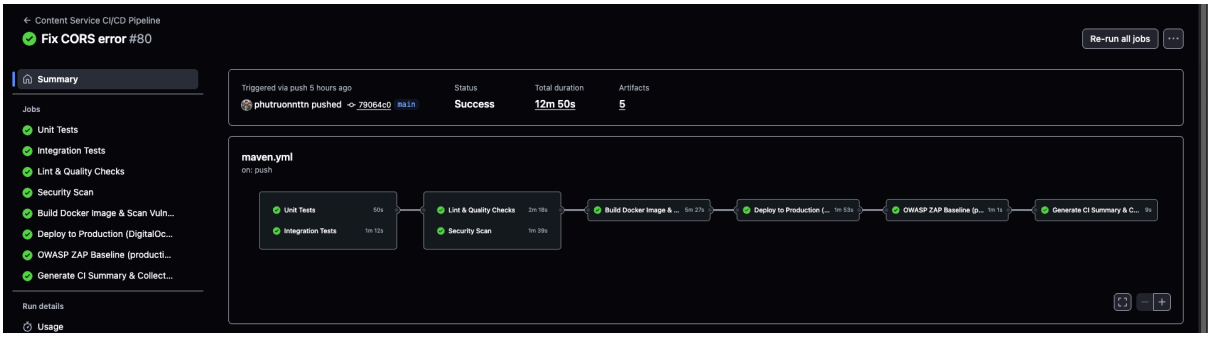
5.2.1 Triggers and Global Pipeline Design

- **Triggers**: Every repository has a CI workflow that runs on
  - push to main – keeps the production branch always in a releasable state.
  - pull_request targeting main (opened / synchronize / reopened) – blocks merge until all checks pass.
- **Concurrency and Safety**
  - We enable concurrency per branch (workflow-ref) so that only one pipeline for the same branch runs at a time. New pushes cancel outdated runs to avoid race conditions.
- **Permissions**
  - Workflows request only the minimum GitHub permissions needed (read for code, write only for checks, security-events and packages when uploading SARIF or Docker images).
- **Caching and Speed**
  - Maven dependencies (.m2/repository) and Node modules (npm cache) are cached between runs to shorten build time.
- **Artifacts and Reports**
  - All important reports (coverage, static analysis, SCA, DAST, Trivy SARIF, ZAP reports, CI summary) are uploaded as artifacts. This creates an audit trail and supports later security and performance analysis.

```yaml
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true  # Cancel previous runs
```

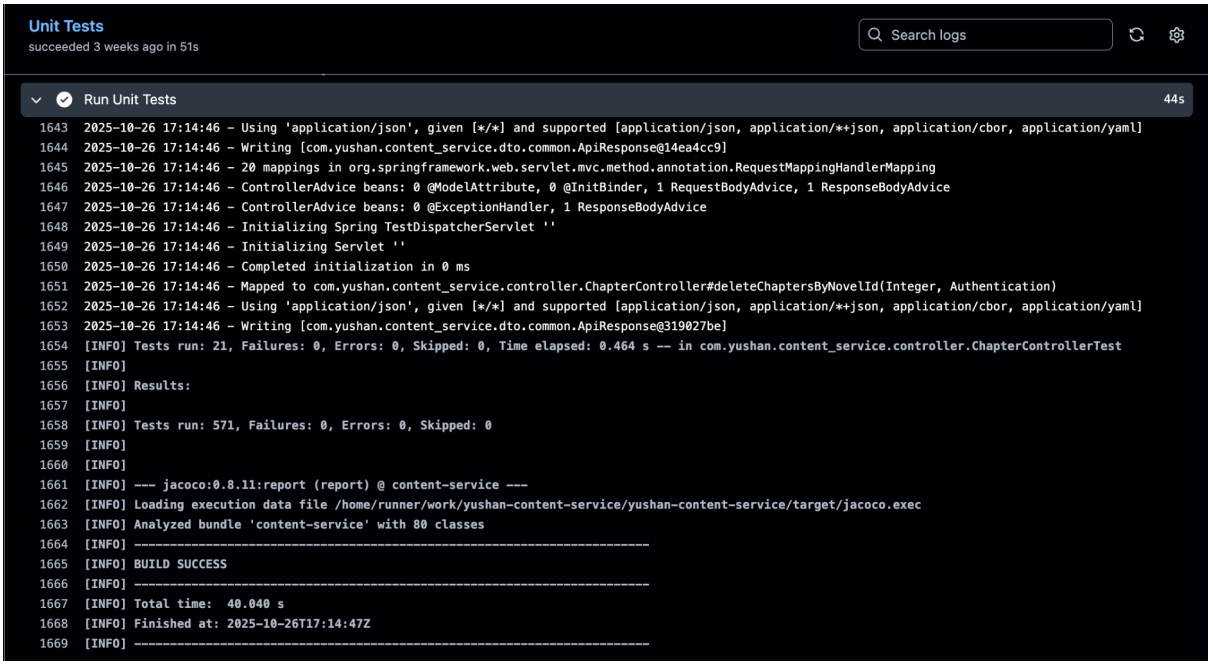
5.2.2 Backend Microservices CI Pipeline

All Spring Boot microservices share the same CI pattern. The pipeline is split into multiple jobs that run in parallel where possible.



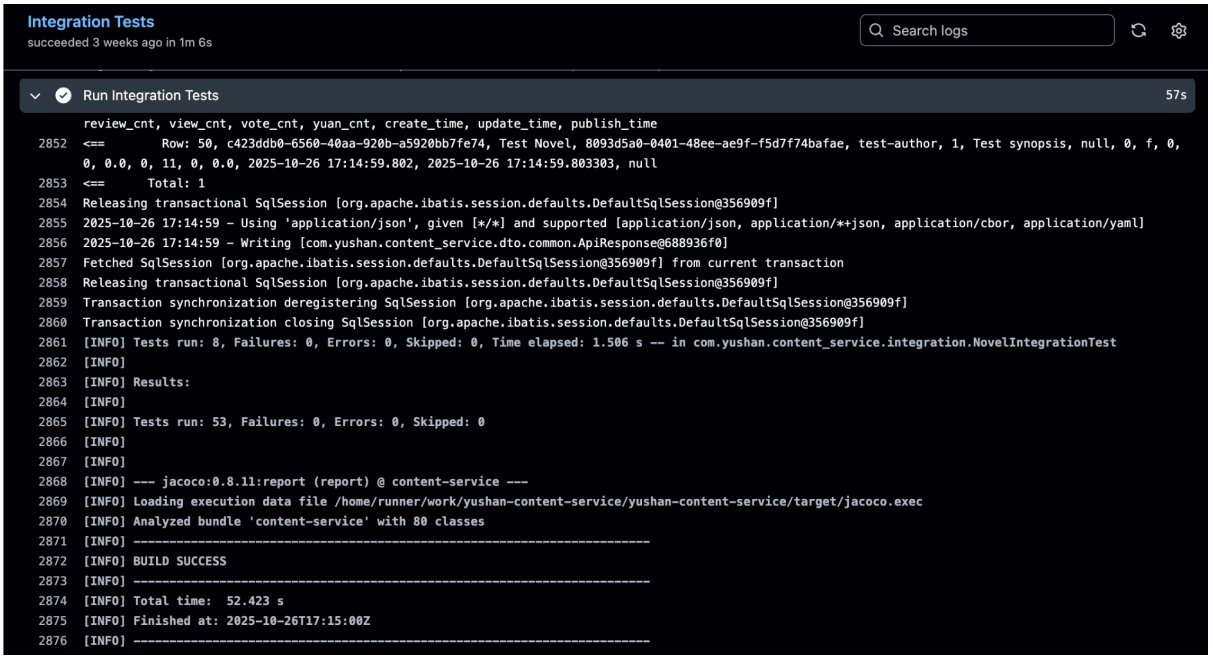
1. Unit Tests

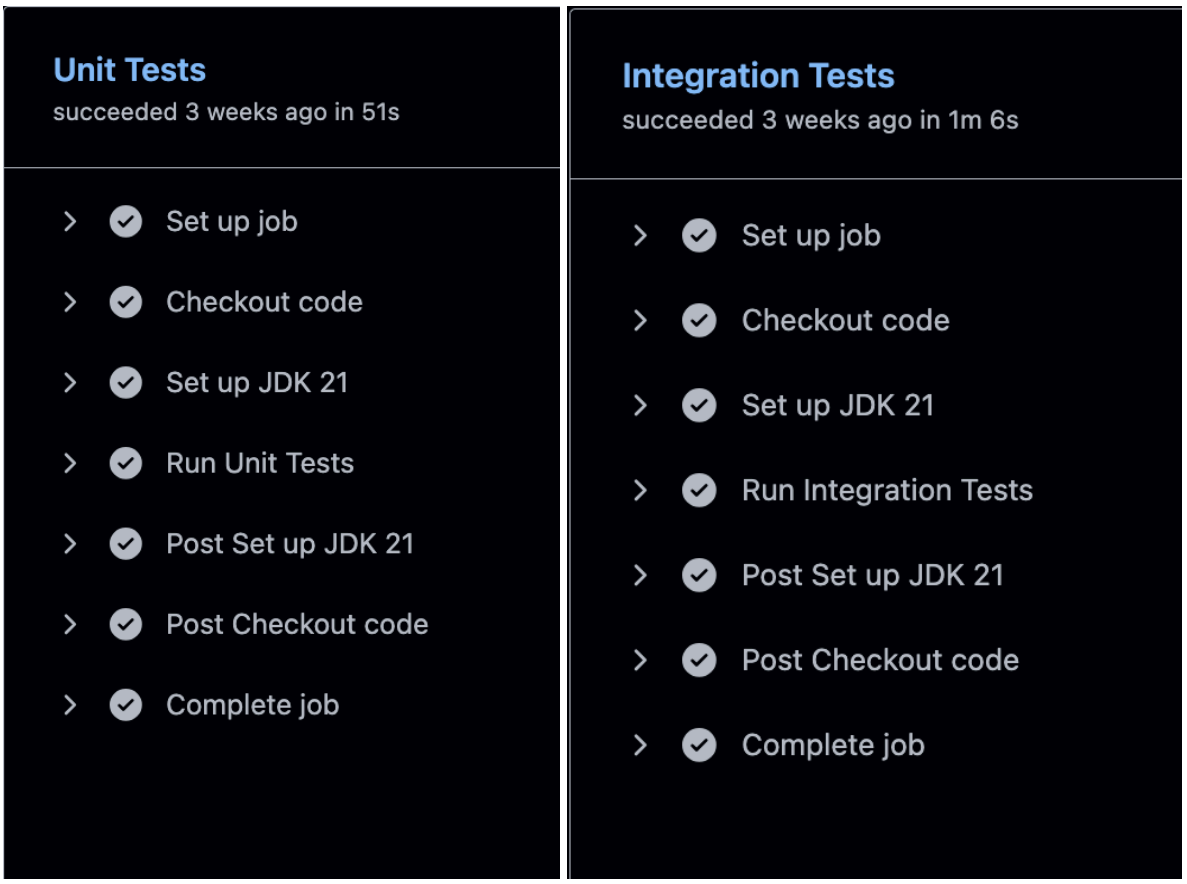
- Triggered on every push / PR.
- Runs unit tests under a dedicated test profile using JUnit and Mockito.
- Focus: test business logic in isolation (service layer, guards, validators).



2. Integration Tests

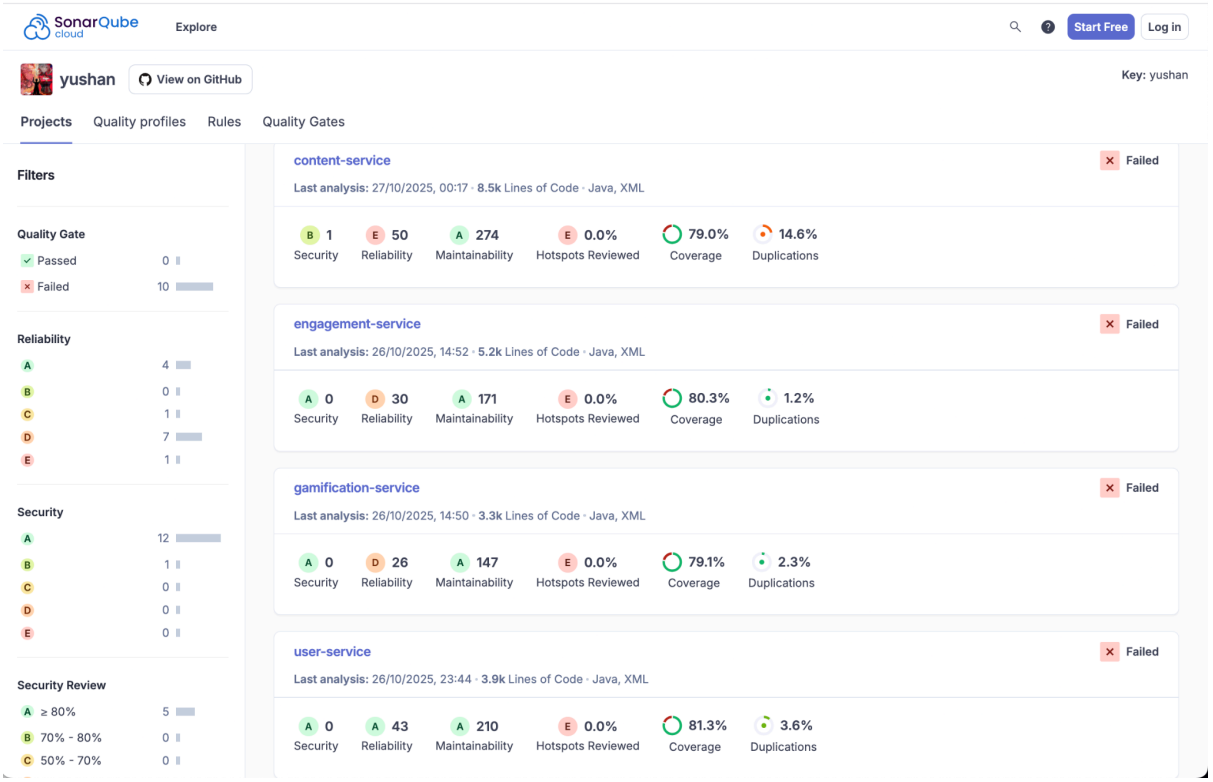
- Separate job using an integration-test profile and CI=true for deterministic behaviour.
- Focus: test the application against real infrastructure (PostgreSQL, Redis, Kafka) or in-memory equivalents, checking repository mappings, messaging behaviour and REST endpoints.





3. Lint & Quality Checks

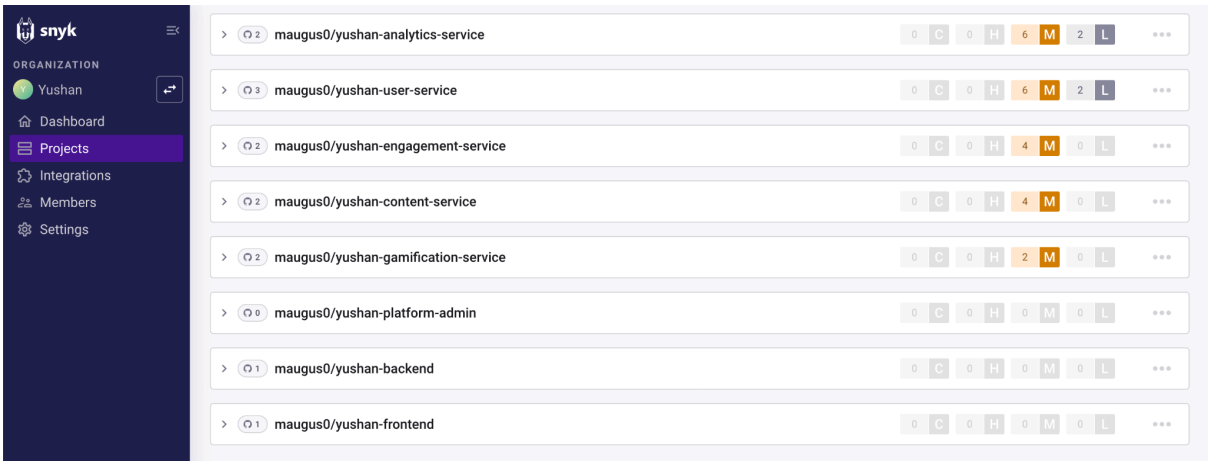
- Static analysis using SpotBugs to detect bug patterns.
- Checkstyle to enforce code style (Google Java Style).
- JaCoCo to produce coverage reports.
- SonarCloud analysis using compiled classes and the JaCoCo XML report as input.
- Quality metrics such as code smells, duplication and coverage are tracked. The target is >80% test coverage and 0 critical vulnerabilities.



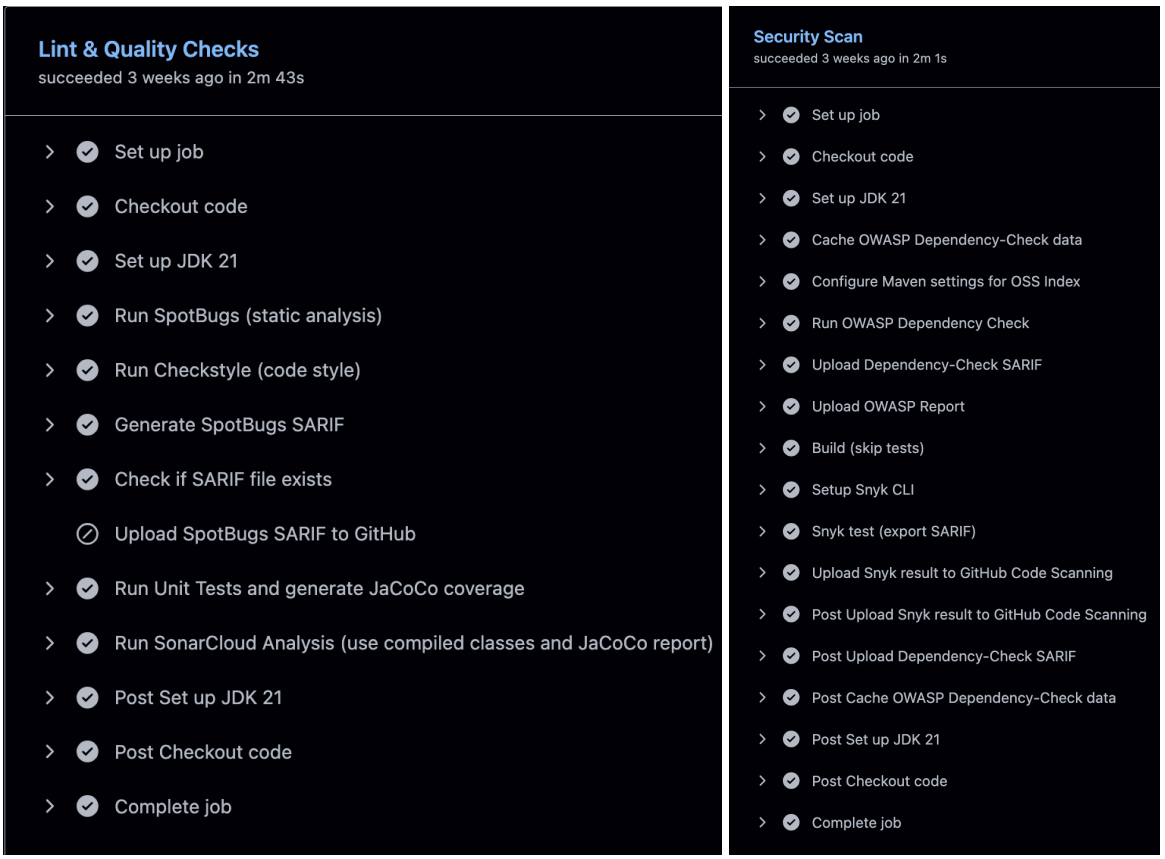
SonarQube Dashboard

4. Security Scan (SCA)

- OWASP Dependency Check integrates with NVD API and OSS Index to scan all dependencies.
- Fails the build if any vulnerability with CVSS ≥ 10 (critical) is found.
- Snyk CLI performs an additional open-source dependency scan.
- Both tools export SARIF files, which are uploaded to GitHub's Security tab and stored as artifacts together with HTML / XML reports.



Snyk Dashboard

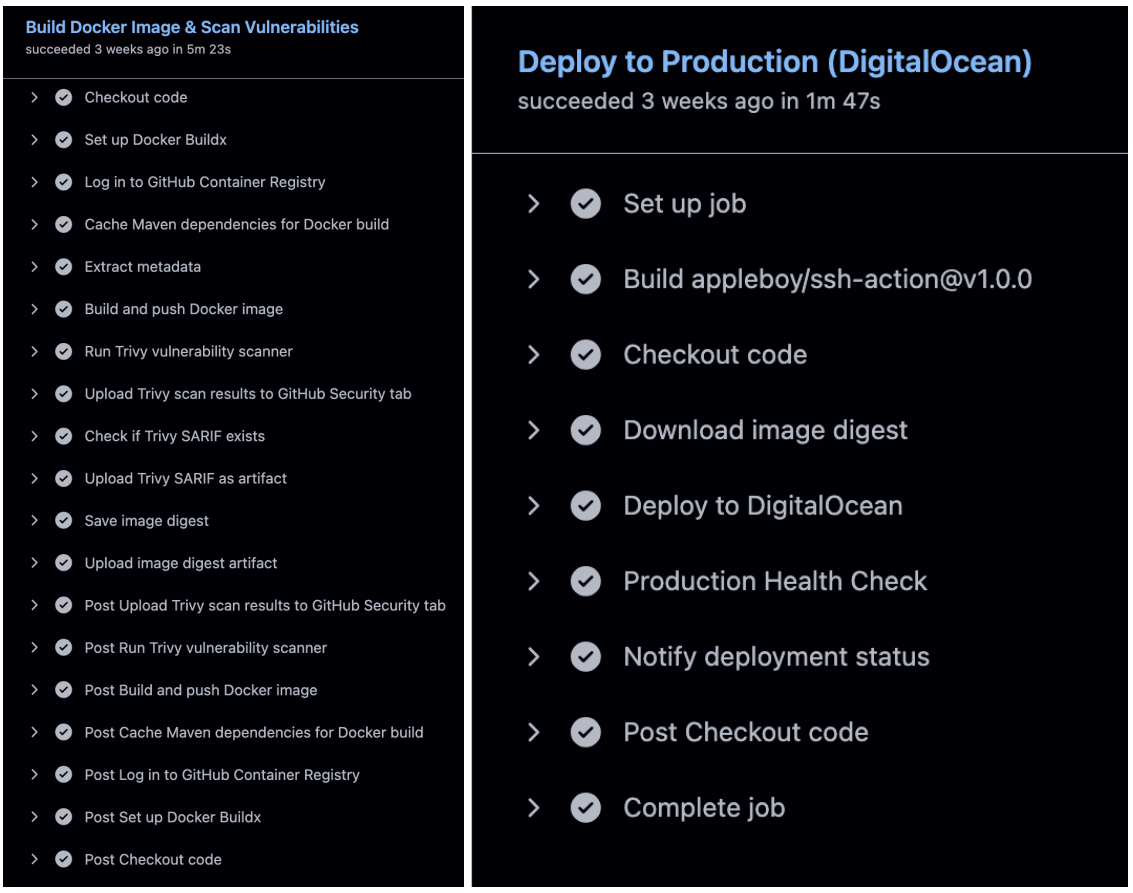


5. Build Docker Image & Scan Vulnerabilities

- Uses Docker Buildx for a multi-stage build:
    - Builder image with JDK 21.
    - Lightweight runtime image (JRE on Alpine Linux) with a non-root user and built-in health check.
- Result: image size reduced from more than 500 MB to about 150–200 MB.
- Trivy scans the final image (OS packages + application dependencies) and uploads SARIF results to GitHub Security plus an artifact.
- An image digest file is produced to guarantee that later deployments use exactly the tested image.

6. Deploy & Health Check (Continuous Delivery hook)

- For the content service, a final job runs only when the pipeline is triggered from main (or a special CI testing branch).
- The job connects to the DigitalOcean droplet via SSH, pulls the latest GHCR image, restarts the container with production environment variables, and waits for the service to start.
- A loop performs several /actuator/health checks; if the service does not become UP, the job fails and the deployment is treated as unsuccessful.

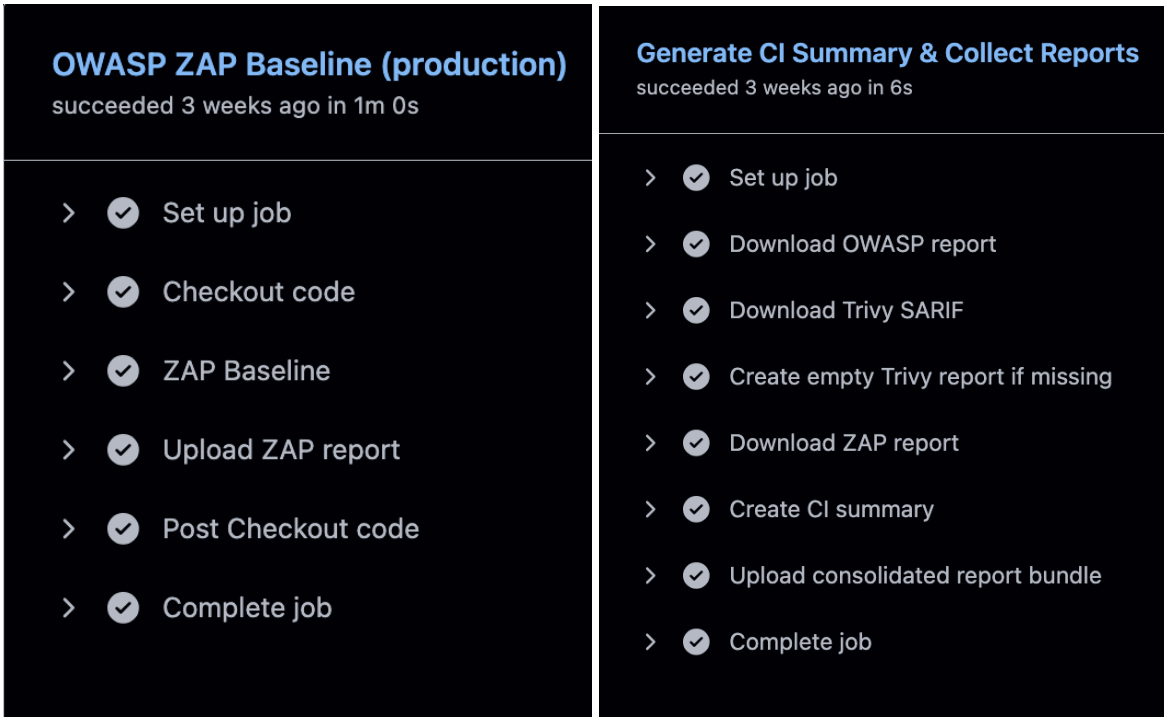


7. OWASP ZAP Baseline (DAST)

- After a successful deployment, an OWASP ZAP baseline scan is run against the public URL (https://yushan.duckdns.org).
- It checks for common web vulnerabilities such as SQL injection, XSS, authentication flaws and security misconfigurations.
- ZAP produces HTML / JSON / Markdown reports which are uploaded as artifacts.

8. Generate CI Summary & Collect Reports

- A final job downloads all previous artifacts (OWASP reports, Trivy SARIF, ZAP results).
- A consolidated summary.md is generated with links to each report.
- This bundle forms the evidence package for SAST, SCA and DAST in the security section of the project report.

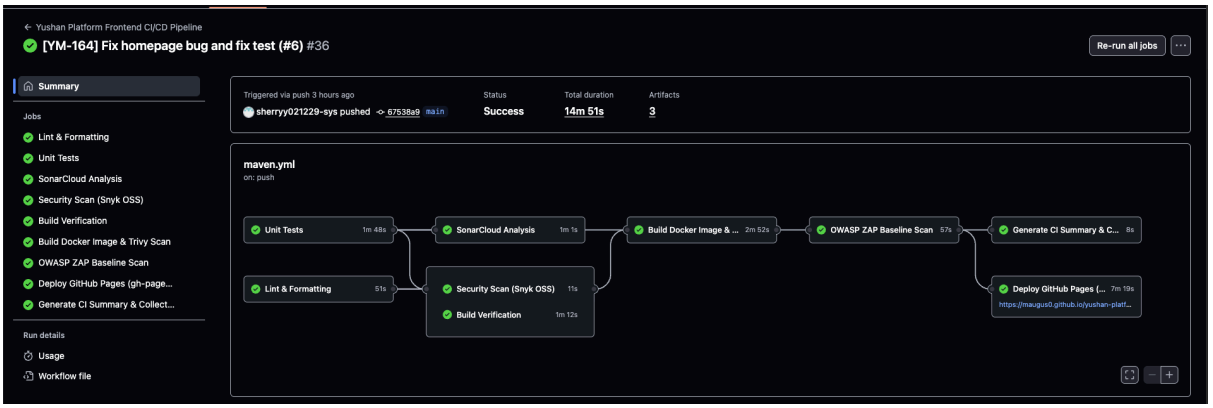


Improvements implemented over time

- Docker build optimization – switching to multi-stage Alpine images reduced size by ~70% and improved pull time.
- Maven dependency caching – using GitHub cache for .m2/repository reduced dependency download from about 5 minutes to around 1 minute.
- Test execution parallelization – running unit tests, integration tests, lint and security scans in separate jobs (instead of sequential steps) shortened total pipeline duration.
- Logging strategy – backend containers send logs through the Docker syslog driver to Logstash (UDP), then to Elasticsearch and Kibana, improving observability during and after deployments.

5.2.3 Frontend CI Pipeline

The frontend repositories (reader and admin UIs) use a Node-based CI pipeline tailored for React applications.

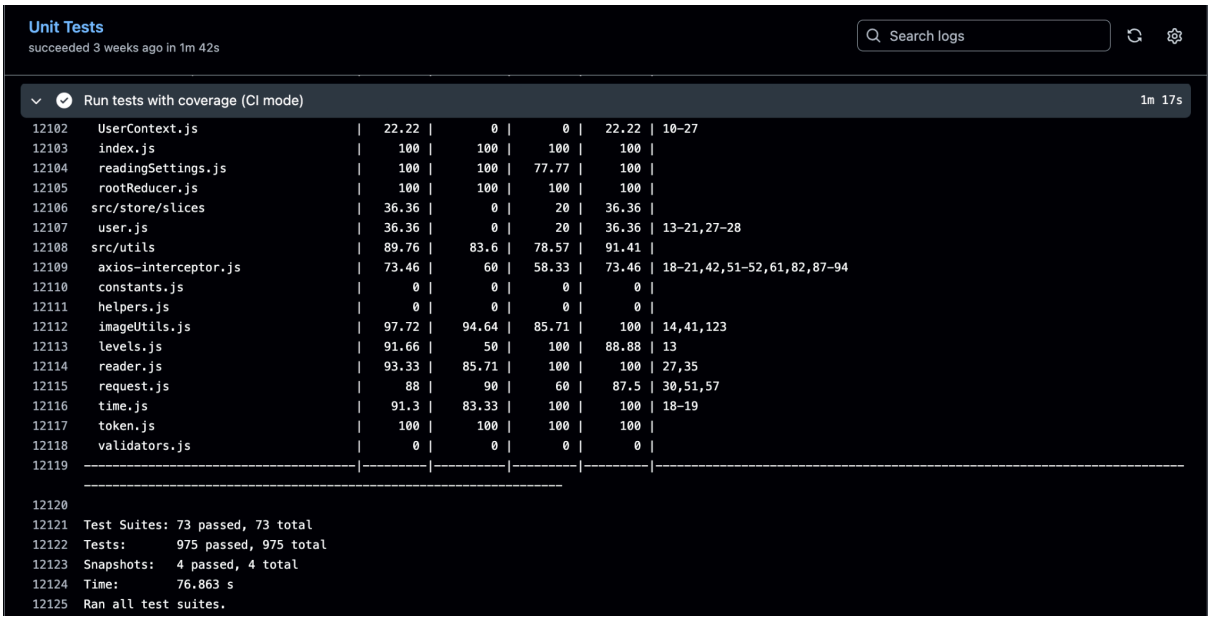


1. Lint & Formatting

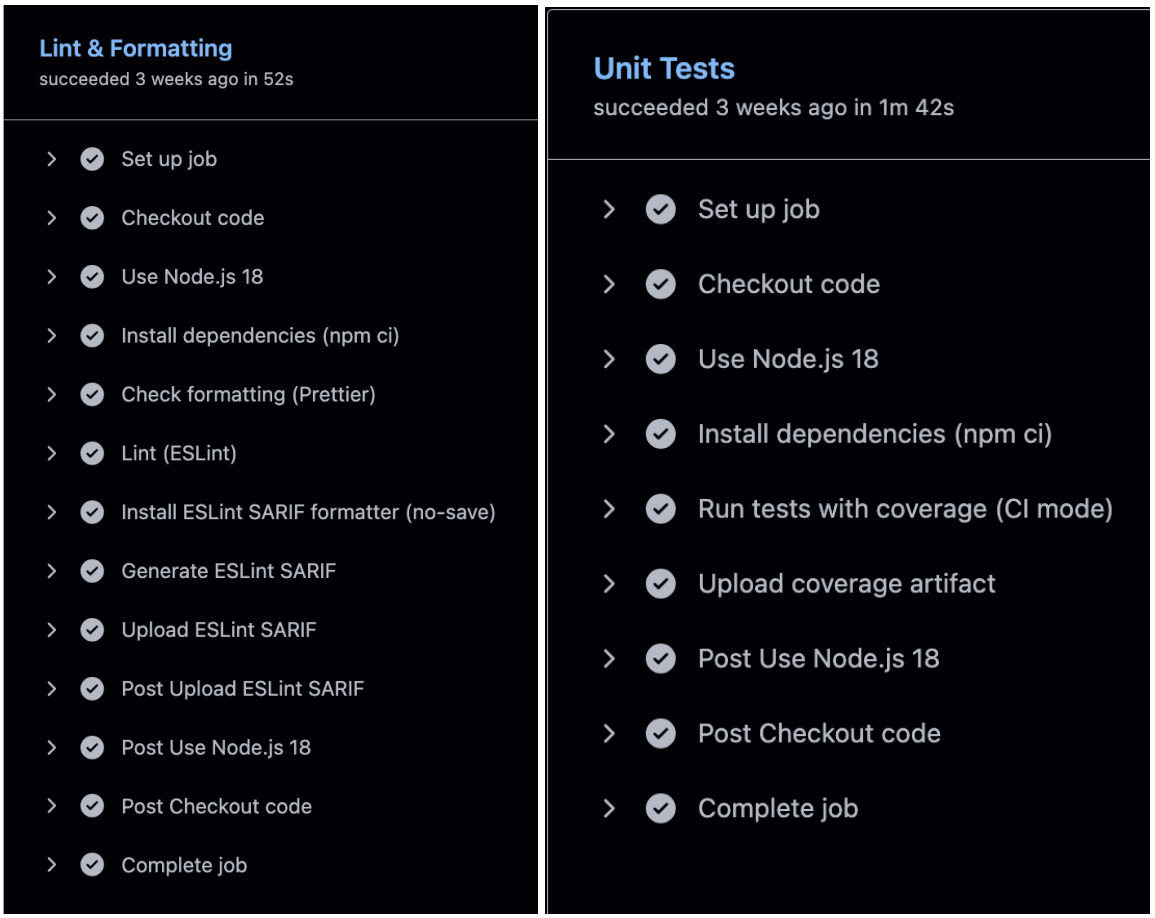
- Runs Prettier for formatting and ESLint for static analysis on all JavaScript / JSX files.
- An ESLint SARIF report is generated and uploaded to GitHub Security.
- Purpose: ensure consistent code style and catch common React / JS issues early.

2. Unit Tests with Coverage

- Runs Jest and React Testing Library in CI mode.
- Generates coverage data (lcov).
- Coverage artifact is stored and later consumed by SonarCloud.

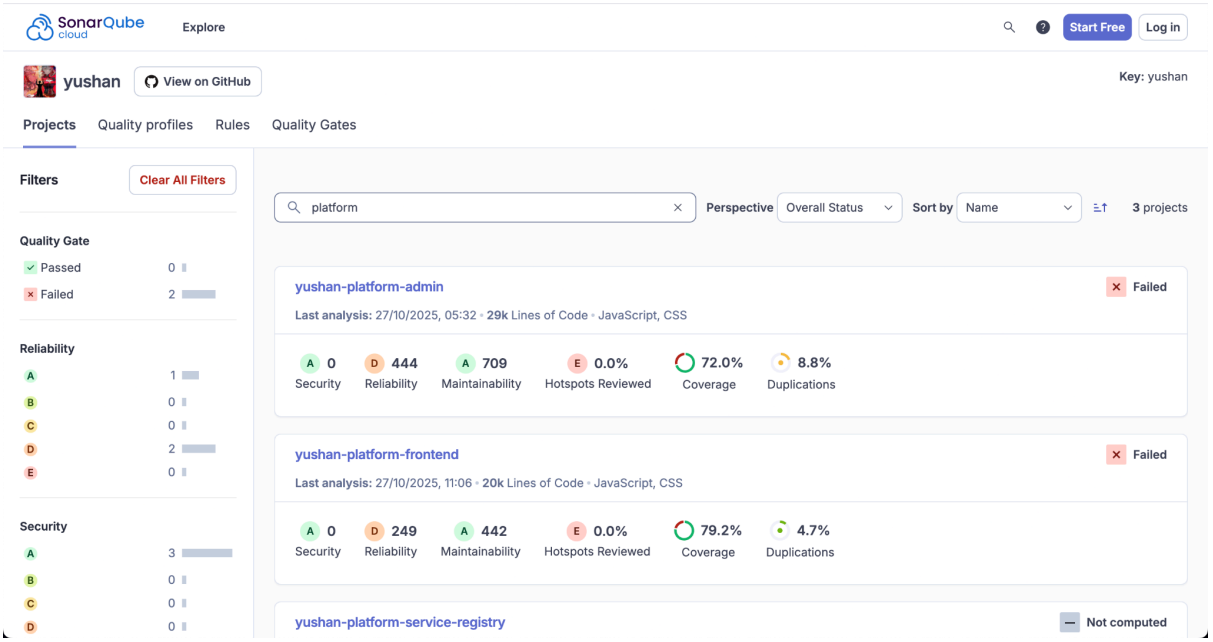


Unit Test of Frontend



3. SonarCloud Analysis

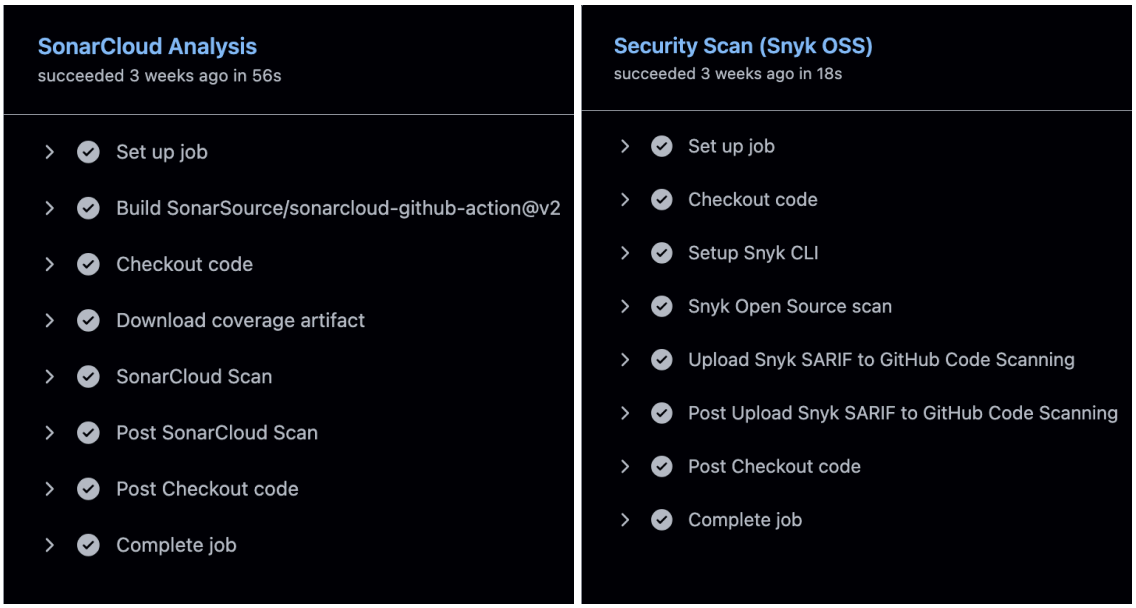
- Uses the official SonarCloud GitHub Action.
- Reads the coverage artifact and uploads metrics (bugs, code smells, duplication, coverage) to the dashboard.



SonarQube Dashboard

4. Security Scan (Snyk OSS)

- Snyk CLI scans package.json dependencies for known vulnerabilities.
- Outputs SARIF and uploads it to GitHub Security.

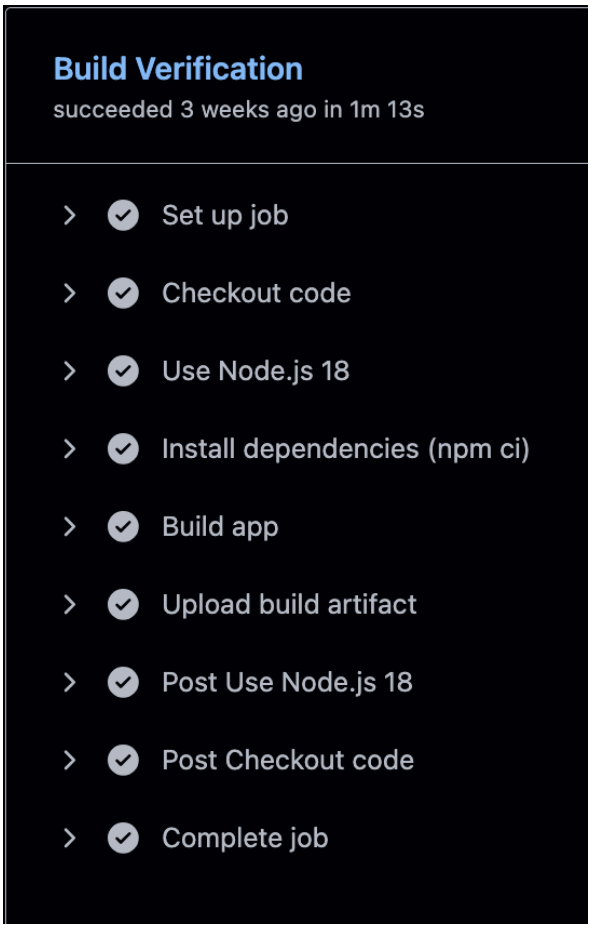


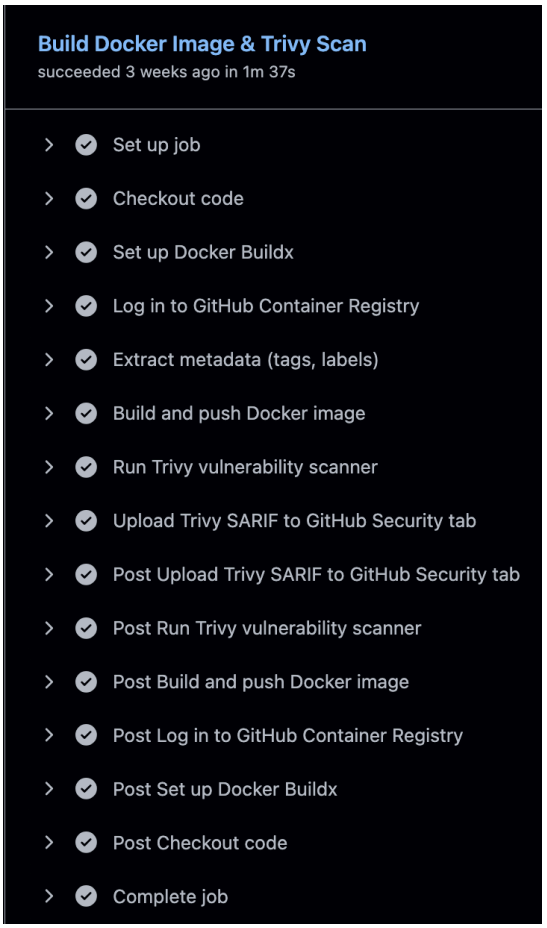
5. Build Verification

- Performs npm run build to ensure the production bundle compiles successfully.
- The build folder is uploaded as a CI artifact for debugging or later use.

6. Docker Build & Trivy Scan

- Builds a Docker image for the frontend (Nginx serving the static React bundle) and pushes it to GHCR.
- Trivy scans the image and uploads SARIF to GitHub Security.



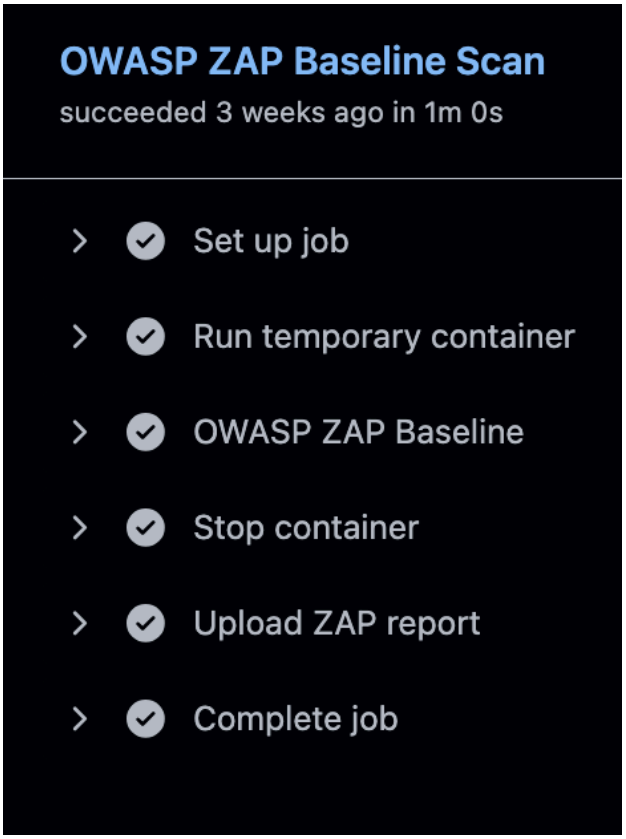


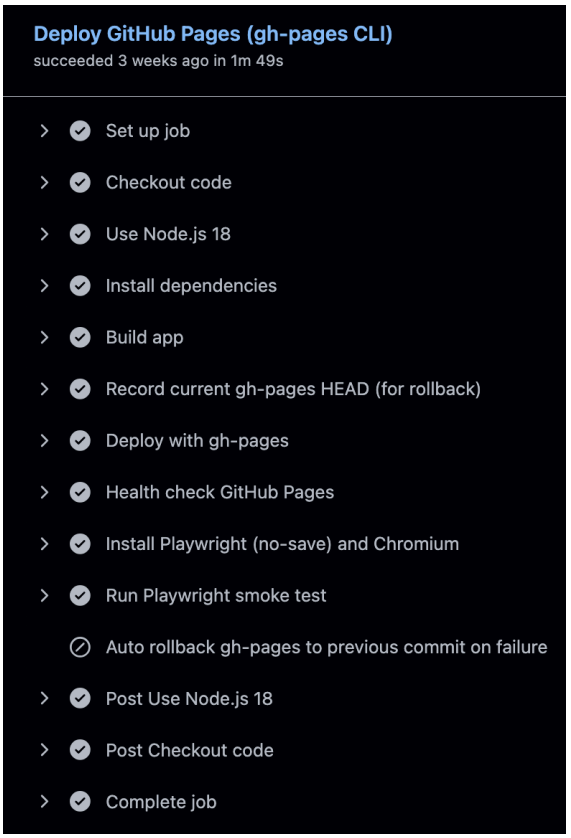
7. OWASP ZAP Baseline (DAST)

- Starts a temporary Docker container with the built frontend.
- Runs ZAP baseline against http://host.docker.internal:8080, producing HTML / JSON / Markdown reports.
- This checks for XSS and other client-side vulnerabilities before deployment.

8. Deploy to GitHub Pages with Health Check and Auto Rollback

- For the reader UI, the pipeline deploys the built React app to GitHub Pages using the gh-pages CLI.
- Before deployment, it records the previous gh-pages commit SHA for rollback.
- After deployment, a health check script verifies that index.html, the main JS and CSS bundles all return HTTP 200.
- A lightweight Playwright smoke test opens the GitHub Pages URL in Chromium, waits for content under #root, and fails if there are console errors.
- If any of these checks fail, an automatic rollback restores the previous gh-pages commit.





9. CI Report Summary

- A final job aggregates all artifacts (coverage, ESLint SARIF, Snyk SARIF, Trivy SARIF, ZAP HTML) and writes a summary.md similar to the backend.

5.2.4 Benefits and Future Enhancements

- **Benefits**
  - Every code change goes through a full test pyramid: unit tests, integration tests, static analysis, SCA, container scanning, and DAST.
  - CI pipelines for backend and frontend are consistent but independent, matching the multi-repository microservice architecture.
  - Automated publishing of SARIF files to GitHub Security creates a single place to view vulnerabilities.
  - Image digest artifacts ensure that deployments promote exactly the image that passed all checks.
  - Health checks, ZAP scans and Playwright smoke tests provide confidence that the platform is working correctly after each integration.
- **Future enhancements**
  - Add contract testing between microservices (e.g. using Pact) as another CI stage.
  - Use matrix strategies or test sharding to further parallelize test execution and detect flaky tests.
  - Integrate accessibility scanning tools (such as axe) and visual regression tests for the frontend.
  - Overall, the continuous integration setup gives the team fast feedback, strong security assurance, and a repeatable path from commit to deployable artefact for every microservice and frontend application.

5.3 Continuous Delivery

Our continuous delivery pipeline automates deployment from the main branch to the running environments and verifies that each deployment is healthy and secure.

5.3.1 Environments and Overall Approach

- **Local / Development**
  - Each developer runs the services locally using Docker Compose or IDE run configurations.
  - Environment variables are loaded from shared .env / sample files so that local behaviour is close to production.
- **Planned Staging vs. Actual Deployment**
  - In the initial design, we planned two shared environments on DigitalOcean:
    - **Staging** – for regression, load and security testing.
    - **Production** – for real users and final demos.
  - However, each extra droplet and managed database increases monthly cost. To keep the project affordable for a student team, we finally **provisioned only the production environment** and used local setups for pre-production testing.
  - The deployment scripts and Terraform configuration still support a future staging environment, but in this project only the production stack is actually running.
- **Production**
  - Backend microservices (user, content, engagement, gamification, analytics) and platform components (service registry, config server, API gateway, Kafka, Redis, PostgreSQL, Elasticsearch) run on DigitalOcean droplets.
  - The main reader frontend is served via **GitHub Pages**, and an Nginx-based Docker image is also available for running the frontend inside a container when needed.

The key principle is: **merging to main implicitly approves a production deployment**. Because this is a small academic team, the combination of code review + passing CI pipeline acts as the promotion gate instead of a separate manual approval step.

5.3.2 Backend Delivery to DigitalOcean

For backend services, the content service pipeline is the reference pattern and is fully automated end-to-end.

Trigger and Approval

- The **production-deploy** job runs automatically when:
  - an update is pushed to main, or
  - a special CI test branch is used for dry-run deployment.
- Only code that has passed all CI jobs (unit tests, integration tests, linting, SCA, container scanning) can reach this stage.
- In practice, **the reviewer who approves the Pull Request into main is responsible for approving the deployment**.

Deployment Steps

1. **Select immutable image**
   - The CI pipeline builds a Docker image for the service and publishes it to GitHub Container Registry.
   - An image-digest.txt artifact is generated so that the deploy job can pull **exactly the image that was tested**.
2. **Remote deployment via SSH**
   - The deploy job connects to the DigitalOcean droplet using the SSH GitHub Action.

- It pulls the latest image from GHCR, stops and removes the existing container, then runs a new container with:
  - database, Redis and Kafka configuration,
  - S3-compatible object storage credentials (DigitalOcean Spaces),
  - Eureka and config-server endpoints,
  - centralized logging settings (Docker syslog driver → Logstash → Elasticsearch).

3. **Automated health verification on the server**
   - The script waits for the container to start, then calls /actuator/health several times with retries.
   - If the service does not report status: "UP" within the retry window, the job fails and the deployment is considered unsuccessful.
4. **External production check and security validation**
   - After the service is up, a separate job runs **OWASP ZAP Baseline** against the public URL (https://yushan.duckdns.org).
   - ZAP checks for SQL injection, XSS, authentication flaws and security misconfigurations and uploads HTML/JSON/Markdown reports as artifacts.

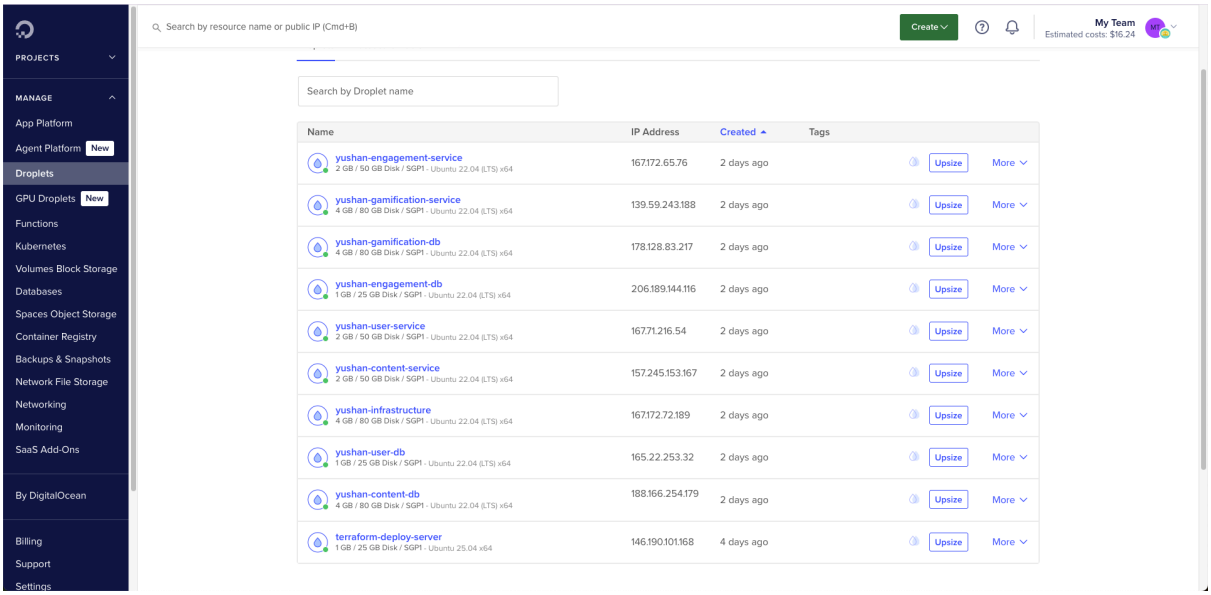
Who verifies the deployment

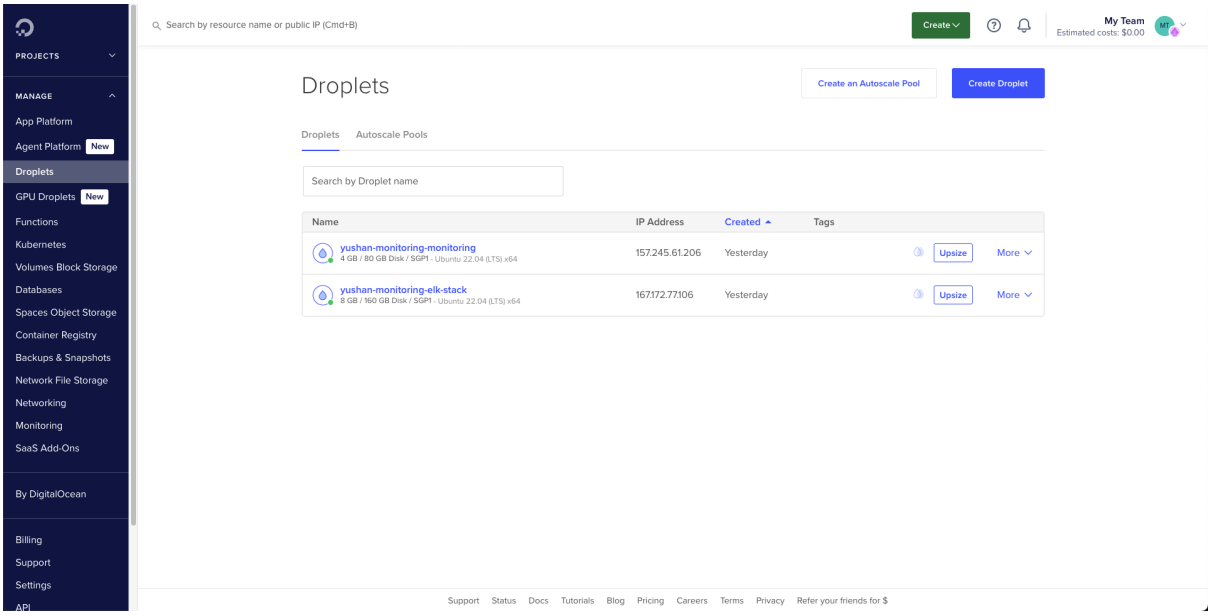
- The **DevOps lead / backend owner** checks the GitHub Actions run, confirms that:
  - health checks passed,
  - ZAP did not report new critical issues,
  - logs are flowing into the monitoring stack.
- For important changes, another teammate does a quick manual smoke test through the frontend (e.g. open novel list, read a chapter, add a comment).

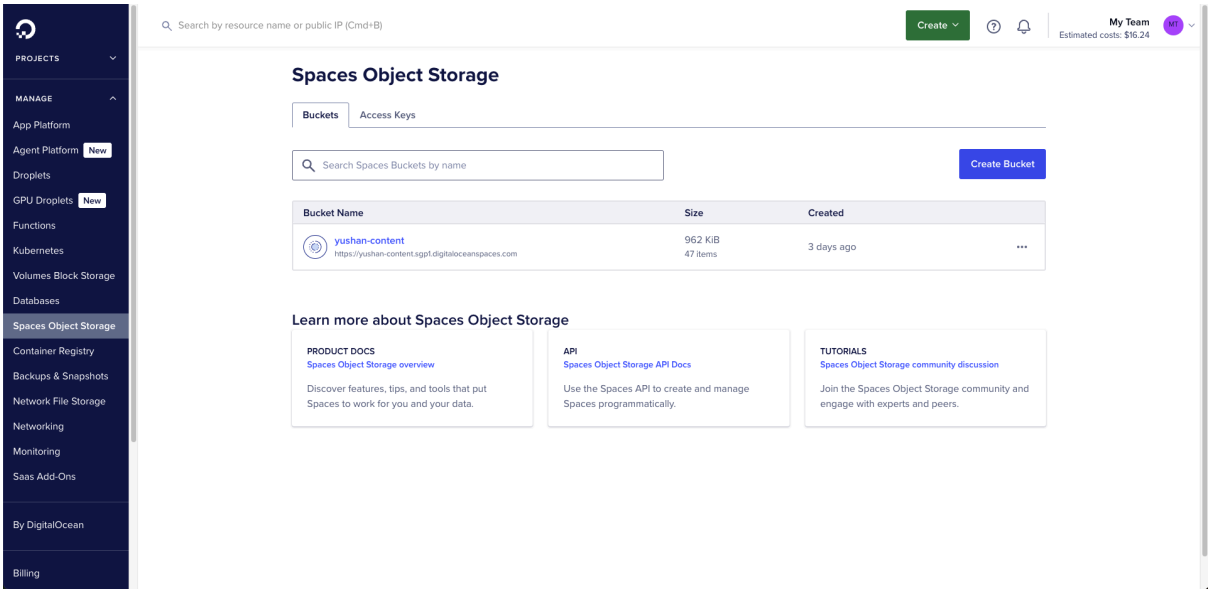
Rollback

- If a deployment causes issues, we can:
  - re-run the deploy job with the **previous image digest** (kept as artifact for 30 days), or
  - manually pull and start the previous image via SSH.
- Because infrastructure is managed by Terraform and Docker, rollback does not require rebuilding code.

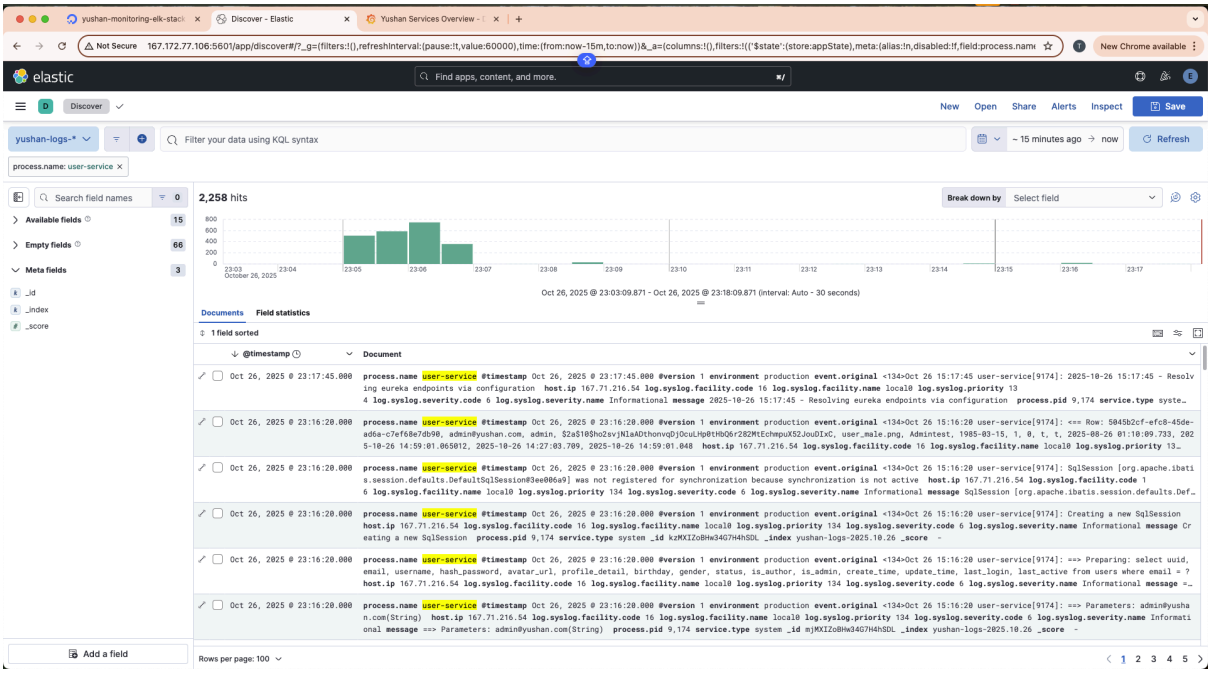
The same deployment pattern is reused for other microservices by changing hosts, ports and secrets, so the delivery model is consistent across the backend.



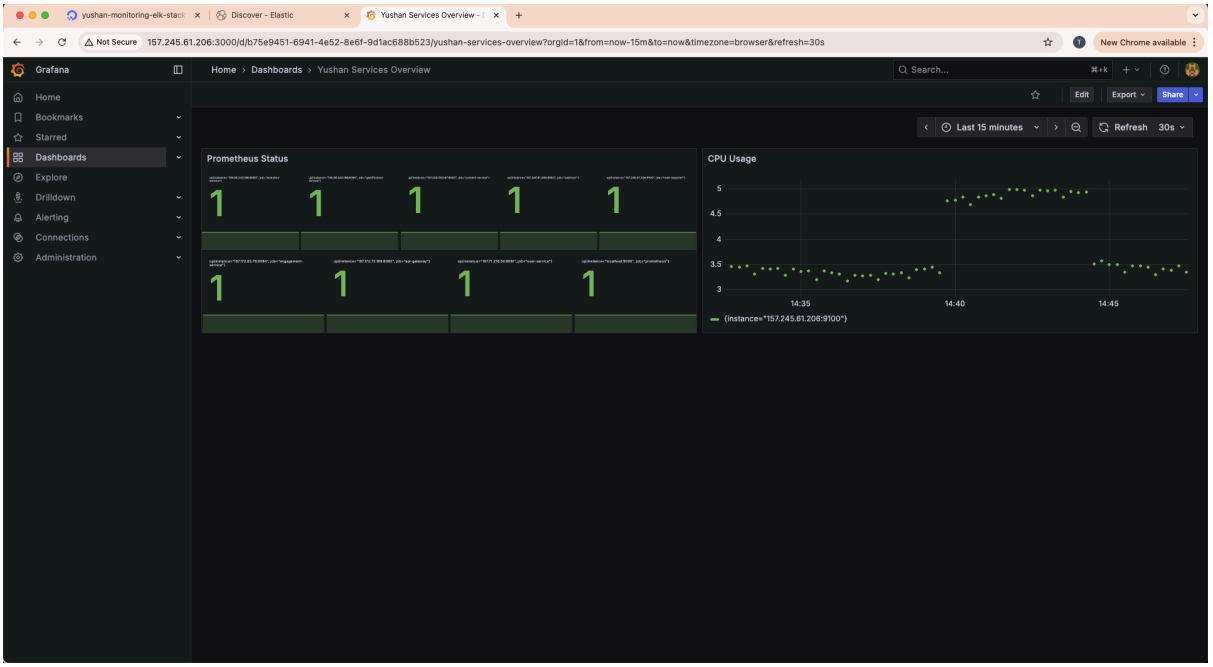




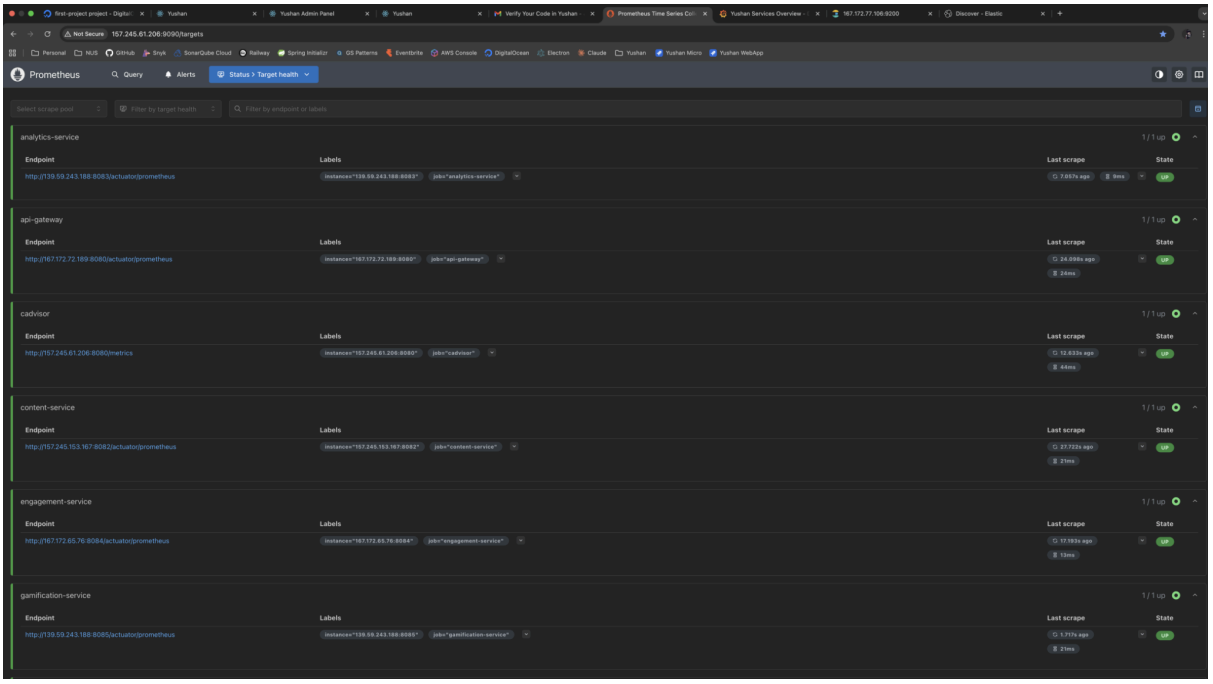
Digital Ocean Setup



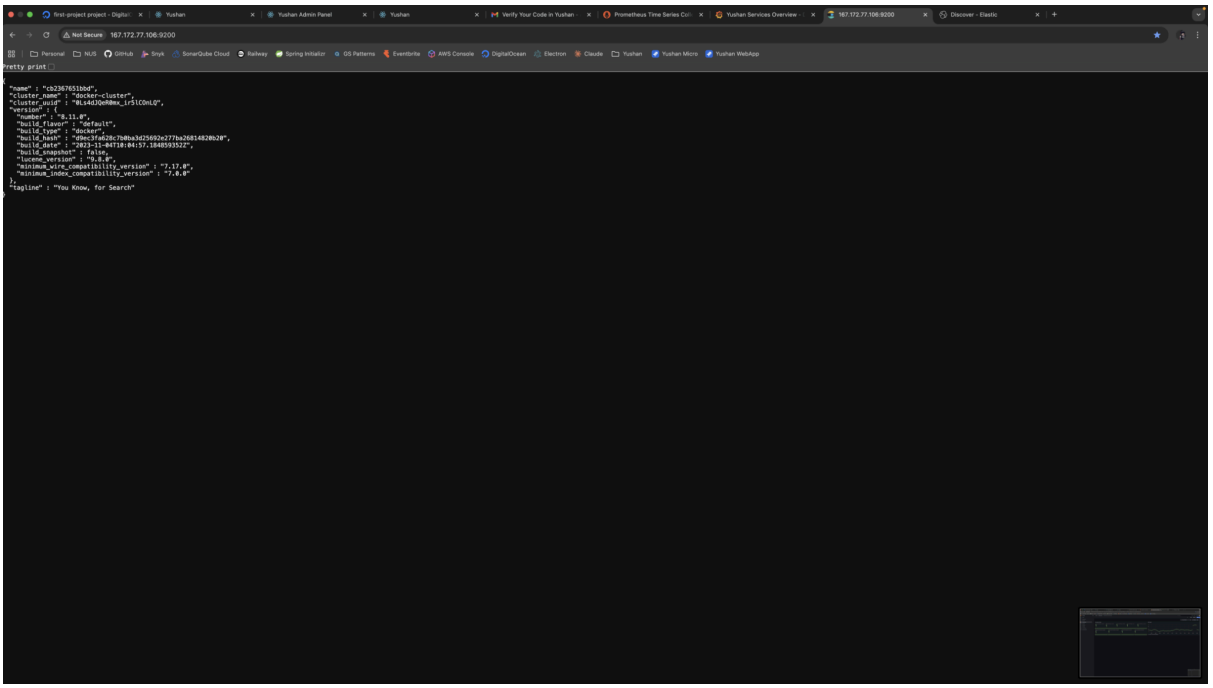
Kibana Dashboard



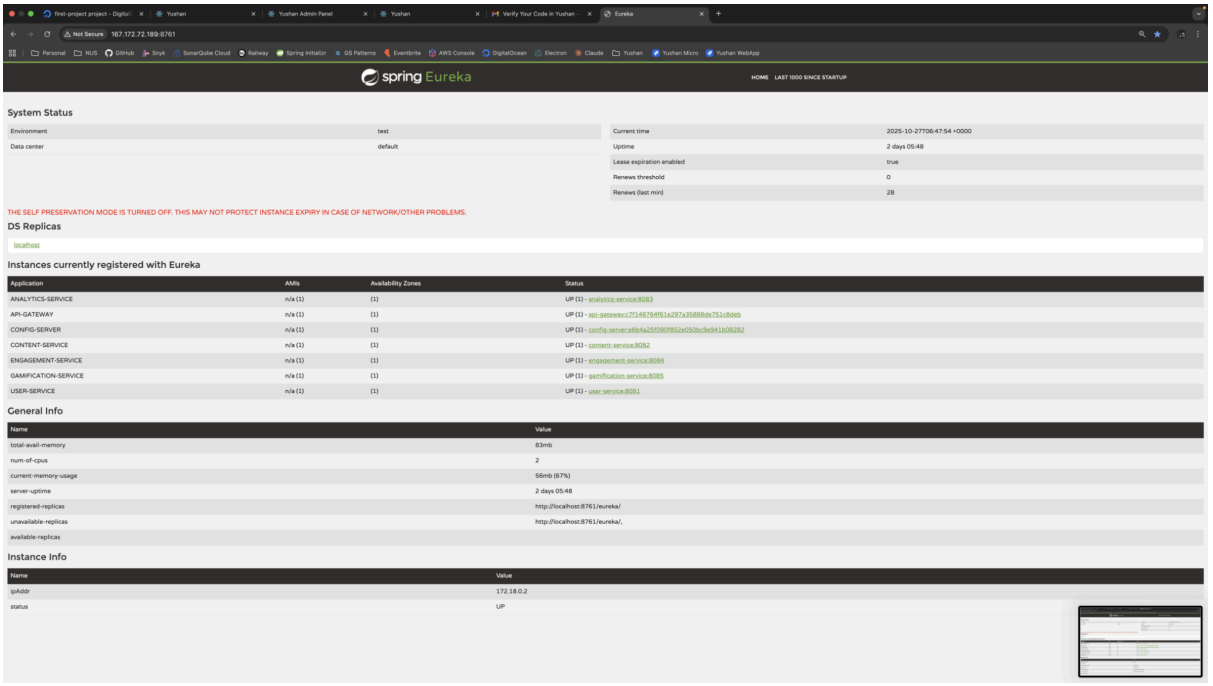
Grafana Dashboard



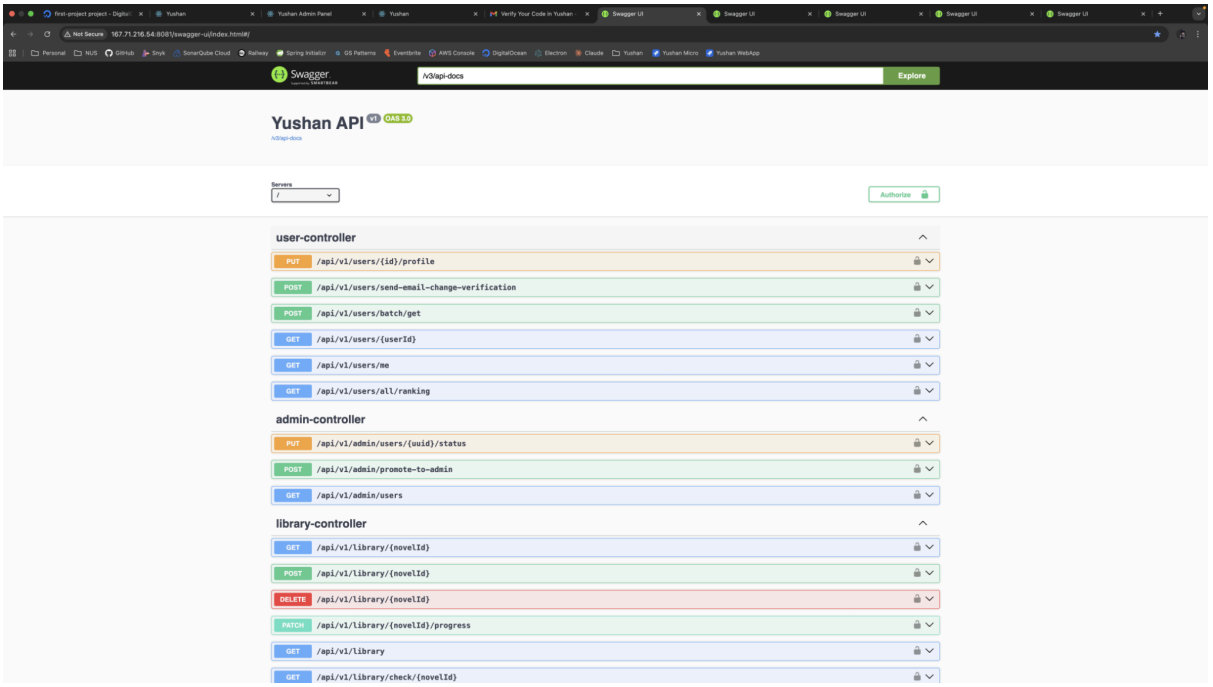
Prometheus Dashboard



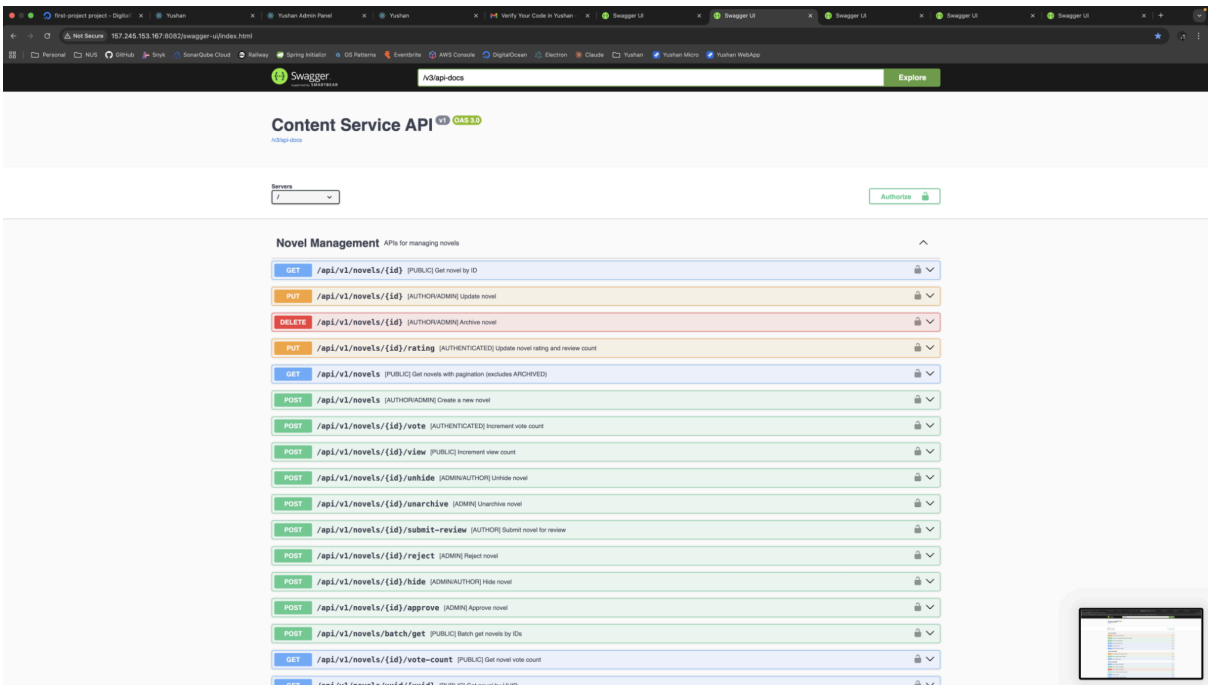
Elasticsearch



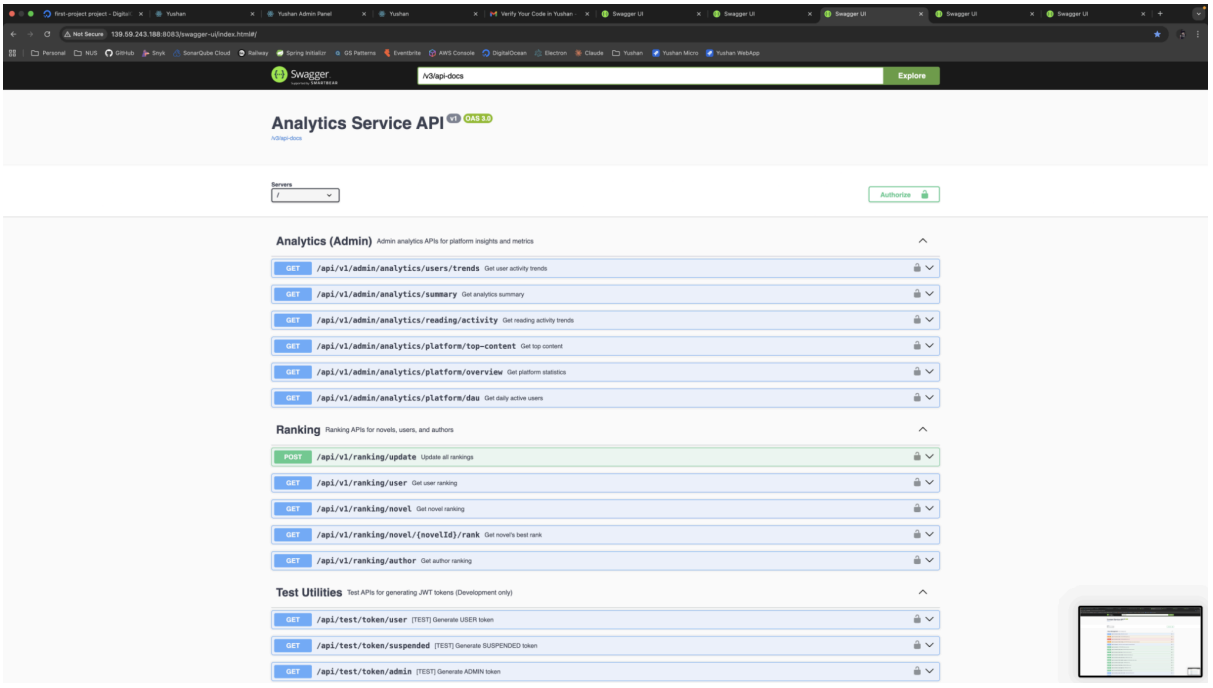
Eureka Dashboard



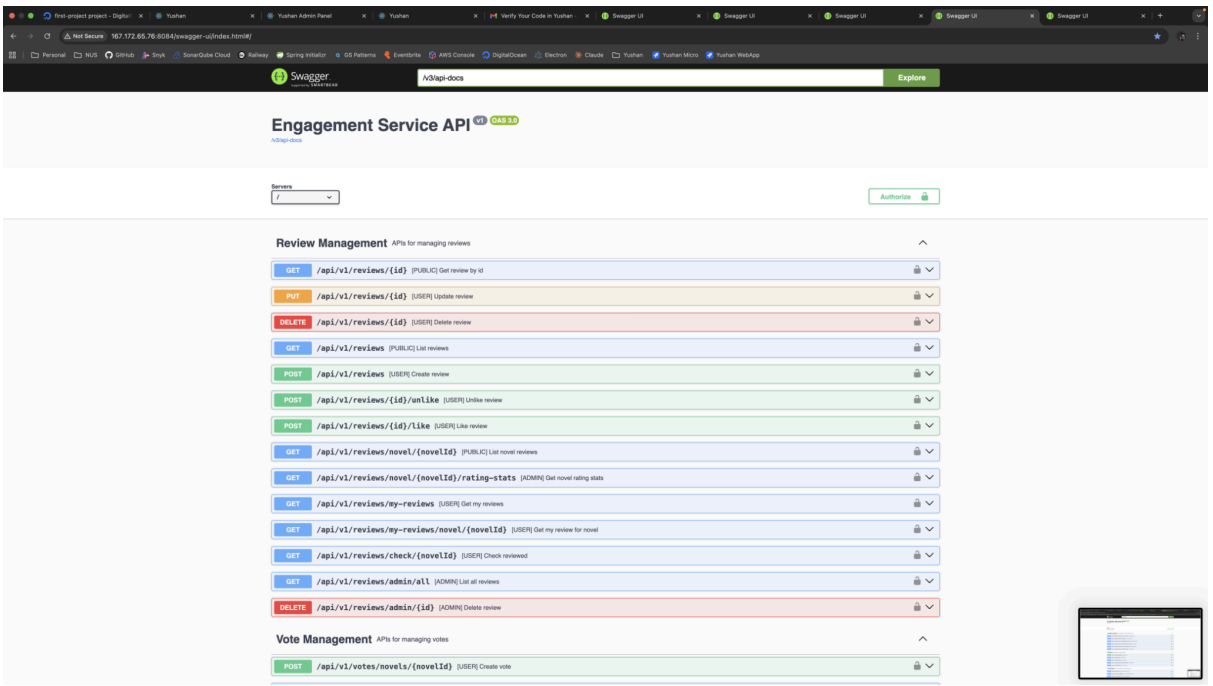
User Service Swagger Page



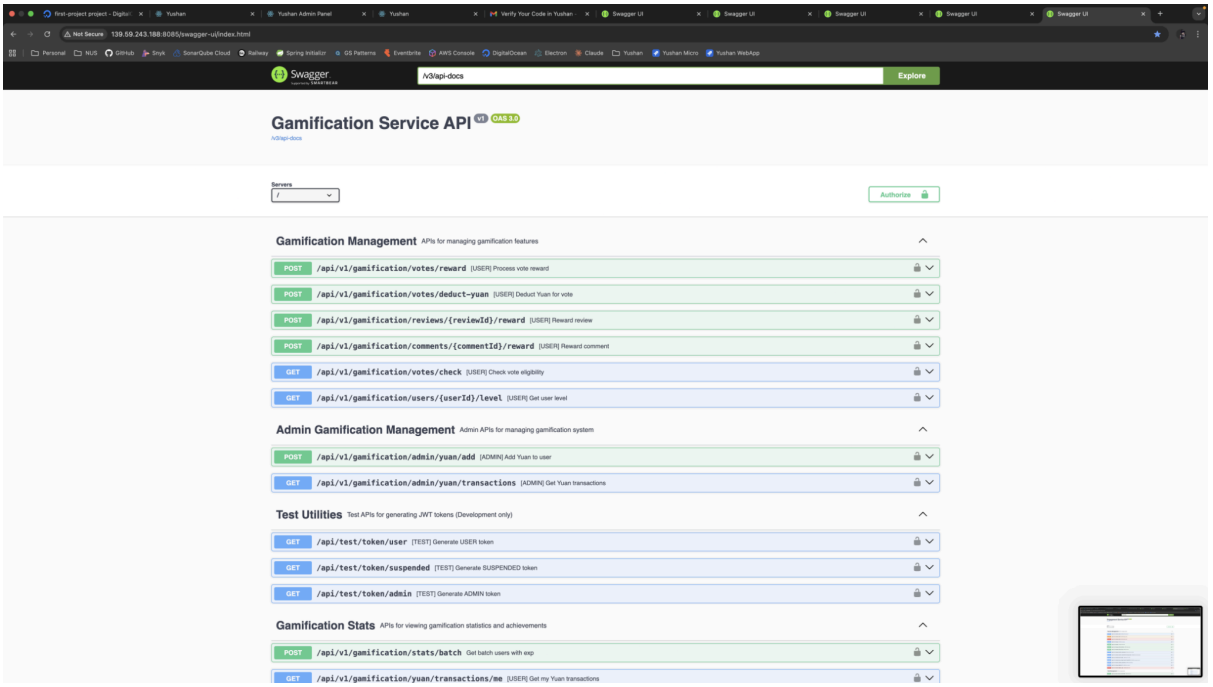
Content Service Swagger Page



Analytics Service Swagger Page



Engagement Service Swagger Page



Gamification Service Swagger Page

5.3.3 Frontend Delivery to GitHub Pages

The reader frontend has its own continuous delivery flow based on GitHub Actions and GitHub Pages.

Trigger and Approval

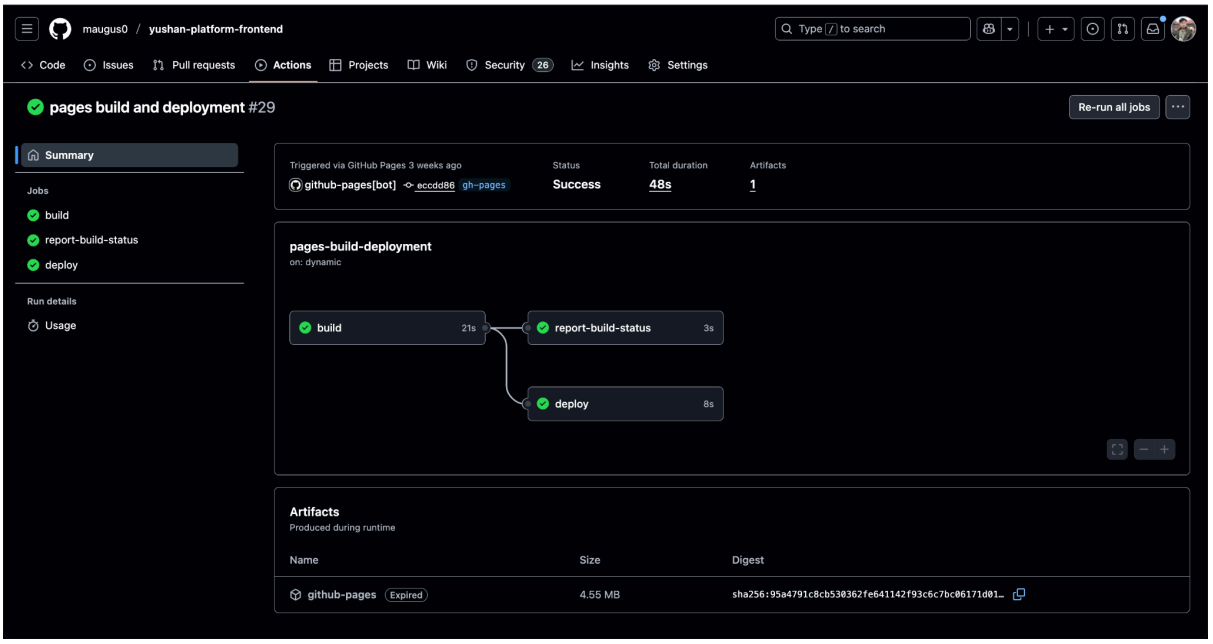
- The deploy-pages job runs when changes are pushed to main or when a PR targets main.
- As with the backend, **the merge into main acts as the approval** to deploy the latest successful build to production.

Deployment Steps

1. **Build and package**
    - The pipeline installs dependencies (npm ci) and runs npm run build to create an optimized React bundle.
2. **Record previous state**
    - Before deployment, the workflow records the current gh-pages commit SHA. This is used for automatic rollback if the new release is not healthy.
3. **Publish to GitHub Pages**
    - The built bundle is pushed to the gh-pages branch using the gh-pages CLI.
    - GitHub Pages serves the static site from this branch.
4. **Automated health check**
    - A script repeatedly checks the public URL:
        - index.html must return HTTP 200.
        - the main JS and CSS bundles referenced in the HTML must also return HTTP 200.
5. **Playwright smoke test**
    - A small Node script uses Playwright + Chromium to:
        - open the GitHub Pages URL,
        - wait until content is rendered under #root,
        - fail if there are any console errors in the browser.
6. **Auto-rollback on failure**
    - If either the HTTP health check or the Playwright smoke test fails, the workflow automatically resets the gh-pages branch back to the previously recorded commit and pushes it.
    - This gives a simple but effective **one-click rollback** mechanism.

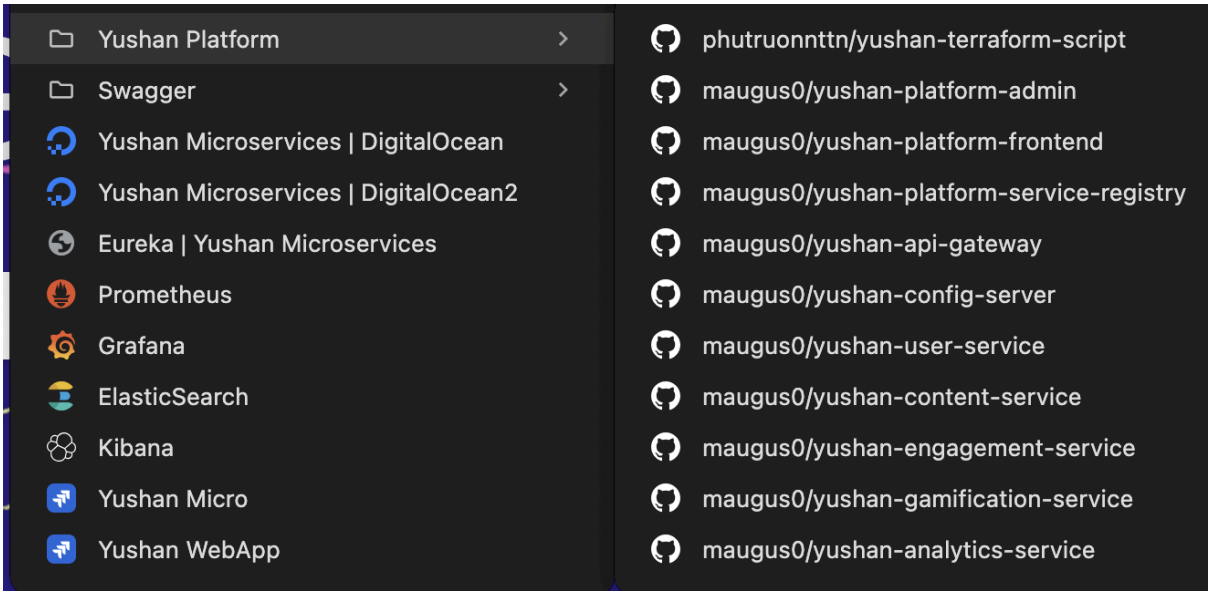
Who verifies the deployment

- The **frontend owner** reviews the GitHub Actions run and, for larger UI changes, does a quick manual sanity check (login, browse novels, open ranking pages).
- Because the rollback is automated, a bad deployment can be reversed quickly without SSH access.



5.3.4 Summary and Future Improvements

- Continuous delivery for Yushan combines **automated deployment** (DigitalOcean for backend, GitHub Pages for frontend) with **post-deploy verification** (health endpoints, ZAP scans, Playwright smoke tests).
- For this project we deliberately operate with **only one shared production environment** to control cost, while keeping the scripts ready for a future staging environment.
- Approvals are kept simple: **passing CI + code review + merge to main** equals permission to deploy.
- In the future, we can extend this setup by:
    - adding a true **staging** environment on DigitalOcean when budget allows,
    - using GitHub Environments or Terraform workspaces with explicit approvers,
    - adopting blue/green or canary deployments, and
    - wiring synthetic monitoring and automatic rollback based on SLO breaches.
- This pipeline ensures that new changes are deployed in a repeatable way, verified automatically, and can be rolled back with minimal effort.



5.4 Non-Functional Testing Strategy

The non-functional testing strategy for Yushan Platform ensures the system meets performance, security, reliability and usability expectations across both backend microservices and the frontend web application. A combined approach of **automated CI pipelines**, **manual validation**, and **industry-standard tooling** is used to verify system behaviour under real-world conditions.

Performance & Load Testing

Backend:
Performance validation is conducted using **Apache JMeter**, simulating real user flows such as login, profile retrieval, browsing, search, and ranking queries. Tests cover load (10–1000 concurrent users), stress, and recovery scenarios. Key metrics tracked include response time percentiles, throughput, error rate, CPU/memory utilisation, and baseline targets (e.g., API p95 < 200ms, DB query < 100ms).

Frontend:
Performance profiling uses **Chrome DevTools** and **Lighthouse** for FCP, LCP, TTI, TBT, CLS, bundle size, and runtime smoothness (60fps target). Network throttling (Fast/Slow 3G) ensures usability on low-bandwidth devices.

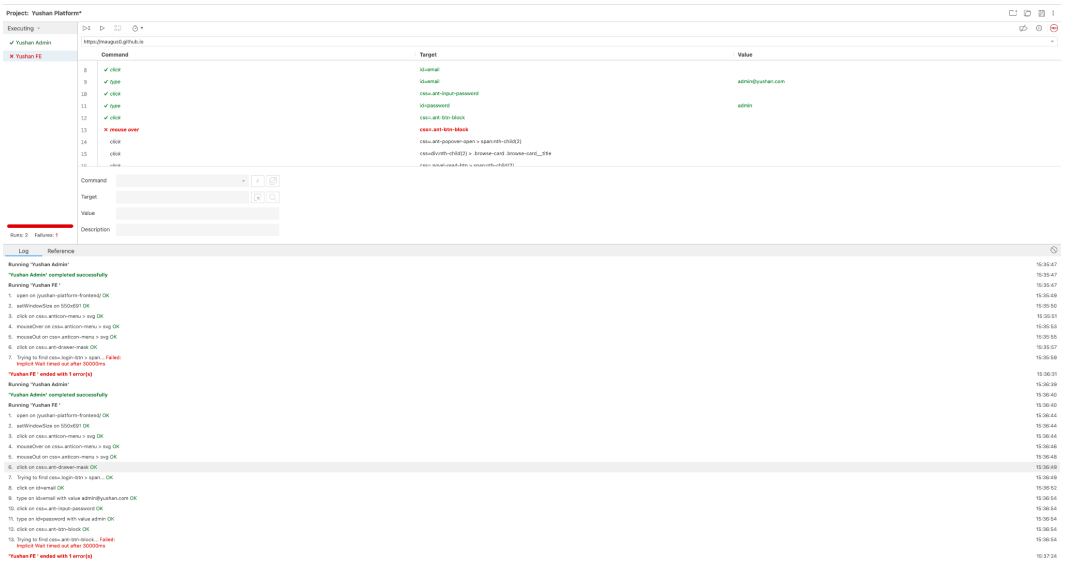
API & Integration Testing

Backend APIs are tested through **Swagger (OpenAPI)** for contract validation and **Postman** collections for authentication flows, CRUD operations, error handling, and boundary checks. Frontend integration tests reuse the same Postman suites to validate CORS behaviour, token handling, UI error states, and large/empty dataset handling. Swagger schemas are used by the frontend team for accurate API consumption.

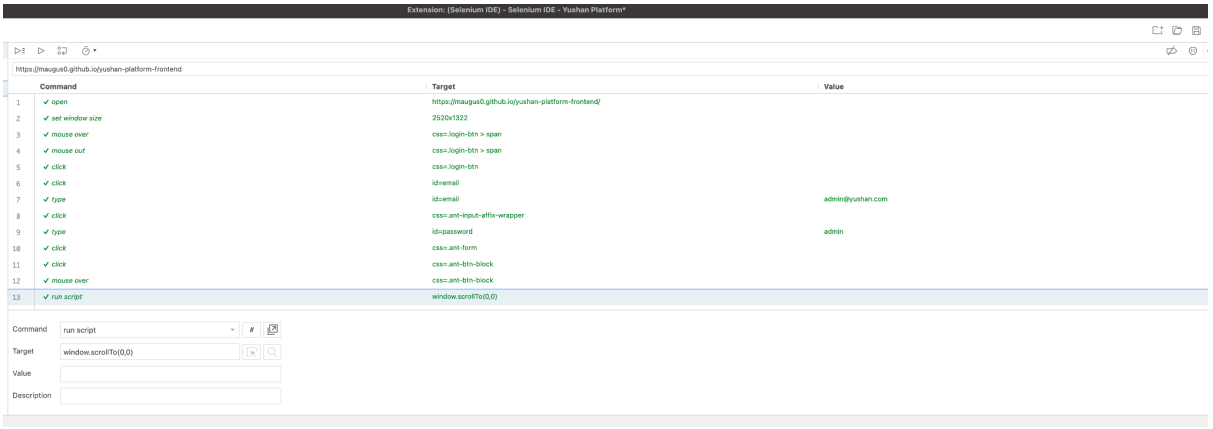
End-to-End & UI Testing

E2E Testing:

- **Selenium IDE** covers core UI workflows such as login, navigation, form validation, search, and content browsing.
- **Playwright smoke tests** run in CI (chromium-headless) to ensure pages load, essential DOM elements render, and no console errors occur, with automatic rollback on failure.



UI/Visual Testing:
Manual visual regression checks are performed on Chrome, Firefox, Safari, Edge across desktop, tablet, and mobile breakpoints. Responsiveness is validated using DevTools Device Mode.



Security Testing

Security validation is integrated into CI via:

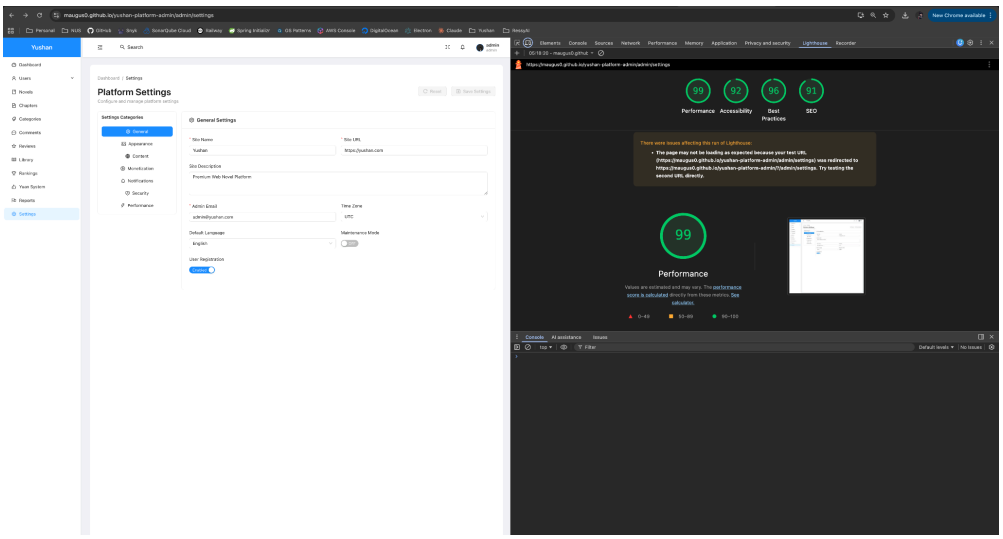
- **Snyk**: dependency vulnerability scanning
- **Trivy**: container image scanning
- **OWASP ZAP Baseline DAST**: scanning deployed site for XSS, missing headers, and exposure risks
- **CSP header validation**: manual testing to ensure inline scripts/styles are blocked

High-severity findings automatically fail CI.

Compatibility & Accessibility Testing

Cross-browser and cross-device testing ensures consistent behaviour across:

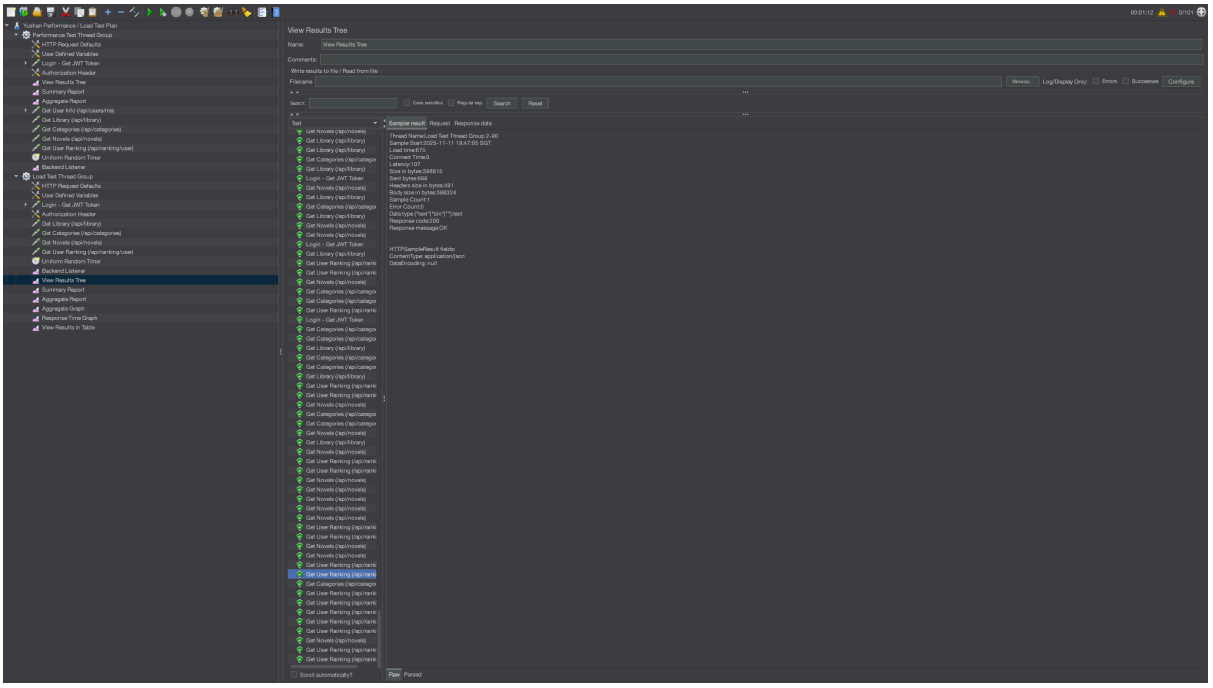
- Chrome, Firefox, Safari (macOS/iOS), Edge, Windows, macOS, iOS, Android, and standard screen resolutions.
- Accessibility validations include keyboard navigation, proper form labels, and screen-reader-friendly structures (basic WCAG alignment).



Usability, Exploratory & Regression Testing

Regular **exploratory sessions** and **bug-bash events** ensure discovery of edge-case behaviours. Regression is maintained via:

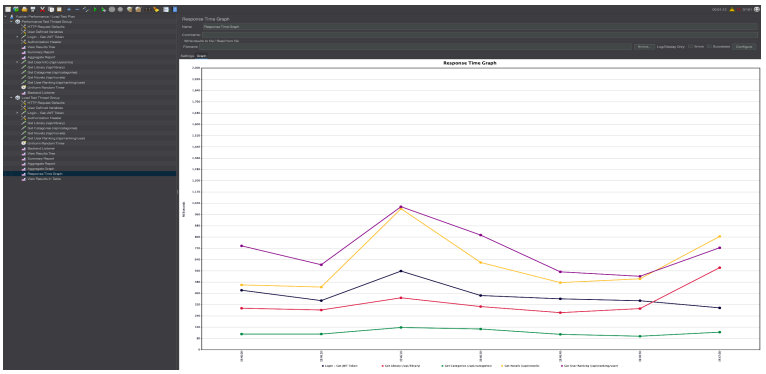
- Automated Jest unit tests (>80% coverage)
- UI regression via Playwright
- Manual regression checklists before release (authentication, navigation, search, profile management)



Monitoring & Deployment Validation

- CI/CD deploys run automated health checks (HTTP 200, bundle load success, no JS errors).
- Real-user monitoring includes production console error tracking and feedback review.
- Performance and stability metrics guide iterative optimisation.

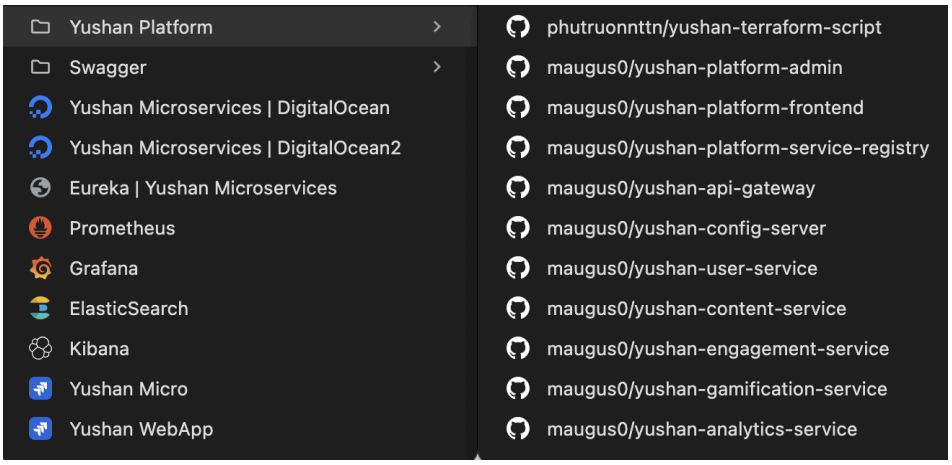
Summary: Yushan's non-functional testing strategy integrates backend performance and resilience testing, frontend performance and compatibility validation, API contract assurance, and multi-layered security scanning. The combination of automated pipelines and targeted manual testing provides strong confidence that the platform meets expected standards of speed, stability, security, usability, and scalability across all environments.



6. Individual Members Activity Contribution Summary

We have the created and setup these following repositories as part of ASS Implementation:

- https://github.com/phutruonnttn/Digital_Ocean_Deployment_with_Terraform
- https://github.com/maugus0/yushan-platform-admin
- https://github.com/maugus0/yushan-platform-frontend
- https://github.com/maugus0/yushan-platform-service-registry
- https://github.com/maugus0/yushan-api-gateway
- https://github.com/maugus0/yushan-config-server
- https://github.com/maugus0/yushan-user-service
- https://github.com/maugus0/yushan-content-service
- https://github.com/maugus0/yushan-engagement-service
- https://github.com/maugus0/yushan-gamification-service
- https://github.com/maugus0/yushan-analytics-service



The **Yushan Platform Frontend** and **Yushan Platform Admin** repositories were initially cloned from the earlier monolithic implementation, but required substantial adjustments.

For the **Yushan Platform Frontend**, major modifications were completed by **Zhu Yuhui** and **Yang Shuang**, who were also fully responsible for developing all associated test cases.

For the **Yushan Platform Admin**, all functional adjustments were implemented by **Ahan Jaiswal**, while test code coverage was jointly completed by **Ahan Jaiswal (60%)** and **Yang Shuang (40%)**.

Across backend services, the **User Service** was fully developed and tested by **Zhang Yan**, who additionally contributed part of features and test coverage for the **Gamification Service**.

The **Content**, **Engagement**, and primary portions of the **Gamification Service** were developed and tested by **Truong Phu Nguyen**.

The **Analytics Service** was entirely developed and tested by **Ahan Jaiswal**, who handled this service independently while also owning all frontend responsibilities, which resulted in relatively lower test code coverage for analytics service compared to other services.

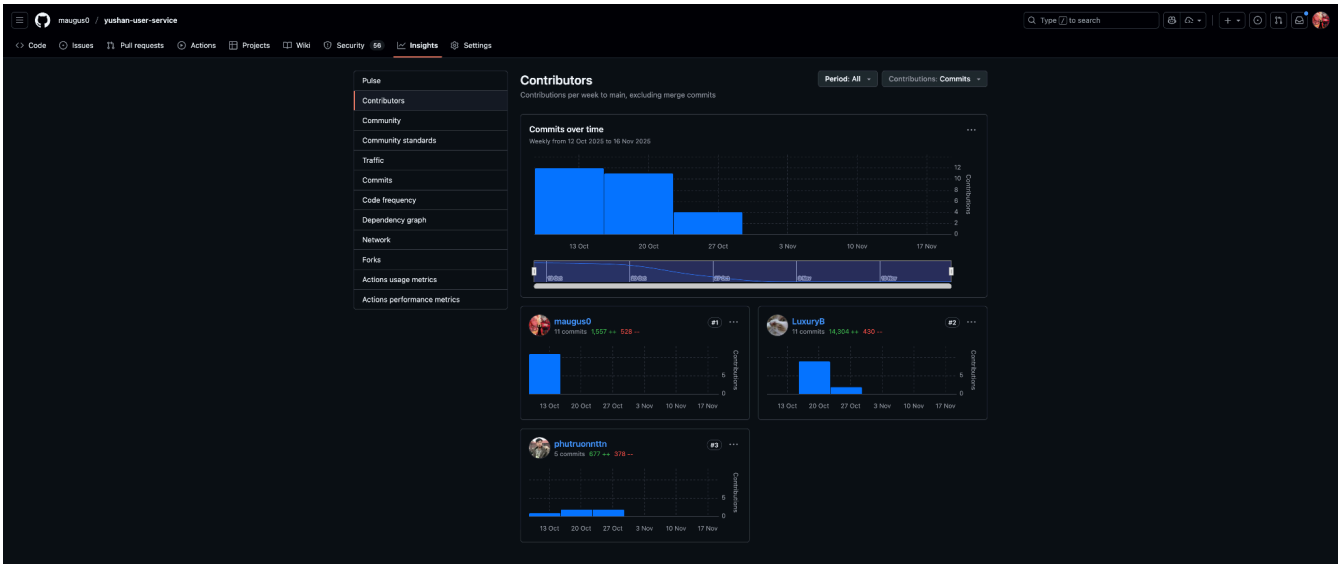
Common infrastructure repositories, including the **Service Registry (Eureka)** and **API Gateway**, were set up by **Ahan Jaiswal**, with additional contributions from **Truong Phu Nguyen**.

The **Config Server** was set up by **Truong Phu Nguyen** with support from **Ahan**, while the **Terraform deployment code** was authored entirely by **Truong Phu Nguyen**.

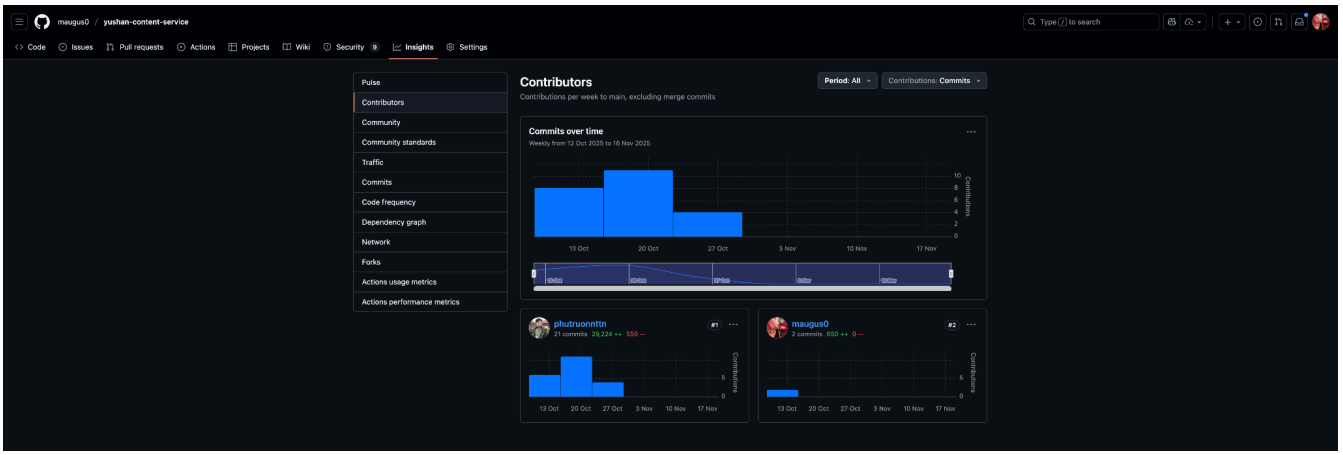
Yang Shuang contributed to GamificationSvc & **Zhu Yuhui** also contributed to EngagementSvc.

Github Code Contribution Screenshots:

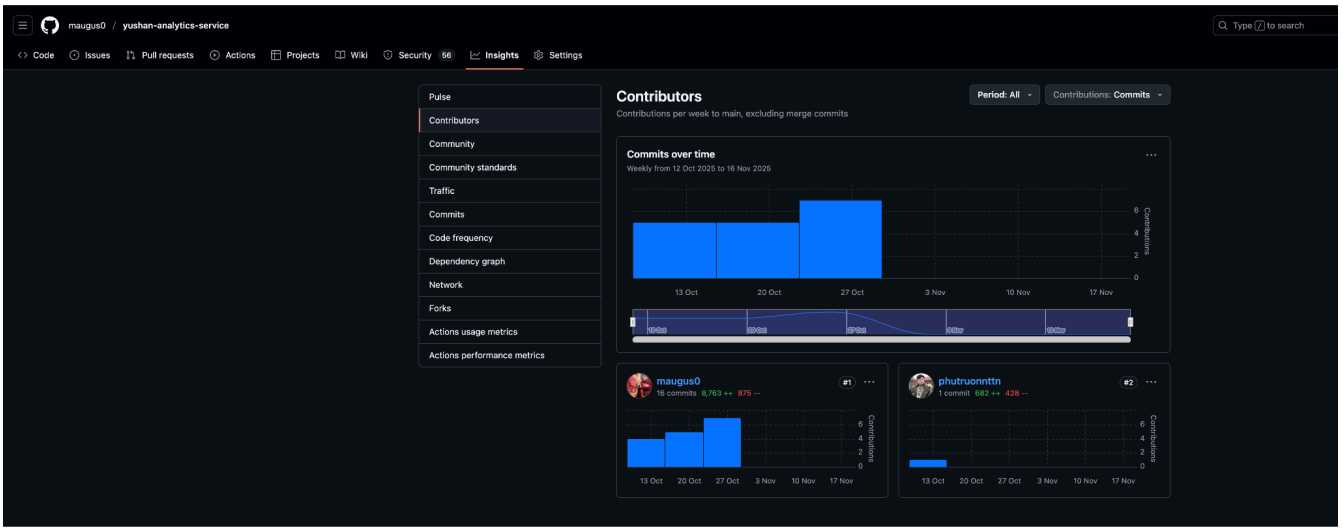
- User Service



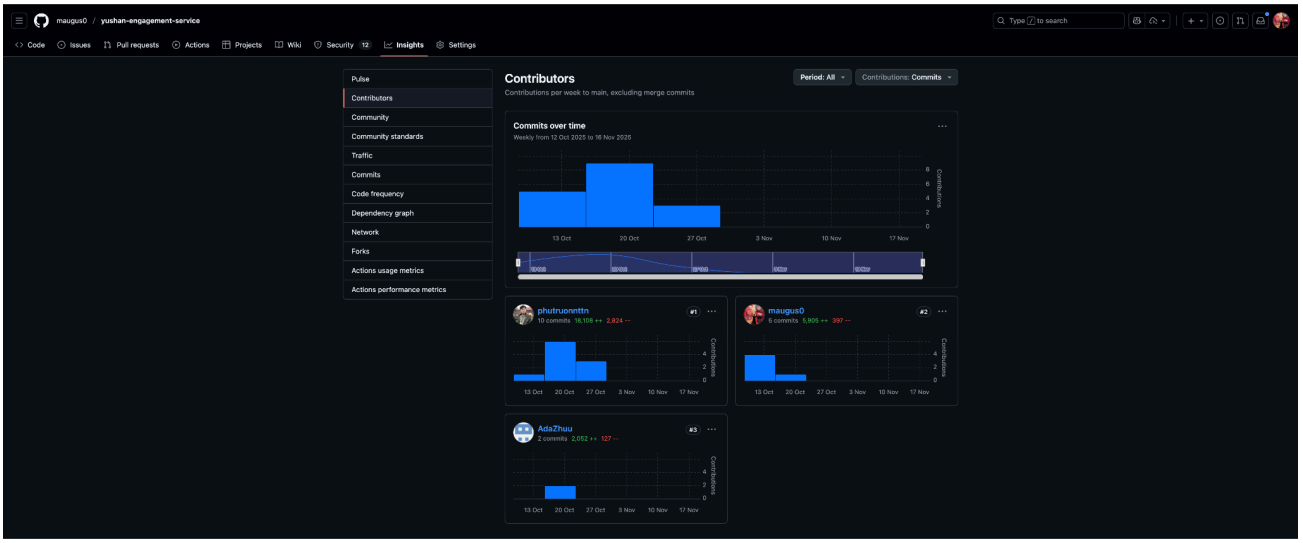
- Content Service



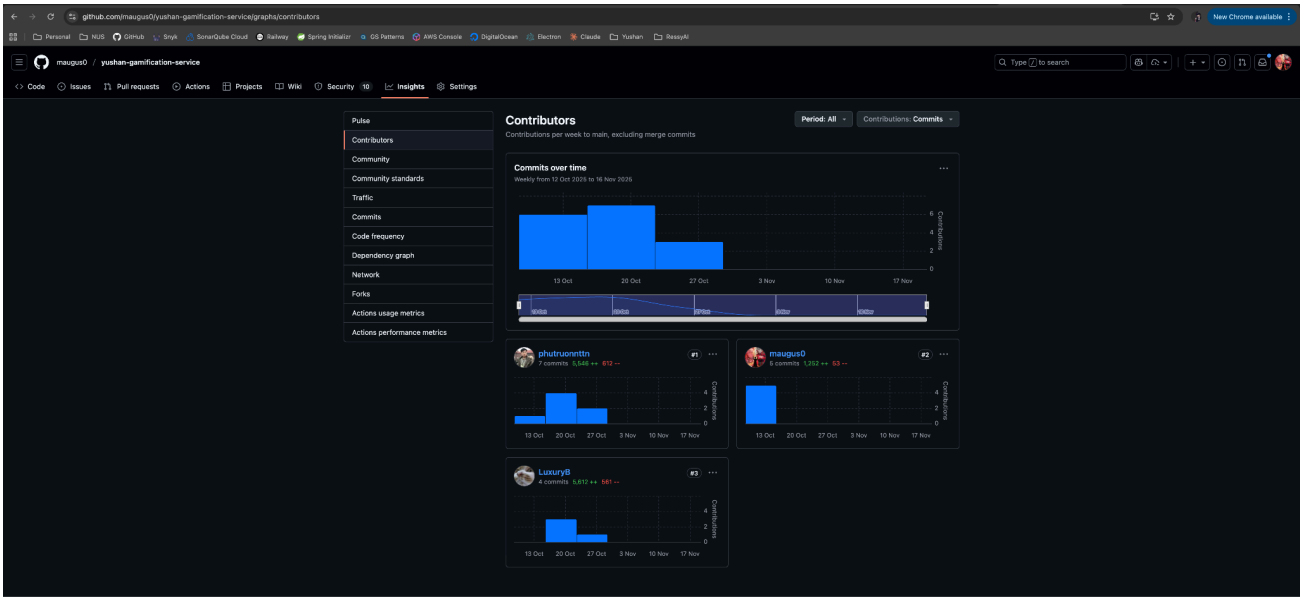
- Analytics Service



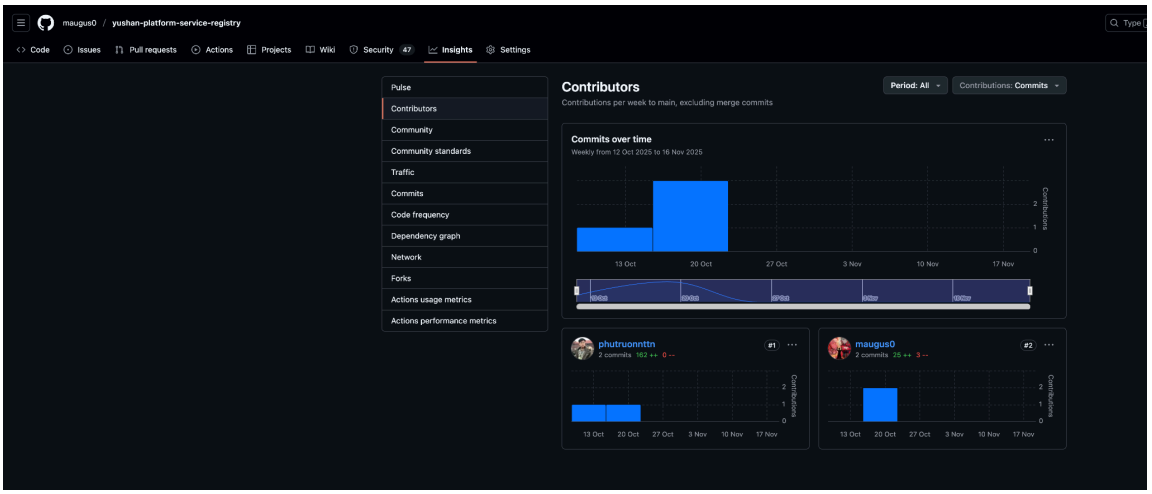
- Engagement Service



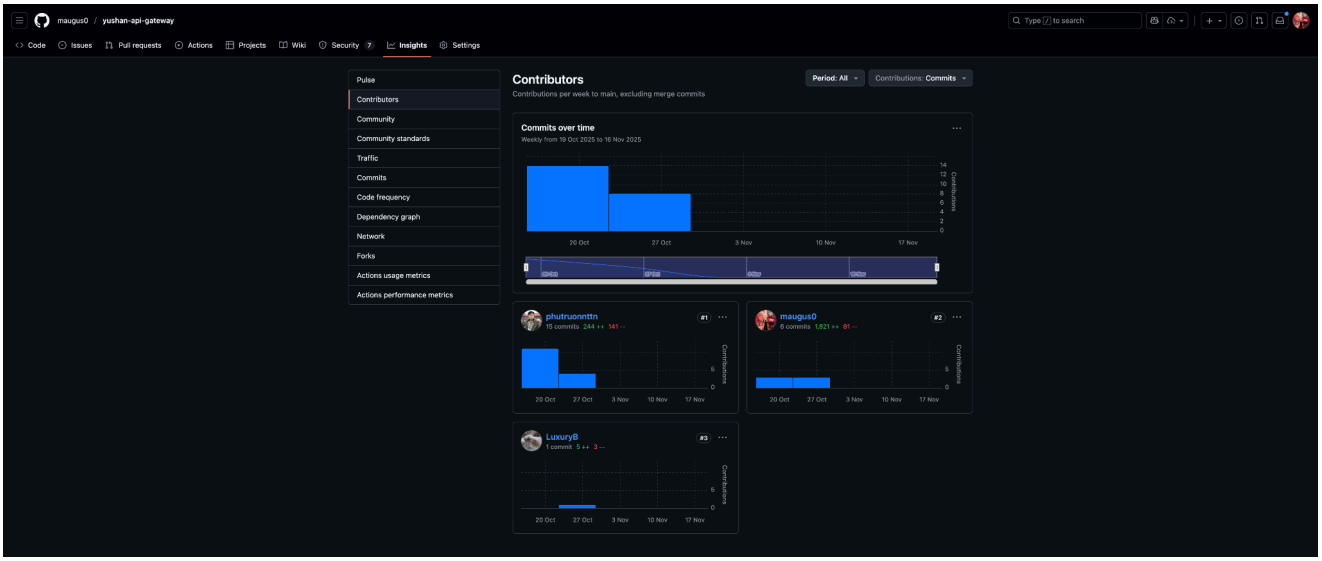
- Gamification Service



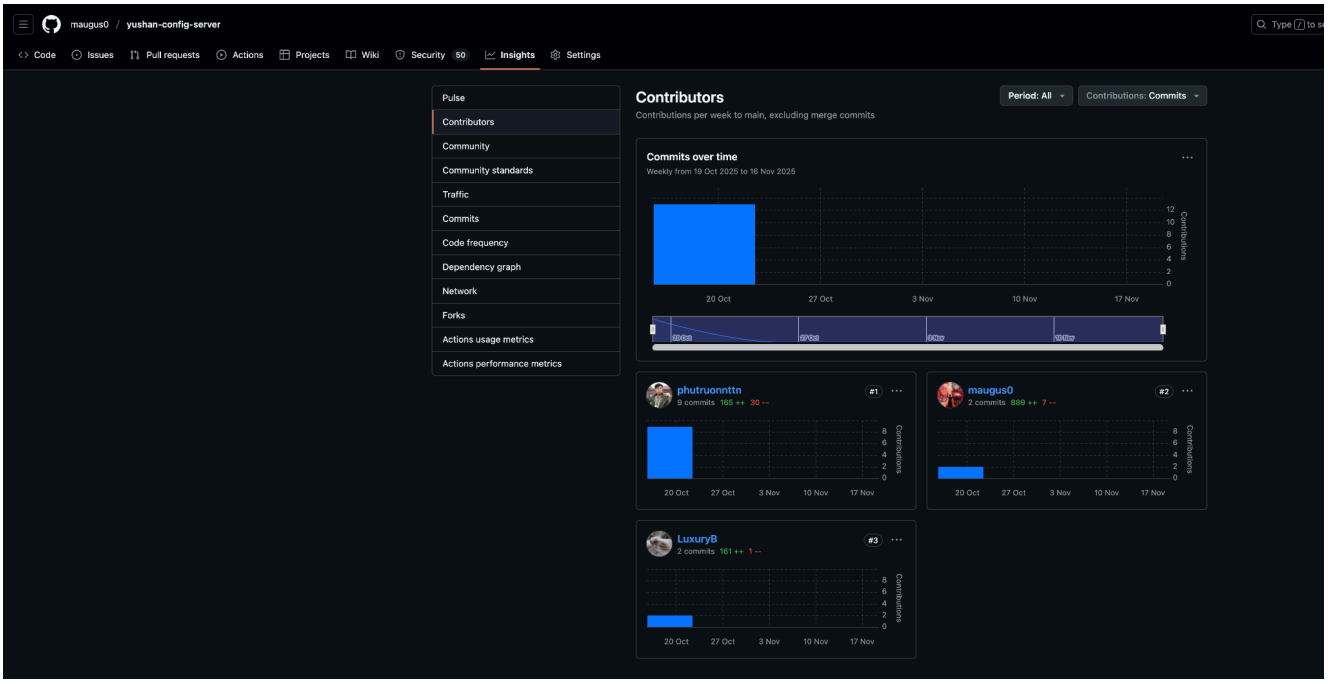
- Service Registry



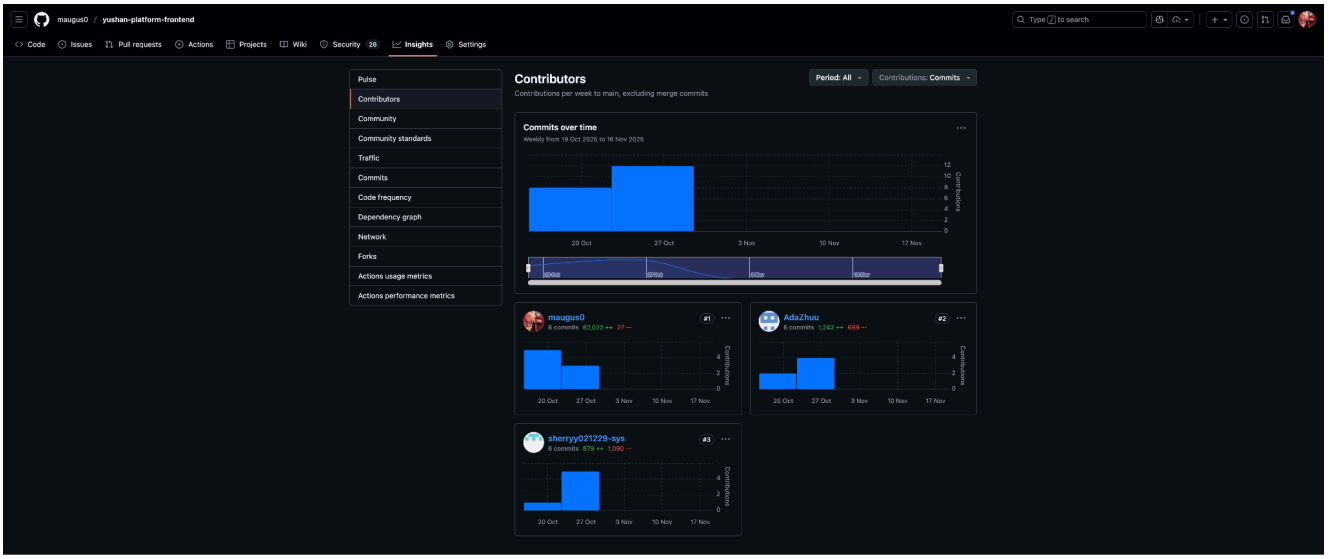
- API Gateway



- Config Server

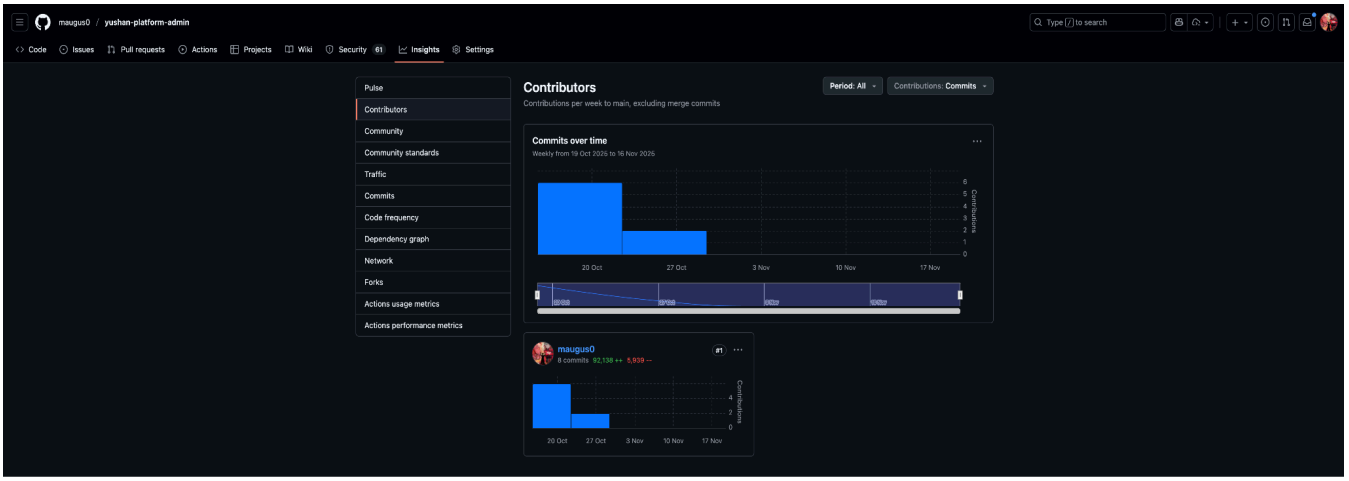


- Yushan Platform Frontend



62k lines are copied from DMSS Implementation, rest of the changes are integration with microservice APIs.

- Yushan Platform Admin



Cloned the Implementation from DMSS and adjusted for microservices by Ahan Jaiswal.

AI Declaration

The Yushan project team incorporated the use of modern AI-assisted tools, specifically **Anthropic's Claude AI (Claude Sonnet 4.5)** and other general-purpose large language models, to enhance productivity during the development, documentation, and refinement phases of this project. These tools were used selectively and responsibly, with a clear understanding of their role as supportive assistants rather than primary contributors to the system's design or implementation.

Scope of AI Assistance:
AI tools were primarily utilized for the following tasks:

- Refining technical documentation to improve clarity, coherence, and structural flow
- Generating boilerplate templates, sample code snippets, and configuration examples that were subsequently reviewed and adapted
- Providing suggestions for code optimization, error explanations, and general best practices
- Formatting reports, reorganizing content, and improving readability in project deliverables

- Supporting brainstorming sessions by offering alternative phrasing, structural outlines, and documentation improvements
- Performing grammar checks, proofreading, and helping maintain consistency across project documentation

These AI-driven contributions were supplementary in nature and designed to speed up writing tasks or provide inspiration when drafting technical content.

Important Clarifications and Boundaries:
Despite the use of AI for supportive activities, the team emphasizes that:

- **All substantial development work**, including microservice implementation, frontend/backend integration, database design, architecture decisions, and CI/CD setup, was performed entirely by human team members.
- **All testing activities**, including unit tests, integration tests, regression tests, frontend tests, E2E scripts, and non-functional testing, were manually written, executed, and validated by team members.
- AI-generated code suggestions, when referenced, served strictly as **initial scaffolds** and were not used verbatim. Every piece of code in the repositories was **written, modified, reviewed, and approved** by the development team.
- **No critical architectural, security, infrastructure, or design decision** was delegated to AI. All such decisions were made exclusively through team discussions, technical analysis, and collaborative human judgment.
- The team exercised constant caution to ensure that AI-generated content aligned with academic integrity guidelines and project evaluation standards.

Human Oversight and Validation:
Every instance of AI-generated text or code underwent thorough human verification to ensure:

- Technical accuracy
  Consistency with the project's design and requirements
- Compliance with best practices and security guidelines
- Relevance and correctness within the broader system

No unverified AI output was included in the final project deliverables. The team maintained full control and responsibility over all submitted work.

Commitment to Academic and Professional Integrity:
This declaration affirms that the core intellectual, technical, and implementation effort behind the Yushan platform remains the product of human learning, creativity, and collaboration. AI tools were used only to streamline documentation tasks, improve communication quality, and accelerate the writing process. They did **not** replace human contribution, decision-making, or problem-solving at any stage. We acknowledge the value of AI-powered tools as modern productivity enhancers while asserting that the fundamental development, testing, and architectural design of this project reflect genuine human effort and skill.

Team Members:
Ahan Jaiswal
Nguyen Phu Truong
Yang Shuang
Zhang Yan
Zhu Yuhui

Date: November 17, 2025