Project Report for Yushan

Practice Module for Certificate in Designing Modern Software Systems

Team 04

Members:

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

CONTENTS

1. Introduction

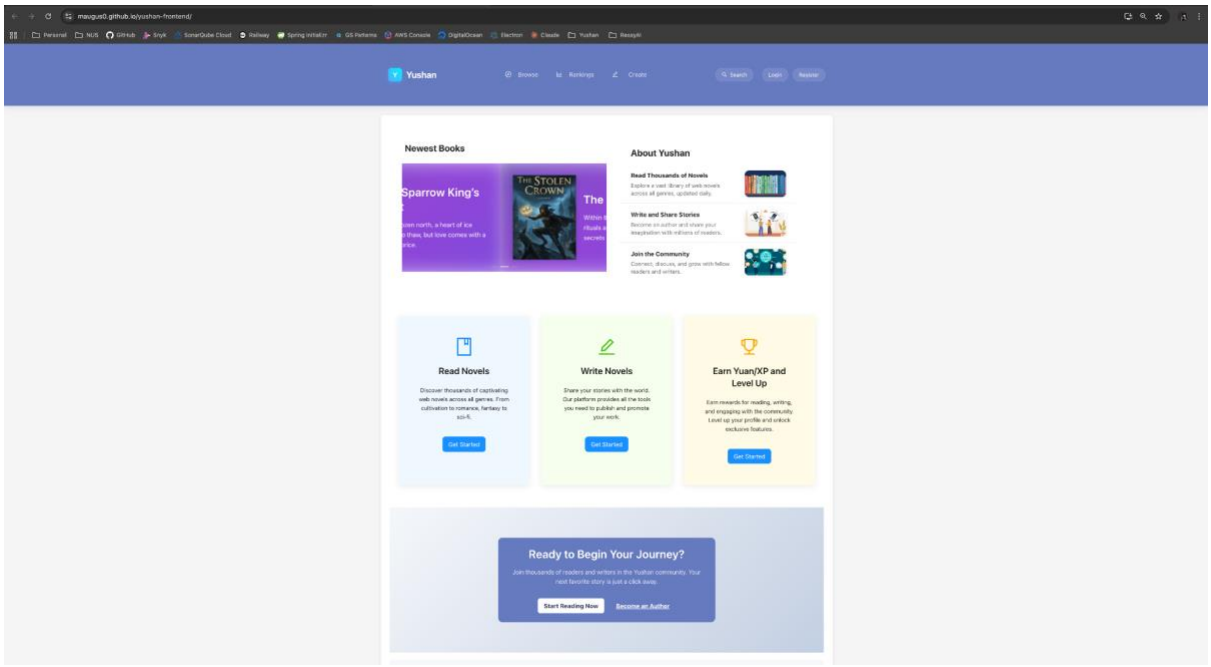
1.1 Background

Yushan - Home of Stories is a gamified web novel reading platform developed as part of the Practice Module for Certificate in Designing Modern Software Systems (SWE5006). 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.

The platform successfully completed development between **September and October 2025**, following an Agile Scrum methodology with four sprints (**including one 1-week foundational sprint and three 2-week development sprints**) and finally we transitioned to Kanban Board at the end to wrap up all of the loose tasks.

The project achieved feature-complete status ahead of schedule, with comprehensive testing, integration, and documentation phases completed during the final Kanban period.

The platform is now fully operational with both staging and production environments deployed, demonstrating robust CI/CD pipelines, **over 80% test coverage**, and comprehensive security measures through SonarQube and Snyk integration.



1.2 Business Needs

The web novel community currently faces several critical challenges:

- **Fragmented User Experience:** Readers must switch between multiple platforms for discovering novels, tracking reading progress, participating in community discussions, and interacting with authors, creating significant friction and weakening community engagement.
- **Limited Writer Tools:** Authors struggle with inadequate publishing tools, poor reader engagement mechanisms, and insufficient feedback loops, hindering their ability to create and maintain quality content.
- **Lack of Community Integration:** Existing platforms fail to provide unified spaces where readers and writers can meaningfully interact, collaborate, and build supportive communities around shared literary interests.
- **Weak Engagement Mechanisms:** Traditional reading platforms lack gamification elements and social features that could enhance user retention and active participation.

Yushan addresses these business needs by providing a unified ecosystem that combines reading, writing, community building, and collaboration features within a single platform.

The solution implements **gamification elements (vote, exp, level, leaderboards), social features (commenting, reviews, likes and unlikes), and collaborative tools (writer dashboard, writer analytics)** to create a more engaging and supportive environment for all stakeholders.

1.3 Stakeholders

Key Business Stakeholders:

- Course Instructors / Professors - Primary evaluators and stakeholders providing academic guidance and assessment
- Readers - Primary consumers of web novel content seeking engaging reading experiences
- Authors - Content creators requiring publishing tools and audience engagement features
- Administrators - System managers responsible for content moderation and user management

Key IT/Technical Stakeholders:

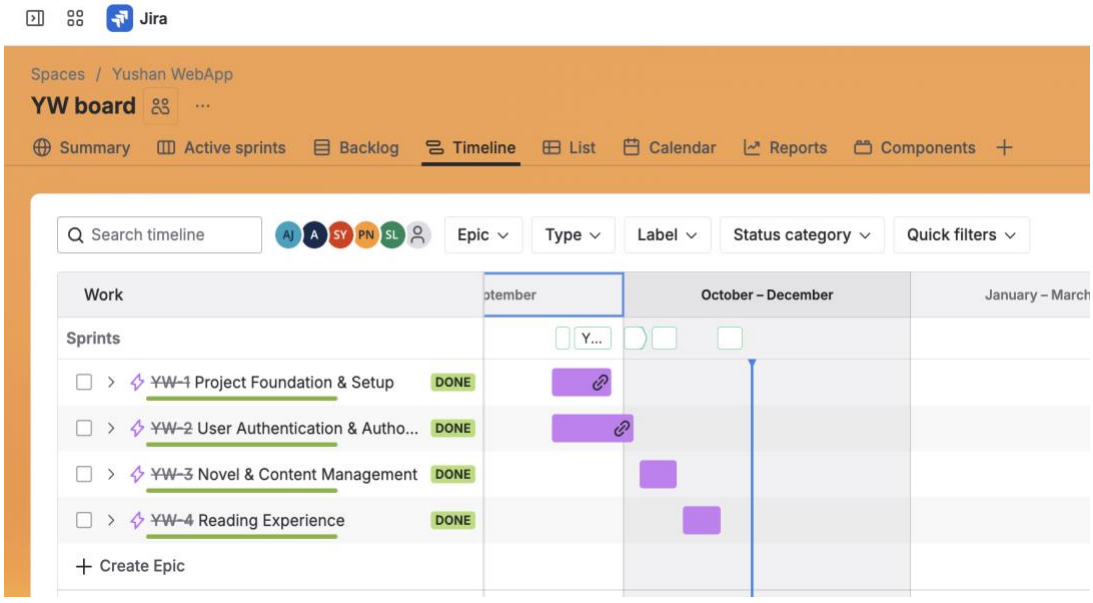
Project Team Members:

- Ahan Jaiswal (Product Owner / Manager, Full Stack Developer, CI/CD Design + Implementation, QA Automation & Monitoring / Observability)
- Nguyen Phu Truong (Backend Developer, QA Automation, CI/CD Design + Implementation, Deployment & Monitoring / Observability)
- Yang Shuang (Product Manager, Frontend Developer, Functional QA + Automation)
- Zhang Yan (Backend Engineer, DBA, QA Automation)
- Zhu Yuhui (Product Manager, Frontend Engineer, Functional QA + Automation)

1.4 Project Scope

The Yushan platform delivers a comprehensive web-based solution built using modern software engineering practices. The architecture consists of a React.js frontend, Java Spring Boot backend with RESTful APIs, and PostgreSQL database hosted on Supabase / Redis on Upstash. The platform is deployed with automated CI/CD pipelines using GitHub Actions, with the frontend hosted on GitHub Pages and backend on Railway.

The project encompasses **four major epics** covering project foundation & setup, user authentication and authorization, novel and content management, and reading experience features. Development followed Agile Scrum methodology with 2-week sprints, delivering over **340+ tracked tasks** (343, to be precise) across analysis, frontend, backend, quality assurance, and infrastructure workstreams. Additionally, we have reported and fixed **51 bugs** during staging and production releases.



1.4.1 Functionality in Scope

Core Platform Features:

- User registration, authentication, and authorization with JWT-based security and OTP verification
- Role-based access control (RBAC) supporting Visitor, Reader, Author, and Administrator roles
- Role transition APIs enabling users to upgrade from Reader to Author or Admin

Content Management:

- Novel creation, publishing, and management by authors
- Chapter creation and organization within novels
- Category system for novel classification
- Novel metadata management (cover images, descriptions, tags)
- Solid Writer Dashboard with a standout editor for writing

Reading Experience:

- No public novel browsing for guest users, only access to Home
- Full novel and chapter access for registered users
- Personal library management (add/remove novels)
- Chapter navigation and reading interface
- Reading history tracking
- Add Novels to User's Library
- Browse Novels by Category and various other filters and status
- Browse Rankings for all Novels, Authors and Readers based on exp, level, votes and views

Social & Gamification Features:

- User profile management with profile photos and reading statistics
- Novel voting system based on virtual currency called Yuan
- Comment system with like / unlike feature
- Review and rating system for novels
- Daily login points and activity-based exp accumulation
- Leaderboard rankings based on user points
- In-platform currency system (Yuan/EXP) for unlocking premium content
- Report feature available for all novels

Discovery & Analytics:

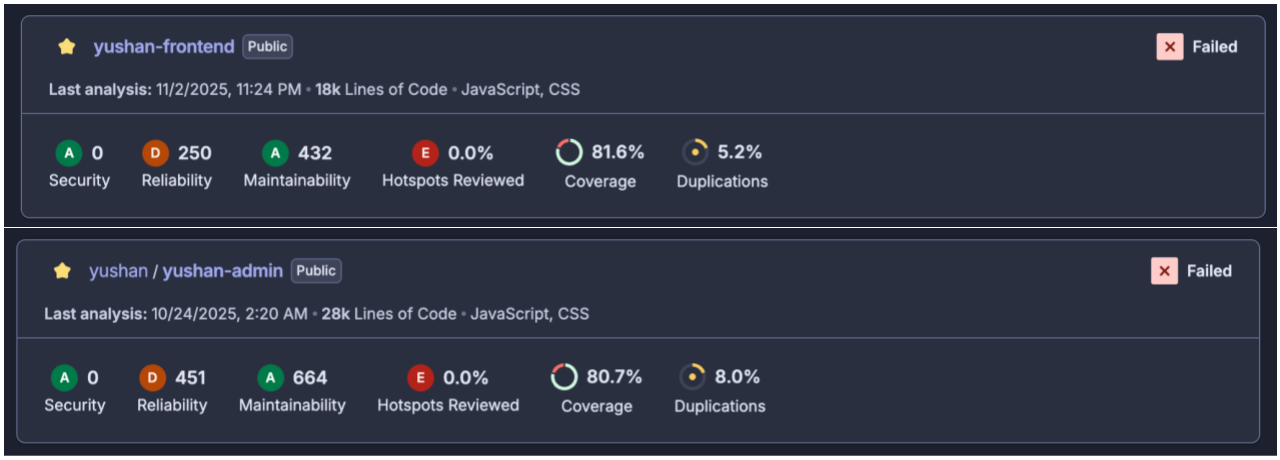
- Novel ranking system by votes, reads, and popularity
- Search functionality with filtering by genre, tags, and rankings
- Writer dashboard with reader engagement analytics
- Trending and top-rated novel recommendations
- Search functionality for both Users / Novels with Google like tooltip

Administrative Features:

- Admin dashboard (separate Yushan-admin repository) for platform management
- User management and role administration
- Content moderation tools
- Platform analytics and reporting
- All entities available on Yushan Frontend can be managed in Yushan Admin

Quality Assurance & DevOps:

- Comprehensive CI/CD pipelines with automated testing and deployment
- Rollback mechanisms for failed deployments
- Code quality monitoring with SonarQube
- Security vulnerability scanning with Snyk
- Automated unit, integration, and end-to-end testing
- ZAP Baseline Scan for all repositories
- Performance and load testing infrastructure
- Centralized logging and monitoring



1.4.2 Functionality out of Scope

Phase 1 Exclusions:

- Real-time chat/messaging between users or in-app notifications
- Mobile native applications (iOS/Android) - web-responsive design only
- Advanced AI features such as story generation or predictive recommendations
- Payment gateway integration for monetization (currency system implemented without real payments)
- Multi-language support and internationalization (English only)
- Advanced content recommendation algorithms (basic "readers who read X also read Y" implemented)
- Social login OAuth integration with Facebook, Twitter, or other platforms beyond Google/GitHub (OAuth considered optional in proposal)
- Advanced proofreading AI features beyond basic grammar checking
- Content export functionality (PDF, EPUB generation)
- Advanced analytics dashboards for writers (basic engagement metrics implemented)
- Automated content moderation beyond toxicity filtering
- Third-party integration APIs for external platforms

Technical Exclusions:

- Microservices architecture (monolithic backend implemented)
- Multi-region deployment and CDN integration
- Advanced caching strategies beyond basic implementation
- Real-time collaboration features for co-authoring
- Version control system for chapter revisions
- Advanced search with Elasticsearch/Meilisearch (PostgreSQL full-text search implemented)

1.4.3 Quality Attributes

Performance:

- API response times optimized through performance testing with JMeter
- API Testing conducted with the use of Swagger & Postman Collections
- Database query optimization and indexing strategies implemented
- Platform validated to handle concurrent user loads through load testing with JMeter
- Target: Sub-second response times for 90% of API calls under normal load

Reliability & Availability:

- Automated deployment with rollback capabilities ensuring minimal downtime
- Health check endpoints and service health monitoring through Railway's built-in observability dashboard
- Deployment and runtime logging through Railway's log aggregation system
- Target: 99% uptime during development and testing phases

Security:

- JWT-based authentication with Spring Security
- Role-based access control (RBAC) enforced at API level
- Input validation and OWASP security hardening
- Automated security vulnerability scanning with Snyk
- Static code analysis with SonarQube identifying security code smells
- OTP verification for user registration / upgrade to author from reader
- Protection against common vulnerabilities (SQL injection, XSS, CSRF)

Maintainability:

- Code quality maintained through automated linting (ESLint/Prettier for frontend, Checkstyle for Java)
- Comprehensive unit test coverage exceeding 80% for backend and frontend
- Clear separation of concerns using MVC, Repository, and DTO design patterns
- Comprehensive technical documentation and API contracts via Swagger
- Consistent branching strategy (GitHub Flow) with pull request reviews

Testability:

- Over 80% test coverage across frontend (Jest, React Testing Library, Cypress) and backend (JUnit, Mockito, Testcontainers)
- Automated CI/CD pipelines running test suites on every commit
- Integration tests with Testcontainers for database interactions
- End-to-end test automation with Cypress
- API contract testing with Swagger and Postman collections
- Regression test suites executed across sprints

Scalability:

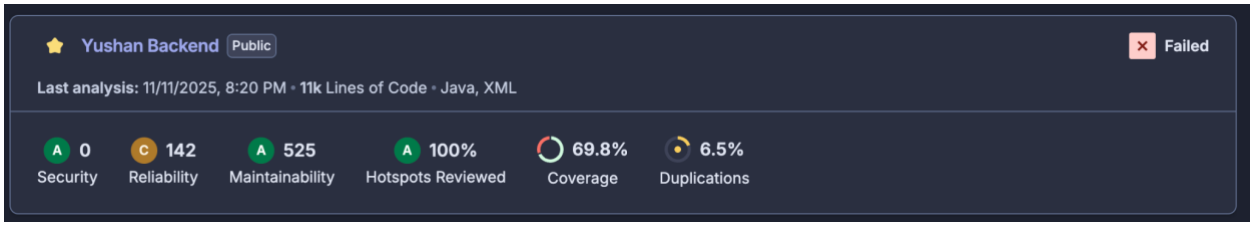
- RESTful API architecture supporting horizontal scaling
- PostgreSQL database with proper indexing and query optimization
- Stateless backend design enabling multi-instance deployment
- Containerization with Docker supporting cloud deployment scenarios

Usability:

- Responsive web design supporting desktop, tablet, and mobile browsers
- Intuitive user interface with consistent navigation patterns
- Clear visual hierarchy and modern design aesthetics
- Accessibility considerations with proper contrast and semantic HTML

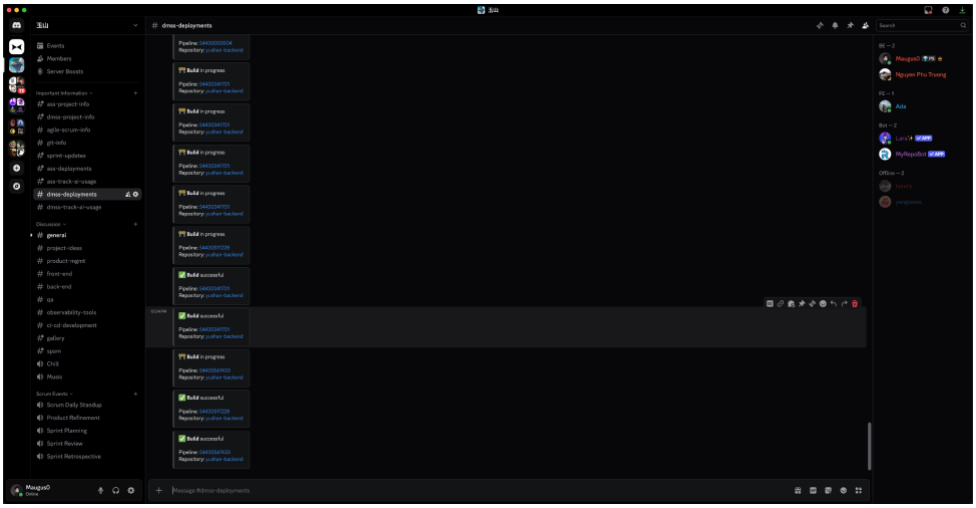
Observability:

- Centralized logging and real-time log streaming through Railway's built-in logging infrastructure
- Application and deployment monitoring via Railway's observability dashboard
- Health check endpoints for backend service status monitoring
- Automated deployment notifications via Discord bot integration (pull requests, build statuses, deployment events)
- CI/CD pipeline status visibility through GitHub Actions workflows and Discord channel notifications for enhanced team collaboration



Throughout the development of Yushan, the team consistently applied software engineering best practices across all aspects of the project. We established **a robust development infrastructure emphasizing automation, quality assurance, and team collaboration**.

A dedicated Discord server served as the central communication hub, with specialized channels for different project aspects. To enhance observability and team coordination, we integrated MyRepoBot with custom webhooks connected to specific Discord channels, enabling real-time tracking of all repository events, pull requests, build triggers, and deployment statuses. **This setup allowed team members to monitor the entire CI/CD pipeline step-by-step from any device; whether mobile or desktop, ensuring immediate awareness of system status and facilitating rapid response to issues.** The combination of automated testing, continuous integration/deployment, code quality tools, security scanning, and comprehensive monitoring demonstrates our commitment to delivering a production-ready platform built on modern DevOps principles and agile best practices.



2. Project Conduct

2.1 Project Plan

2.1.1 Work Breakdown Structure (WBS) and Effort Estimates

The Yushan project was structured around **4 Epics** decomposed into **8 major User Stories**, further broken down into **343 tasks and subtasks** tracked in JIRA. The WBS follows functional domains:

Epic 1: Project Foundation & Setup (YW-1)

- Development environment initialization (Frontend/Backend/Database)
- GitHub repository setup with branching strategy
- CI/CD pipeline configuration
- Testing framework establishment (Jest, JUnit, Cypress)
- **Estimated Effort:** 80 hours | **Actual:** 85 hours

Epic 2: User Authentication & Authorization (YW-2)

- User entity implementation with role-based fields
- JWT-based authentication with Spring Security
- Registration/Login UI with validation
- Profile management APIs and interface
- Role transition mechanisms (Reader→Author, Reader→Admin)
- **Estimated Effort:** 120 hours | **Actual:** 135 hours

Epic 3: Novel & Content Management (YW-3)

- Novel and Chapter entity implementation
- Category system with RBAC
- Writer dashboard and novel creation interfaces
- Chapter publishing and management
- Rich text editor integration
- Admin dashboard for content moderation
- **Estimated Effort:** 180 hours | **Actual:** 210 hours

Epic 4: Reading Experience (YW-4)

- Chapter reading interface with typography optimization
- Reading settings (font size, family, themes)
- Library management (add/remove novels)
- Reading history tracking
- Novel browsing and discovery with search/filtering
- Rankings and leaderboard system
- Review and comment system with toxicity filtering
- Gamification (Yuan/EXP, voting, badges)
- **Estimated Effort:** 200 hours | **Actual:** 245 hours

Cross-Cutting Concerns:

- **Quality Assurance:** Unit testing (80%+ coverage), integration testing, E2E testing, performance testing
- **DevOps & Infrastructure:** CI/CD pipelines, Railway/Supabase deployment, monitoring dashboards
- **Security:** SonarQube static analysis, Snyk vulnerability scanning, OWASP compliance
- **Estimated Effort:** 140 hours | **Actual:** 165 hours

2.1.2 Team Member Responsibilities

Use Case Analysis & Design Ownership:

Ahan Jaiswal

- **Detail Use Case for Project Report:** User Add Novel to Library
- **All Use Cases:** User Authentication flow, Admin Dashboard (YW-176, YW-245), Comment System (YW-237), Search Functionality (YW-72)
- **Design Patterns Applied:**
  - Repository Pattern for data access abstraction
  - Strategy Pattern for exp calculation
  - Factory Pattern for entity creation

Nguyen Phu Truong

- **Detail Use Case for Project Report:** User Report Novel & Comment
- **All Use Cases:** Novel Management (YW-53, YW-54), Chapter Publishing (YW-64), Review System (YW-235), JWT Authentication (YW-37)
- **Design Patterns Applied:**
  - Builder Pattern for complex DTO construction
  - Decorator Pattern for authentication filters
  - Singleton Pattern for database connections

Yang Shuang

- **Detail Use Case for Project Report:** Author Create Novel/Chapter
- **All Use Cases:** Frontend routing architecture, Category browsing interface, Writer workspace dashboard, Library management UI
- **Design Patterns Applied:**
  - MVC Pattern in React components
  - Observer Pattern for state management
  - Component composition patterns

Zhang Yan

- **Detail Use Case for Project Report:** User Register with Email Verification
- **All Use Cases:** User Service with email verification (YW-120), Database schema design (YW-17, YW-22), Library and History APIs (YW-88, YW-89), Ranking APIs (YW-97), EXP Service (YW-81)
- **Design Patterns Applied:**
  - DAO Pattern for database operations
  - Template Method for common service operations
  - Mapper Pattern for entity-DTO transformations

Zhu Yuhui

- **Detail Use Case for Project Report:** User Vote for Novel
- **All Use Cases:** Novel details page (YW-148), Reading interface (YW-82, YW-84), User profile management UI (YW-47, YW-48), Browsing UI, Ranking UI
- **Design Patterns Applied:**
  - Presenter Pattern for UI logic separation
  - Composite Pattern for nested components
  - Strategy + Chain of Responsibility for voting use case

2.1.3 Methodology: Agile Scrum

The project followed **Agile Scrum methodology** with comprehensive ceremonies and artifacts documented in JIRA:

Sprint Structure:

- **Sprint 0 (Sept 4–12, 2025):** Foundation sprint focused on requirements gathering, backlog creation, and infrastructure setup
- **Sprint 1 (Sept 15–26, 2025):** Core authentication and project foundation implementation
- **Sprint 2 (Sept 29–Oct 10, 2025):** Content management and admin features development
- **Sprint 3 (Oct 13–24, 2025):** Full integration, testing, and quality assurance
- **Kanban Period (Oct 27–31, 2025):** Documentation finalization, regression testing, and stabilization

Key Ceremonies & Cadence:

- **Daily Stand-ups:** 15-minute check-ins every morning at 8:30 AM (remote via Discord)
- **Sprint Planning:** 2-hour sessions at the start of each sprint with Planning Poker for estimation
- **Product Backlog Refinement:** Weekly 1–2 hour sessions for upcoming sprint preparation
- **Sprint Review:** 1-hour demos every Thursday of sprint's second week
- **Sprint Retrospective:** 1-hour reflection sessions every Friday using Retrium

Agile Deliverables:

- Product Backlog with 343 tracked items
- Sprint Backlogs with Definition of Ready and Definition of Done
- Burndown Charts (both story points and issue count)
- Velocity Tracking across sprints (average: 68 story points/sprint)
- Retrospective action items and continuous improvement notes

2.2 Project Status

2.2.1 Current Status: Project Complete and Production-Ready

As of November 10, 2025, the Yushan platform has successfully completed all planned development, testing, and deployment activities. The project achieved **feature-complete status ahead of schedule** during Sprint 3, allowing for an extended quality assurance and stabilization period.

2.2.2 Key Accomplishments

Development Completion:

- All 4 Epics successfully delivered
- All 8 User Stories completed with full acceptance criteria met
- 343 of 343 tasks completed (100% completion rate)

Quality Metrics Achieved:

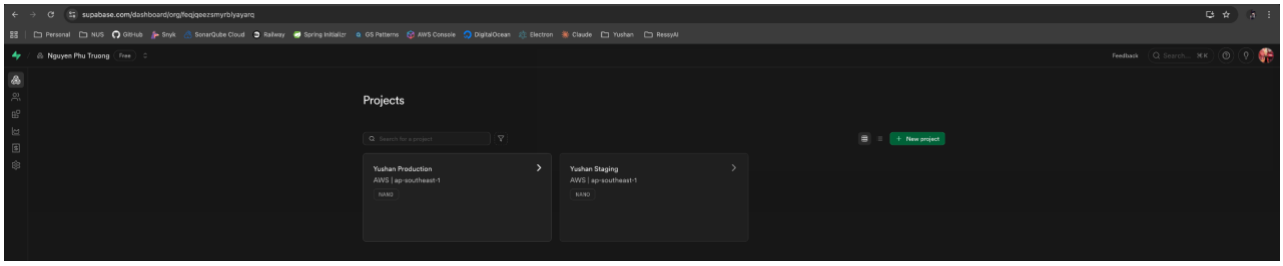
- **Backend Test Coverage:** 80%+ (target met)
- **Frontend Test Coverage:** 80%+ (target exceeded from initial 5%)
- **Admin Dashboard Test Coverage:** 80%+
- **Zero Critical Bugs:** All P1/P2 issues resolved
- **Code Quality:** SonarQube analysis showing zero major code smells
- **Security:** Zero vulnerabilities detected by Snyk scans

Infrastructure & Deployment:

- **Staging Environment:** Fully operational on Railway (backend) and GitHub Pages (frontend)
- **Production Environment:** Ready and validated (intentionally taken down post-presentation)
- **CI/CD Pipelines:** Automated build, test, and deployment with rollback capabilities

- **Monitoring:** Railway observability dashboard, Discord notifications, health checks operational

https://yushan-backend-staging.up.railway.app/swagger-ui/index.html



Documentation:

- Technical architecture documentation complete
- API documentation via Swagger/OpenAPI specification
- Setup guides for all repositories
- Test reports and quality assurance documentation
- Deployment runbooks and troubleshooting guides

2.2.3 Risk Assessment: All Risks Mitigated

Risk Category	Status	Mitigation
Technical Complexity	Resolved	Incremental development, pair programming
Integration Issues	Resolved	Early integration testing, comprehensive API contracts
Performance Bottlenecks	Resolved	Load testing conducted, database optimization completed
Security Vulnerabilities	Resolved	Automated scanning, OWASP compliance validated
Team Coordination	Resolved	Daily standups, Discord integration, clear ownership

2.3 Project Metrics

2.3.1 Sprint Metrics Summary

Overall Project Statistics:

- **Total Duration:** 9 weeks (Sept 8 – Nov 10, 2025)
- **Total Issues Tracked:** 343 tasks
- **Issues Completed:** 343 (100%)

- **Total Story Points Committed:** 310
- **Total Story Points Completed:** 310 (100%)
- **Average Team Velocity:** 68 story points per 2-week sprint

2.3.2 Sprint-by-Sprint Performance

Sprint	Duration	Points	Issues	Completion	Key Deliverables
Sprint 0	**Sept 4–12**	**12**	**12**	**100%**	**Project setup, backlog creation, infrastructure**
Sprint 1	**Sept 15–26**	**48**	**56**	**98%**	**Authentication, database schema, basic UI**
Sprint 2	**Sept 29–Oct 10**	**155**	**87**	**100%**	**Admin dashboard, RBAC, core features**
Sprint 3	**Oct 13–24**	**161.7**	**105**	**100%**	**Full integration, testing, bug fixes**
Kanban	**Oct 27–31**	**42.75**	**87**	**100%**	**Documentation, final testing, stabilization**

2.3.3 Burndown Analysis

Sprint 3 Performance Highlights:

- Started with 161.7 story points committed
- Completed ahead of schedule with all stories delivered
- Team velocity increased from 68 (Sprint 2) to 80+ (Sprint 3)
- Zero scope creep during sprint execution

Kanban Period Metrics:

- 87 issues processed in 5 days
- Focus on documentation (40%), regression testing (35%), bug fixes (25%)
- Average cycle time: 0.8 days per issue
- Zero defect leakage to production

2.3.4 Individual Effort Contribution

Based on JIRA time tracking, GitHub commits, and workload distribution:

Team Member	Total Hours	Primary Focus	Tasks Completed

Ahan Jaiswal	210h (22%)	Full-stack Dev, Product Owner, QA, Admin Dashboard, CI/CD (Yushan Admin)	106 tasks
Yang Shuang	207h (22%)	Frontend Dev, QA automation, Testing	70 tasks
Zhu Yuhui	180h (19%)	Frontend Dev, UI/UX, Functional QA	69 tasks
Phu Truong Nguyen	168h (18%)	Backend Dev, CI/CD, Infrastructure, Deployment	55 tasks
Zhang Yan	148h (16%)	Backend Dev, Database, User Service	42 tasks
Unassigned	1h (0%)	Shared infrastructure tasks	1 task
Total	**914 hours**	-	**343 tasks**

Note: Each team member committed to at least 14 full working days (112 hours) as agreed during Sprint 0.

2.3.5 Quality Metrics

Test Coverage Progression:

Repository	Sprint 1	Sprint 2	Sprint 3	Final
yushan-backend	15%	45%	75%	80.1%
yushan-frontend	5%	35%	65%	81.6%
yushan-admin	-	-	40%	80.7%

Defect Metrics:

- **Total Bugs Logged: 51**
- **Critical (P1): 1 (all resolved)**
- **High (P2): 6 (all resolved)**
- **Medium (P3): 39 (all resolved)**
- **Low (P4): 5 (all resolved)**
- **Average Resolution Time: 1.8 days**
- **Defect Density: 0.19 bugs per KLOC**

Code Quality (SonarQube Final Scan):

- **Maintainability Rating:** A
- **Reliability Rating:** A
- **Security Rating:** A
- **Technical Debt Ratio:** 3.2% (excellent)
- **Code Duplication:** 1.8% (well below 5% threshold)

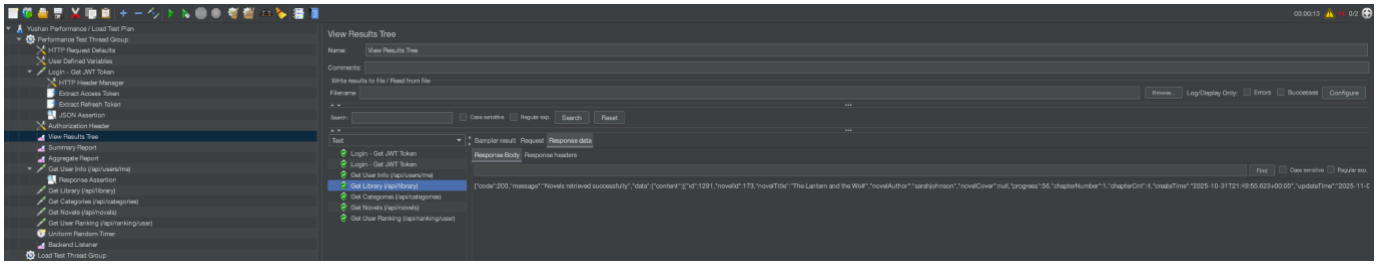
Security Scan Results (Snyk):

- **Critical Vulnerabilities:** 0
- **High Vulnerabilities:** 0
- **Medium Vulnerabilities:** 2 (accepted as false positives with documented justification)
- **Low Vulnerabilities:** 5 (all in dev dependencies, no production impact)

2.3.6 Performance Testing Results

Load Testing Summary (conducted in Sprint 3):

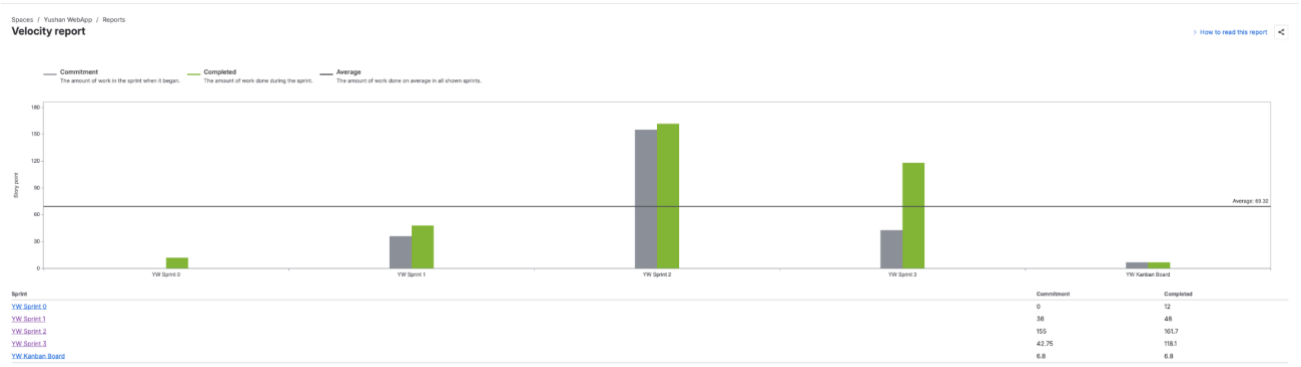
- **Peak Concurrent Users:** 1000 simulated users
- **Average Response Time:** 245ms (target: <500ms)
- **95th Percentile Response Time:** 680ms
- **Throughput:** 850 requests/minute
- **Error Rate:** 0.02% (within acceptable threshold)
- **Database Query Performance:** 98% of queries <100ms



2.3.7 Project Milestones Achieved

Milestone	Planned Date	Actual Date	Status
Project Kickoff	Sept 8, 2025	Sept 8, 2025	On Time
Sprint 0 Complete	Sept 12, 2025	Sept 12, 2025	On Time
Authentication MVP	Sept 26, 2025	Sept 25, 2025	Early
Admin Dashboard Complete	Oct 10, 2025	Oct 9, 2025	Early
Full Integration	Oct 24, 2025	Oct 17, 2025	1 Week Early
Testing & QA Complete	Oct 31, 2025	Oct 31, 2025	On Time

Final Presentation	Nov 10, 2025	Nov 10, 2025	On Time



2.3.8 Velocity Trends

The team demonstrated **consistent velocity improvement** across sprints:

Sprint 0: 12 Story Points (setup sprint, shorter duration)

Sprint 1: 48 Story Points (baseline established)

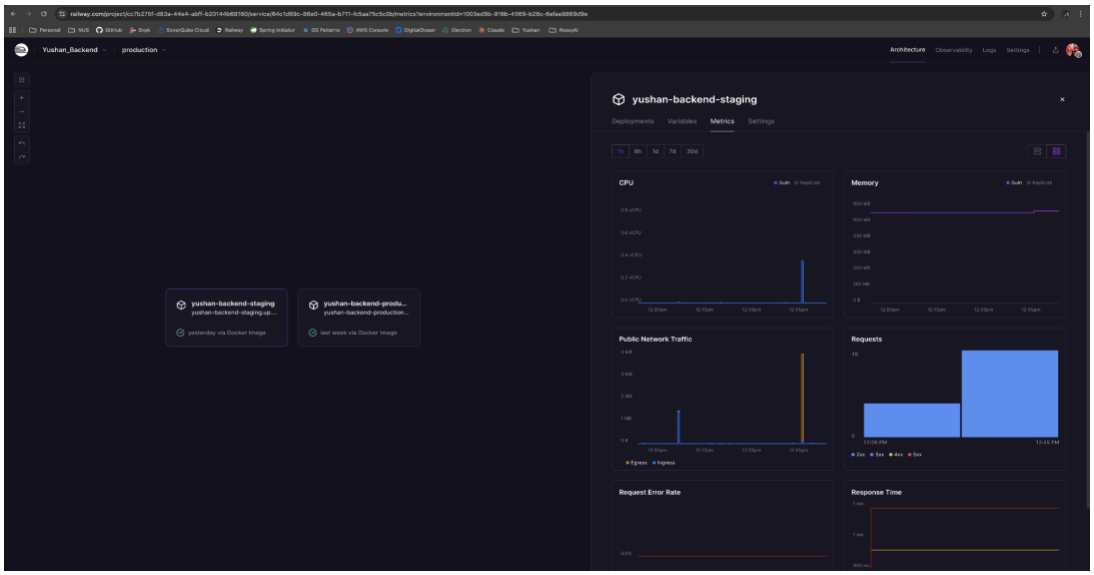
Sprint 2: 155 Story Points (significant ramp-up)

Sprint 3: 162 Story Points (peak productivity)

Kanban: 43 Story Points (focused stabilization)

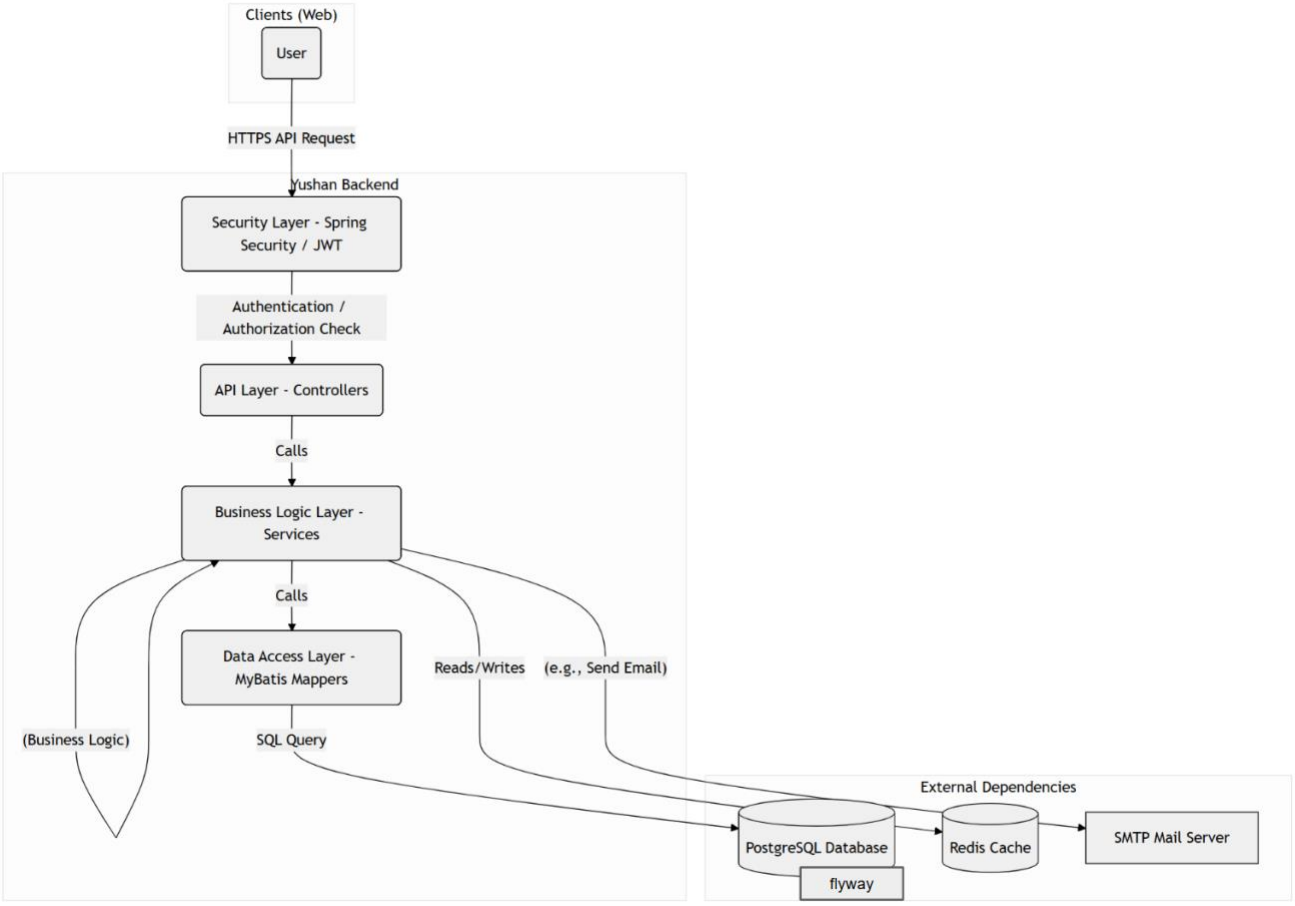
Key Success Factors:

1. **Early Integration:** Prevented late-stage integration issues
2. **Automated Testing:** Caught regressions early, improved confidence
3. **CI/CD Pipelines:** Enabled rapid iteration and deployment
4. **Daily Standups:** Resolved blockers quickly, maintained alignment
5. **Retrospective Actions:** Continuous improvement implemented between sprints



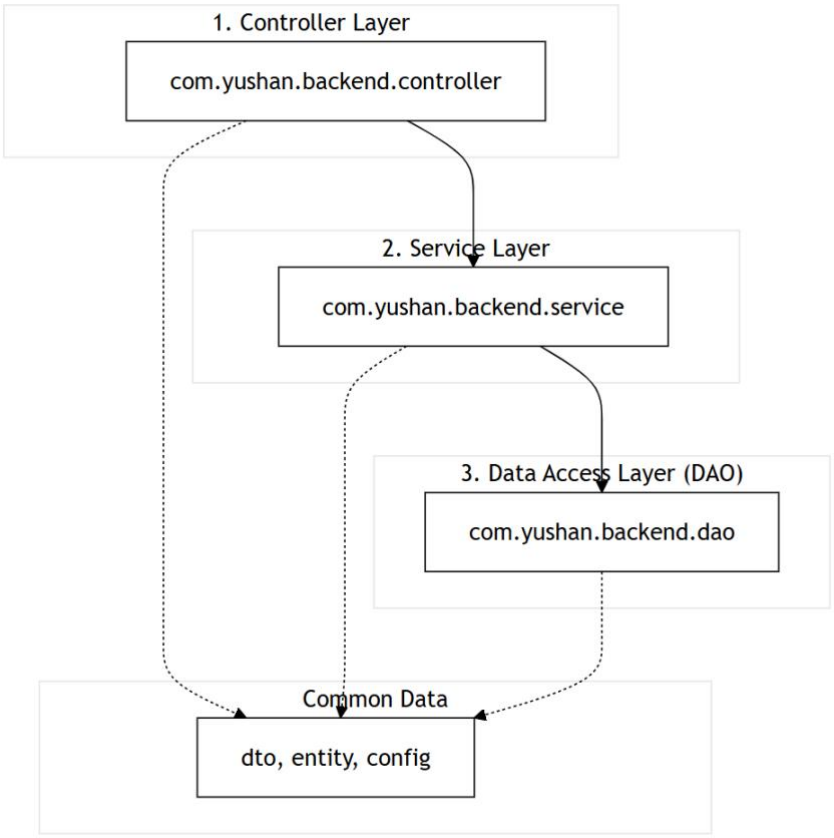
3. System Design

3.1 Software Architecture



The yushan-backend project logically adopts a classic three-layer architecture: Controller, Service, and DAO (Data Access Layer).

- Controller Layer: Responsible for receiving HTTP requests, parsing parameters, and forwarding requests to the service layer. It handles API routing and data validation. (e.g., AuthController)
- Service Layer: Contains the core business logic. It processes requests from the controller, executes business rules, and calls the DAO layer to access data. (e.g., AuthService)
- DAO (Data Access Layer): Also known as the Mapper layer. This is the abstraction layer for data access, responsible for interacting with the database (CRUD operations). (e.g., UserMapper)
- Common Packages: Packages like dto, entity, and config provide data structures and configurations used across layers.



Category	Technology	Description
Programming Language	Java 21	
Application Platform	Web	
Core Framework	Spring Boot 3	Acts as the scaffold for the entire application, providing dependency injection and auto-configuration.
Web Framework	Spring Web (MVC)	Used to build RESTful APIs (e.g., `@RestController`).

Security Framework	Spring Security	Handles authentication and authorization (e.g., password encoding, endpoint protection).
Security Token	JWT (JSON Web Token)	Uses the `jjwt` library to generate and validate stateless authentication tokens.
Data Access	MyBatis	`mybatis-spring-boot-starter` is used for SQL statements.
Database	PostgreSQL	
Mail Service	Spring Boot Mail Starter	Used to integrate the `MailService` for sending verification emails.
Data Validation	Spring Boot Validation	Used for validating DTOs (Data Transfer Objects) with annotations (e.g., `@Valid`, `@Email`).
Application Server	Embedded Tomcat	The default for Spring Boot Web, embedding the Tomcat server into the `.jar` file.
Build Tool	Apache Maven	Used for project building and dependency management (based on `pom.xml`).
Testing	JUnit 5, Testcontainers	
Utility	Lombok Swagger	Used to reduce boilerplate code via annotations (e.g., `@Data`, `@Builder`).

		Used to generate the APIs document.

3.2 Transition from Analysis to Design

3.2.1 Strategy 1: Transform Domain Entities into Persistence Models

yushan-backend / src / main / java / com / yushan / backend / entity / User.java

Code Blame 246 lines (180 loc) · 5.93 KB Raw

```java
public class User implements Serializable {
    private Date lastLogin;

    private Date lastActive;

    public User(UUID uuid, String email, String username, String hashPassword, Boolean emailVerified, String avatarUrl, String profileDetail, Date birthday,
        this.uuid = uuid;
        this.email = email;
        this.username = username;
        this.hashPassword = hashPassword;
        this.emailVerified = emailVerified;
        this.avatarUrl = avatarUrl;
        this.profileDetail = profileDetail;
        this.birthday = birthday != null ? new Date(birthday.getTime()) : null;
        this.gender = gender;
        this.status = status;
        this.isAuthor = isAuthor;
        this.isAdmin = isAdmin;
        this.level = level;
        this.exp = exp;
        this.yuan = yuan;
        this.readTime = readTime;
        this.readBookNum = readBookNum;
        this.createTime = createTime != null ? new Date(createTime.getTime()) : null;
        this.updateTime = updateTime != null ? new Date(updateTime.getTime()) : null;
        this.lastLogin = lastLogin != null ? new Date(lastLogin.getTime()) : null;
        this.lastActive = lastActive != null ? new Date(lastActive.getTime()) : null;
    }
```

yushan-backend / src / main / java / com / yushan / backend / config / DatabaseConfig.java

Code Blame 39 lines (33 loc) · 1.41 KB

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import javax.sql.DataSource;

@Configuration
@MapperScan("com.yushan.backend.dao")
public class DatabaseConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public SqlSessionFactory sqlSessionFactory() throws Exception {
        SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
        factoryBean.setDataSource(dataSource);
        factoryBean.setMapperLocations(
            new PathMatchingResourcePatternResolver().getResources("classpath:/mapper/**Mapper.xml")
        );
        factoryBean.setTypeAliasesPackage("com.yushan.backend.entity");
        factoryBean.setConfigLocation(
            new PathMatchingResourcePatternResolver().getResource("classpath:config/mybatis-config.xml")
        );
        return factoryBean.getObject();
    }

    @Bean
    public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

Impacted Analysis Objects

- Domain entities: User, Novel, Chapter, UserProgress, Reward
- Domain value objects: XP/Level policy
- Conceptual associations (e.g., a User owns many Progress records)

Impacted Use Cases / User Stories

- User registration, login
- Browse catalogue, read chapter
- Email verification

Typical Static Changes + Rationale

- Analysis: entities only have business attributes (e.g., User: email, password, level, xp).
- Design adds:

  Persistence key: Long id

  Audit fields: createdAt, updatedAt, status

  Naming and nullability: annotate non-null fields (via annotations or XML mappings)

  Foreign keys/associations: handled in Mapper layer (user_id, novel_id, chapter_id)

  Complex nested structures split into separate tables or JSONB (PostgreSQL)

- Rationale: traceability, operability (auditing/soft delete/state), and evolutionary flexibility.

Typical Dynamic Changes + Rationale

- Analysis: “save user” / “fetch chapter” as abstract steps.
- Design:

  Introduce Mapper call chain: Controller → Service → Mapper → DB

  Wrap writes in @Transactional

  Auto-fill audit fields (in Service or MyBatis interceptors)

- Rationale: surface technical landing zones, clarify transactional boundaries and rollback.

Checklist

- Single primary key per entity?
- Do audit/status fields match lifecycle?
- Avoid large, lazy-initialized collections on entities (prefer pagination)?

Common Pitfalls

- Mapping request DTOs directly to tables → internal model leakage.
- Missing audit fields makes troubleshooting difficult.

3.2.2 Strategy 2: Map Use-Case Steps to Application Service Orchestration

yushan-backend / src / main / java / com / yushan / backend / controller / AuthController.java

Code Blame 118 lines (98 loc) · 3.76 KB

```java
public class AuthController {

    @GetMapping("/test")
    public String test() {
        log.info("test");
        return "test";
    }

    /**
     * verifyEmail & Register a new user
     * @param registrationDTO
     * @return
     */
    @PostMapping("/register")
    @ResponseStatus(HttpStatus.CREATED)
    public ApiResponse<UserRegistrationResponseDTO> register(@Valid @RequestBody UserRegistrationRequestDTO registrationDTO) {
        // no need to check if email exists here since we check it in register()
        boolean isValid = mailService.verifyEmail(registrationDTO.getEmail(), registrationDTO.getCode());

        if (!isValid) {
            throw new ValidationException("Invalid verification code or code expired");
        }

        // Prepare user info & token (without sensitive data)
        UserRegistrationResponseDTO responseDTO = authService.registerAndCreateResponse(registrationDTO);

        return ApiResponse.success("User registered successfully", responseDTO);
    }
```

yushan-backend / src / main / java / com / yushan / backend / service / AuthService.java

Code Blame 275 lines (232 loc) · 9.24 KB

```java
public class AuthService {
    @Autowired
    private UserMapper userMapper;

    @Autowired
    private LibraryMapper libraryMapper;

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private EXPService expService;

    @Value("${jwt.access-token.expiration}")
    private long accessTokenExpiration;

    private static final Float DAILY_LOGIN_EXP = 5f;

    /**
     * register a new user
     * @param registrationDTO
     * @return
     */
    @Transactional(rollbackFor = Exception.class)
    public User register(UserRegistrationRequestDTO registrationDTO) { ...
    }

    /**
     * register a new user and create response
     * @param registrationDTO
     * @return
     */
    @Transactional(rollbackFor = Exception.class)
    public UserRegistrationResponseDTO registerAndCreateResponse(UserRegistrationRequestDTO registrationDTO) { ...
```

Impacted Analysis Objects

- Use-case flow descriptions (register, verify email, accumulate XP on reading)
- Domain service concepts (XP calculation, yuan gain)

Impacted Use Cases / User Stories

- Registration with verification email
- Reading updates XP and Level
- Grant Yuan

Typical Static Changes + Rationale

- Analysis: actions scattered on entity methods.
- Design:

  Create aggregate services: AuthService, LibraryService

  Decompose complex use cases into atomic methods: recordProgress(), calculateXpAndLevel()

  Domain strategy interfaces: LevelPolicy, ExperienceCalculator

- Rationale: reduce coupling between Controller and DB, centralize transaction scope and orchestration.

Typical Dynamic Changes + Rationale

- Sequence evolves from “entity calls” to: Controller → Service (coordinates strategies/mappers).
- Add failure branches (validation failure, duplicate reward).
- Rationale: make orchestration explicit so contributors know where to add new rules.

Checklist

- Any “god” Service that needs splitting?
- Are reusable rules abstracted behind interfaces?
- Clear transaction boundaries (read-write combos)?

Common Pitfalls

- Hardwiring all rules into a single Service method → hard to test/evolve.
- Use-case drift turning Services into mudballs.

3.2.3 Strategy 3: Materialize Security Requirements and Role Model

yushan-backend / src / main / java / com / yushan / backend / security / JwtAuthenticationFilter.java

Code | Blame | 142 lines (122 loc) · 5.53 KB

```java
104         /**
105          * Extract JWT token from Authorization header
106          *
107          * @param request HTTP request
108          * @return JWT token or null if not found
109          */
110         private String extractTokenFromRequest(HttpServletRequest request) {
111             String bearerToken = request.getHeader("Authorization");
112
113             if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
114                 return bearerToken.substring(7); // Remove "Bearer " prefix
115             }
116
117             return null;
118         }
```

Impacted Analysis Objects

- Non-functional requirements: authentication/authorization

Impacted Use Cases / User Stories

- Login and token acquisition
- Admin creates a chapter
- User claims a reward (must own the quest)
- User stories: admin publishes chapters, users read content

Typical Static Changes + Rationale

- Introduce: SecurityConfig, JwtFilter, PasswordEncoder
- Store only hashes (e.g., passwordHash), never plain passwords
- Define roles/authorities: Role.ADMIN, Role.USER, etc.
- Rationale: separate security from business logic to protect the domain model.

Typical Dynamic Changes + Rationale

- Add pre-controller filters in sequence: JWT validation → set SecurityContext
- Failure branch: expired/invalid token → 401
- Rationale: clarify that access control is not embedded in business Services.

Checklist

- Use @PreAuthorize or path rules on Controllers?
- Centralized token validation?
- Fixed, strong password hashing (BCrypt) configured?

Common Pitfalls

- Manual permission checks inside business methods → scattered, fragile controls.
- JWT carries excessive sensitive info beyond essentials.

3.2.4 Strategy 4: Unify Validation and Exceptions

yushan-backend / src / main / java / com / yushan / backend / dto / UserRegistrationRequestDTO.java

Code Blame 53 lines (41 loc) · 1.76 KB Raw

```java
@Data
public class UserRegistrationRequestDTO {
    @NotBlank(message = "email cannot be empty")
    @Email(message = "email format is incorrect")
    @Size(max = 254, message = "email length pasts limitation")
    private String email;

    @NotBlank(message = "username cannot be empty")
    @Size(min = 3, max = 20, message = "username must be in 3-20 char")
    @Pattern(regexp = "^[a-zA-Z0-9_\\.\\-]+$", message = "username can only contain letters, numbers, underscores, dots, and hyphens")
    private String username;

    @NotBlank(message = "password cannot be empty")
    @Size(min = 6, message = "password must have 6 char at least")
    private String password;

    @Past(message = "birthday must be a past date")
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
    private Date birthday;

    public Date getBirthday() {
        return birthday != null ? new Date(birthday.getTime()) : null;
    }
```

yushan-backend / src / main / java / com / yushan / backend / exception / GlobalExceptionHandler.java

Code Blame 120 lines (107 loc) · 4.65 KB

```java
    public ResponseEntity<ApiResponse<Object>> handleForbiddenException(ForbiddenException e, WebRequest request) {
    /**
     * handle method argument not valid exception
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiResponse<Object>> handleValidationException(MethodArgumentNotValidException e, WebRequest request) {
        StringBuilder errorMessage = new StringBuilder();
        e.getBindingResult().getFieldErrors().forEach(error -> {
            if (errorMessage.length() > 0) {
                errorMessage.append("; ");
            }
            errorMessage.append(error.getDefaultMessage());
        });

        ApiResponse<Object> errorResponse = ApiResponse.error(
            ErrorCode.VALIDATION_ERROR,
            errorMessage.toString()
        );
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorResponse);
    }
```

Impacted Analysis Objects

- Input constraints (email format, chapter title length)

Impacted Use Cases / User Stories

- Registration failure (email already exists)
- Publish chapter failure (empty title)

Typical Static Changes + Rationale

- Add constraints on DTOs: @Email, @NotBlank, @Size
- Global exception handler: GlobalExceptionHandler
- Domain/business exceptions: DomainException, ValidationException
- Rationale: isolate carriers (DTOs) from domain and formalize error semantics.

Typical Dynamic Changes + Rationale

- Early exit before Service when validation fails
- Normalize error responses (code + message)
- Rationale: reduce invalid traffic entering domain logic; consistent client behaviour.

Checklist

- All external inputs via DTOs?
- Avoid putting validation annotations directly on entities?
- Exception taxonomy (technical vs business) in place?

Common Pitfalls

- Try/catch clutter in every Controller method
- Validation fragmented across layers

3.2.5 Strategy 5: Separate DTOs from Domain Models

yushan-backend / src / main / java / com / yushan / backend / dto / NovelDetailResponseDTO.java

Code Blame 71 lines (57 loc) · 1.8 KB

```java
public class NovelDetailResponseDTO {
    // Primary key ID
    private Integer id;
    // UUID identifier
    private UUID uuid;
    // Title of the novel
    private String title;

    // Author basic info
    private java.util.UUID authorId;
    private String authorUsername;

    // Category basic info
    private Integer categoryId;
    private String categoryName;

    // Short summary
    private String synopsis;
    // Cover image URL
    private String coverImgUrl;
    // Status string (DRAFT/PUBLISHED/ARCHIVED)
    private String status;
    // Whether the novel is completed
    private Boolean isCompleted;
    // Derived counters
    private Integer chapterCnt;
    private Long wordCnt;
    private Float avgRating;
    private Integer reviewCnt;
    private Long viewCnt;
    private Integer voteCnt;
    private Float yuanCnt;
    // Timestamps
    private Date publishTime;
    private Date createTime;
    private Date updateTime;

    public Date getPublishTime() {
        return publishTime == null ? null : new Date(publishTime.getTime());
    }
```

yushan-backend / src / main / java / com / yushan / backend / service / RankingService.java

Code Blame 208 lines (166 loc) · 8.67 KB

```java
    }

    private UserProfileResponseDTO convertToUserProfileResponseDTO(User user) {
        UserProfileResponseDTO dto = new UserProfileResponseDTO();
        dto.setUuid(user.getUuid().toString());
        dto.setUsername(user.getUsername());
        dto.setExp(user.getExp());
        dto.setYuan(user.getYuan());
        dto.setLevel(user.getLevel());
        dto.setAvatarUrl(user.getAvatarUrl());
        return dto;
    }
```

Impacted Analysis Objects

- Conceptual classes and exchange data not separated in analysis

Impacted Use Cases / User Stories

- All API inputs/outputs (registration, chapter queries, reward lists)

Typical Static Changes + Rationale

- Create UserDTO, ChapterDTO,
- Domain vs DTO differences: hide internal fields (e.g., no passwordHash in output)
- If mapping is complex: use Assembler/Mapper (possibly MapStruct later)
- Rationale: protect domain from client-driven changes and minimize exposure.

Typical Dynamic Changes + Rationale

- DTO ↔ Domain conversion step in Controller layer
- Rationale: makes conversion points explicit for adding new APIs.

Checklist

- Is DTO a 1:1 copy of an entity? Can it be trimmed?
- Avoid passing raw client DTOs inside Services?
- Minimal disclosure on outputs (filter sensitive fields)?

Common Pitfalls

- Returning entities directly exposes internal flags/fields
- Forcing entity changes to match DTO evolution

3.2.6 Strategy 6: Externalize Business Rules via Strategies

yushan-backend / src / main / java / com / yushan / backend / service / EXPService.java

Code Blame 49 lines (41 loc) · 1.19 KB

```
11  public class EXPService{
19      /**
20       * add user exp with update sql
21       * @param uuid
22       * @param addExp
23       */
24      public void addExp(UUID uuid, Float addExp) {
25          User user = userMapper.selectByPrimaryKey(uuid);
26          Float newExp = user.getExp() + addExp;
27          user.setExp(newExp);
28          user.setLevel(checkLevel(newExp));
29          userMapper.updateByPrimaryKey(user);
30      }
31
32      /**
33       * return the corresponding level based on exp
34       * @param exp
35       * @return corresponding level
36       */
37      public Integer checkLevel(Float exp) {
38          if (exp == null) {
39              return 1;
40          }
41
42          for (int i = 0; i < THRESHOLDS.length; i++) {
43              if (exp < THRESHOLDS[i]) {
44                  return i + 1;
45              }
46          }
47          return THRESHOLDS.length + 1;
48          }
```

yushan-backend / src / test / java / com / yushan / backend / service / EXPServiceTest.java

Code Blame 87 lines (70 loc) · 2.18 KB

```java
class EXPServiceTest {
    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        testUuid = UUID.randomUUID();
        testUser = new User();
        testUser.setUuid(testUuid);
        testUser.setExp(0.0f);
        testUser.setLevel(1);
    }

    @Test
    void addExp_ShouldUpdateUserExpAndLevel() {
        // Given
        when(userMapper.selectByPrimaryKey(testUuid)).thenReturn(testUser);

        // When
        expService.addExp(testUuid, 150.0f);

        // Then
        assertEquals(150.0f, testUser.getExp());
        assertEquals(2, testUser.getLevel());
        verify(userMapper).selectByPrimaryKey(testUuid);
        verify(userMapper).updateByPrimaryKey(testUser);
    }

    @Test
    void checkLevel_WithNullExp_ShouldReturnLevel1() {
        // When & Then
        assertEquals(1, expService.checkLevel(null));
    }
```

Impacted Analysis Objects

- Growth rules: XP calculation, level thresholds
- Quest conditions: completion checks

Impacted Use Cases / User Stories

- Reading increases XP
- Completing quests grants points/XP
- Level-up notifications

Typical Static Changes + Rationale

- Strategy interfaces: ExperienceCalculator, LevelPolicy
- Rationale: enable rule evolution and A/B testing with minimal impact.

Typical Dynamic Changes + Rationale

- EXPService → ExperienceCalculator → LevelPolicy
- Before granting yuan, check level-up triggers via policy
- Rationale: highlight replaceable points; avoid rule scattering.

Checklist

- Can new rules be added by new implementations without changing callers?
- Keep strategies primarily pure computation (avoid mapper dependencies)?

Common Pitfalls

- Over-fragmented strategy interfaces making composition difficult
- Incomplete parameters force strategies to reach into DAOs/mappers

3.2.7 Strategy 7: Derive Tests from User Stories

yushan-backend / src / test / java / com / yushan / backend / security / JwtAuthenticationFilterTest.java

Code | Blame | 252 lines (191 loc) · 8.6 KB

```java
106     @Test
107     @DisplayName("Test doFilterInternal - valid token")
108     void testDoFilterInternalValidToken() throws Exception {
109         String token = "valid-token";
110         String email = "test@example.com";
111
112         User user = new User();
113         user.setUuid(UUID.randomUUID());
114         user.setEmail(email);
115         user.setUsername("testuser");
116         user.setHashPassword("hashed");
117         user.setIsAuthor(false);
118         user.setIsAdmin(false);
119         user.setStatus(1);
120
121         when(request.getHeader("Authorization")).thenReturn("Bearer " + token);
122         when(jwtUtil.validateToken(token)).thenReturn(true);
123         when(jwtUtil.extractEmail(token)).thenReturn(email);
124         when(userMapper.selectByEmail(email)).thenReturn(user);
125         when(jwtUtil.validateToken(token, user)).thenReturn(true);
126
127         jwtFilter.doFilterInternal(request, response, filterChain);
128
129         verify(filterChain).doFilter(request, response);
```

Impacted Analysis Objects

- Acceptance criteria in user stories
- Business rules (XP calculation, yuan gain)

Impacted Use Cases / User Stories

- Registration success/failure
- Reading accumulates XP and triggers level-ups

Typical Static Changes + Rationale

- Every Service testable in isolation: interfaces + pure strategies
- Integration tests using Testcontainers for DB
- Rationale: structure promotes testability (low coupling, mockable).

Typical Dynamic Changes + Rationale

- Test sequences: Given (initial data) → When (invoke Service) → Then (assert state)
- Optionally show Test Actor in sequence diagrams
- Rationale: unify mapping from use-case steps to test steps.

Checklist

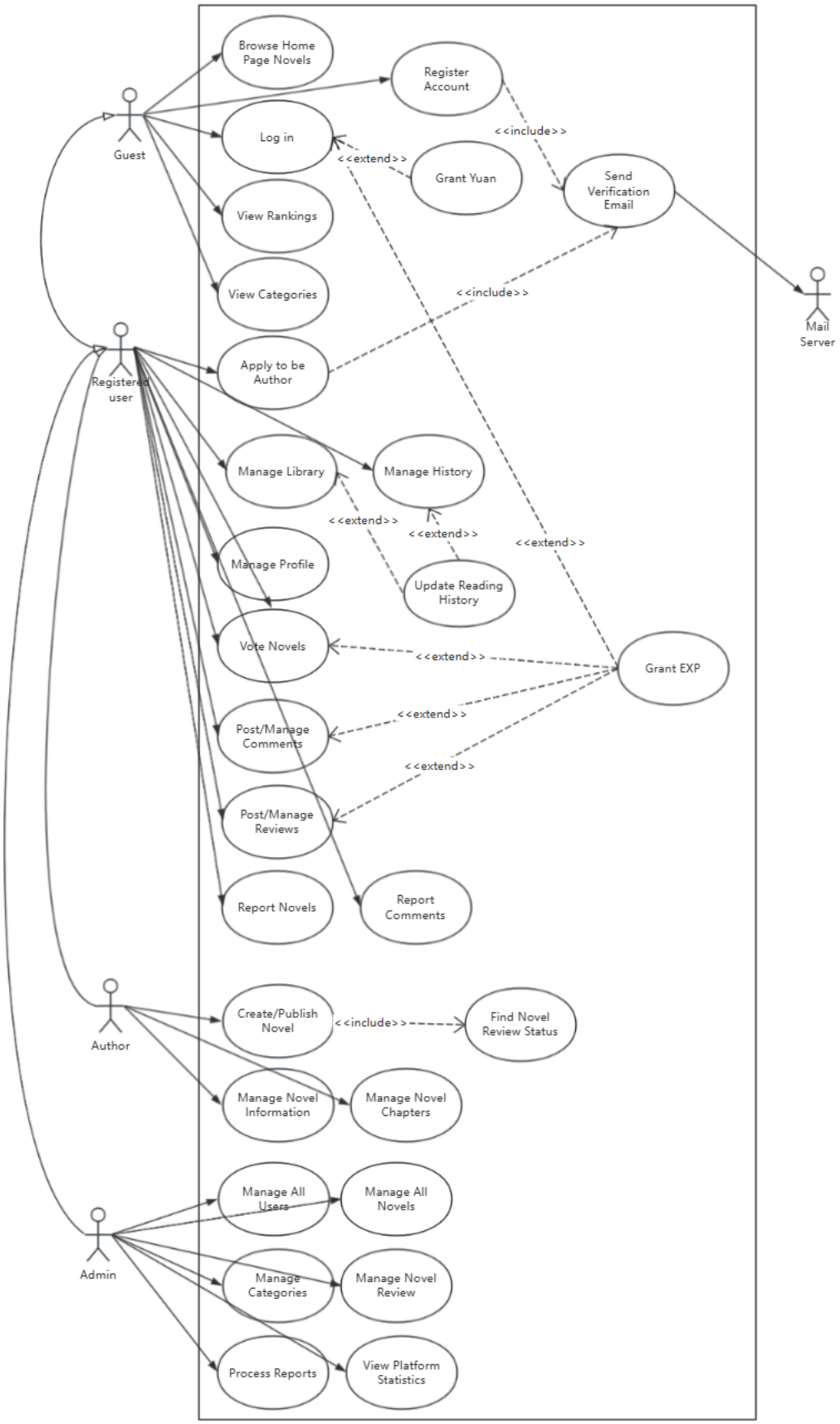
- Complex paths covered by unit + integration tests? Edge/error cases tested (XP overflow)?

Common Pitfalls

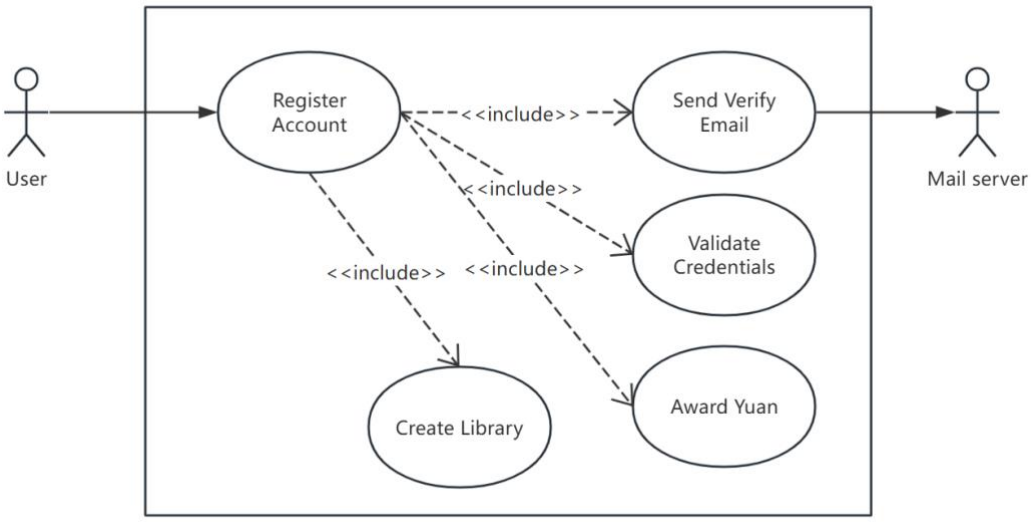
- Only testing Controllers; ignoring domain strategies
- Tests bound to real external services (e.g., email) → flakiness

3.3 Use Case Model

3.3.1 Use Case Diagram (Overall)



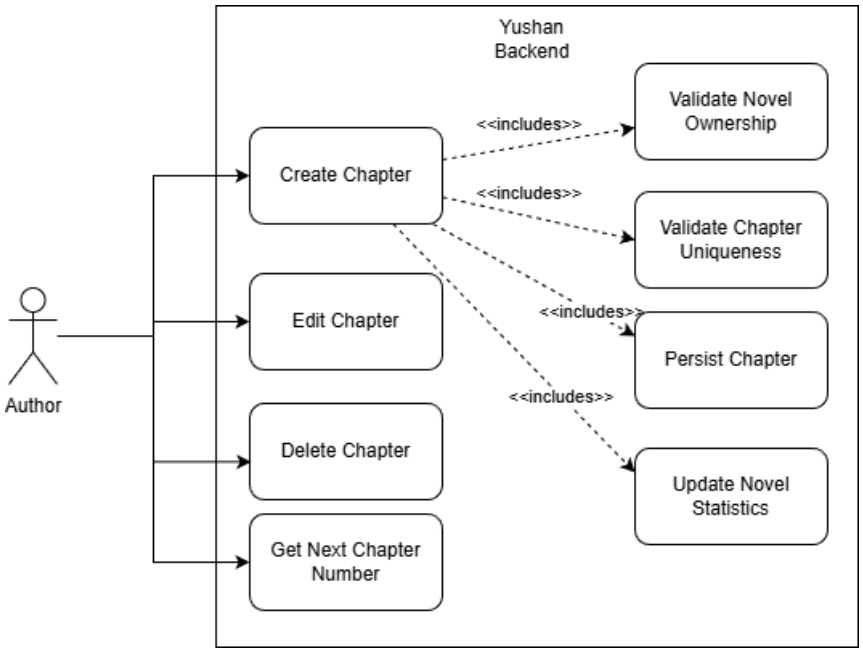
3.3.2 Use Case: User Register with Email Verification (Zhang Yan)



Use Case - User Register with Email Verification	
Use Case ID	**UC-01**
Description	A public user register with email verification
Primary Actors	Public User
Secondary Actors	Mail Server
Main Flow of events	1. The user fills out the registration form with email and clicks "send OTP" button. 2. System validates the email (format and if null). 3. System validates if the email has already been registered. 4. System generates the code and store it into the Redis. 5. System sends OTP to the email. 6. The user fills out the registration form with OTP and other information (username, gender, birthday) and clicks "Submit" button. 7. System validates the information's format. 8. System validates if the email has already been registered. 9. System validates if the code is right. 10. System inserts the user into user table. 11. System creates the library for the user. 12. System awards yuan for the user. 13. System returns user information dto.
Alternative Flow of events	• If email's format is not right and email has been registered -> return validation exception "Email is required" or "Email must be valid" or "Email already existed". • If user requests to send email too frequently -> return validation exception "email sends too often, please try again after xx second(s)".

	• If user provides wrong code or the code expired after 5 minutes-> return validation exception "Invalid verification code or code expired".
Preconditions	• The user is new and does not have an existing account with the provided email.
Postconditions	• On success: insert the user account into the user table, insert the library into the library table. • On failure: no DB side effects.

3.3.3 Use Case: Author Create Chapter (Yang Shuang)

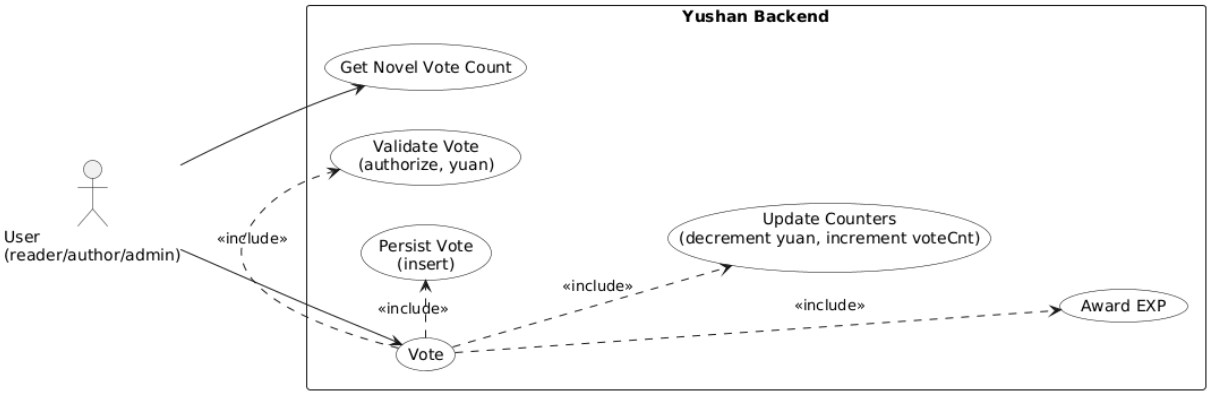


Use Case - Author Create Chapter	
Use Case ID	**UC-02**
Description	A author creates a chapter for a novel
Primary Actors	Author
Secondary Actors	/
Main Flow of events	1. Load novel: NovelMapper.selectByPrimaryKey(req.novelId); if null or ARCHIVED → throw ResourceNotFoundException("novel not found").

	2. Author check: if novel.authorId != userId → throw IllegalArgumentException("only the author can create chapters"). 3. Duplicate check: ChapterMapper.existsByNovelIdAndChapterNumber(req.novelId, req.chapterNumber); if true → throw IllegalArgumentException("chapter number already exists for this novel"). 4. Compute word count if needed: if req.wordCnt is null and content not blank → wordCnt = trimmed content length. 5. Build Chapter entity with fields from req and defaults (uuid=UUID.randomUUID(), isPremium default false, yuanCost default 0.0f, viewCnt=0L, isValid default true, createTime/updateTime=now, publishTime=req.publishTime or now). 6. Persist: ChapterMapper.insertSelective(chapter). 7. Recalculate novel statistics: chapterCnt ← ChapterMapper.countPublishedByNovelId(req.novelId). wordCnt ← ChapterMapper.sumWordCountByNovelId(req.novelId). 8. Set fields on Novel and call NovelMapper.updateByPrimaryKeySelective(novel). 9. Fetch the created chapter by UUID and map to ChapterDetailResponseDTO. 10. Return ChapterDetailResponseDTO to the controller, which returns HTTP 201/success.
Alternative Flow of events	• Novel not found or archived → ResourceNotFoundException("novel not found") → abort before any DB writes. • User is not the novel's author → IllegalArgumentException("only the author can create chapters") → abort; no DB writes. • Chapter number already exists for this novel → IllegalArgumentException("chapter number already exists for this novel") → abort; no DB writes. • Any persistence error after insert (e.g., during statistics update) → transaction rollback (insert is undone).
Preconditions	• User is authenticated (UUID userId available to ChapterService.createChapter).

	• Novel with req.novelId exists and is NOT ARCHIVED (validated with NovelMapper.selectByPrimaryKey and NovelService.mapStatus(ARCHIVED)). • The authenticated user is the author of that novel.
Postconditions	**On success (transaction commits):** • One Chapter row is inserted via ChapterMapper.insertSelective. • Chapter defaults are set as in code: isPremium=false if null, yuanCost=0.0f if null, viewCnt=0L, isValid=true if null, publishTime=req.publishTime or now; wordCnt computed from trimmed content length when req.wordCnt is null and content is provided. • Novel statistics are recomputed in one transaction: chapterCnt = ChapterMapper.countPublishedByNovelId(novelId) [counts valid chapters with publish_time <= NOW]. wordCnt = ChapterMapper.sumWordCountByNovelId(novelId) [sums word_cnt of valid chapters]. • NovelMapper.updateByPrimaryKeySelective persists the new statistics. • A ChapterDetailResponseDTO for the created chapter is returned (built by fetching the chapter by UUID). **On failure:** • Transaction rolls back; no partial inserts or statistic updates remain.

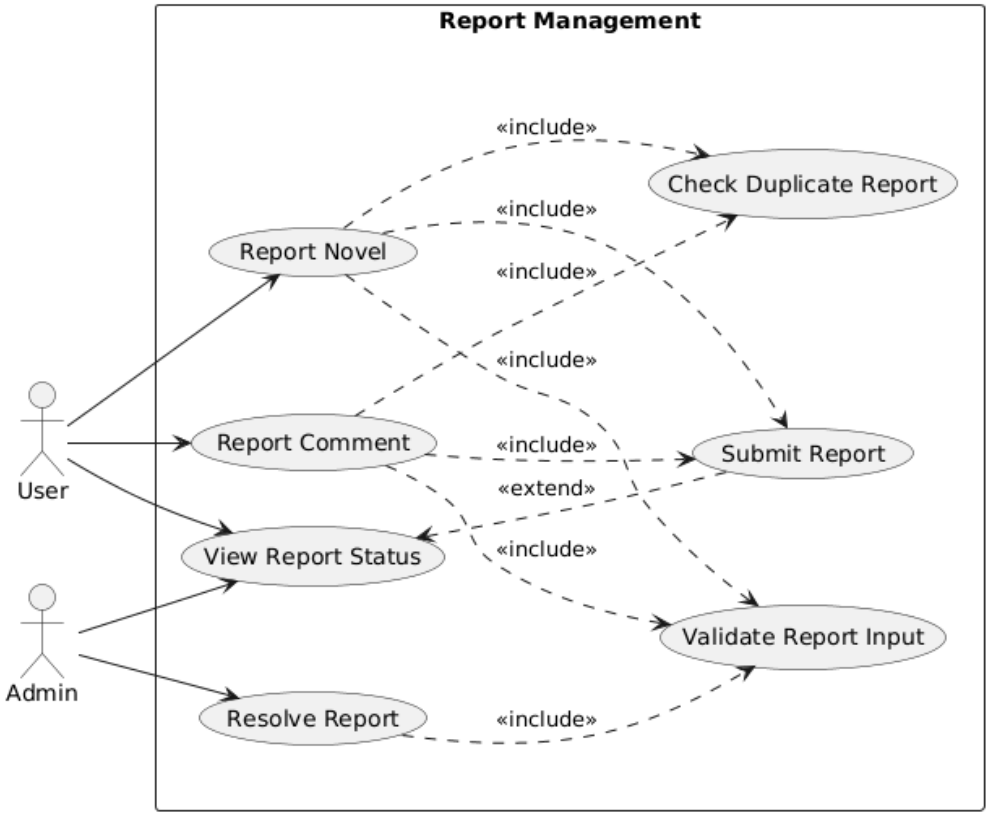
3.3.4 Use Case: User Vote for Novel (Zhu Yuhui)



Use Case - User Vote for Novel	
Use Case ID	**UC-03**
Description	An authenticated user (non-author of the target novel) spends 1 yuan to cast a vote that increases the novel's vote count and awards EXP.
Primary Actors	Authenticated User (Reader or other role not being the novel's author)

Secondary Actors	/
Main Flow of events	1. User initiates a vote action on a novel (UI “Vote” button). 2. System authenticates the JWT and loads user & novel records. 3. System validates business rules: novel exists & accessible; user is not the author; user has ≥ 1 yuan. 4. Within a single database transaction: a. Insert vote record (fail if unique constraint per user+novel). b. Deduct 1 yuan from user (ensure non-negative constraint). c. Increment novel.voteCnt atomically. d. Award EXP to user (EXP delta defined). 5. Commit transaction (if any step fails → rollback). 6. Return VoteResponseDTO (novelId, voteCount, remainedYuan [, expAwarded?]).
Alternative Flow of events	● If user is the author of this novel → validation fails → return error "Cannot vote your own novel". ● If user has not enough yuan → validation fails → return error "Not enough yuan". ● If inserting the vote fails due to duplicate key (already voted) → handle as "already voted" ● Invalid/Deleted novel → return 404/“Novel not found”. ● Unauthorized / expired JWT → 401. ● DB deadlock / transient failure → return 500 with generic message.
Preconditions	● The user is logged in with a valid JWT token. ● The user has at least 1 yuan. ● The target novel exists and is accessible.
Postconditions	● On success: All changes committed atomically. Vote record exists, user.yuan decreased, novel.voteCnt updated, user.exp increased. ● On failure: No persistent side effects; any partial writes rolled back; client may re-attempt if error is transient.

3.3.5 Use Case: User Report Novel & Comment (Nguyen Phu Truong)



Use Case - Report Novel	
Use Case ID	**UC-04**
Description	A reader reports a novel for inappropriate content; the system records the report, prevents duplicates, and informs moderation staff.
Primary Actors	Authenticated User
Secondary Actors	Admin
Main Flow of events	1. The user is browsing the novel detail page and clicks the “Report” button. 2. Frontend opens the report dialog; user selects reportType, types the reason, and submits. 3. Frontend sends POST /api/reports/novel/{novelId} with ReportCreateRequestDTO. 4. Controller validates the payload and resolves reporterId from authentication. 5. Service verifies the novel exists via NovelService. 6. Service validates reportType and checks duplicates through ReportMapper.existsReportByUserAndContent. 7. Service creates a Report entity, sets status IN_REVIEW, saves it (insertSelective), enriches the DTO (reporter username), and returns ReportResponseDTO. 8. Controller wraps the DTO in ApiResponse and the frontend shows a success toast.

Alternative Flow of events	1a. The user closes the dialog without submitting → no request sent, flow ends. 5a. Novel not found → throw ResourceNotFoundException, return 404. 6a. Primary validation fails (reportType invalid or reason blank) → ValidationException, return 400. 6b. Duplicate report detected → ValidationException, return 400. 7a. Persistence failure → transaction rollback, return 500.
Preconditions	1. The user is logged in with a valid JWT token. 2. The target novel exists and is accessible. 3. Request passes validation: reportType is valid, reason is not blank.
Postconditions	1. A new report record with contentType=NOVEL is persisted. 2. Report status is set to IN_REVIEW; timestamps are recorded. 3. The user receives a success response and can view the report in their history.

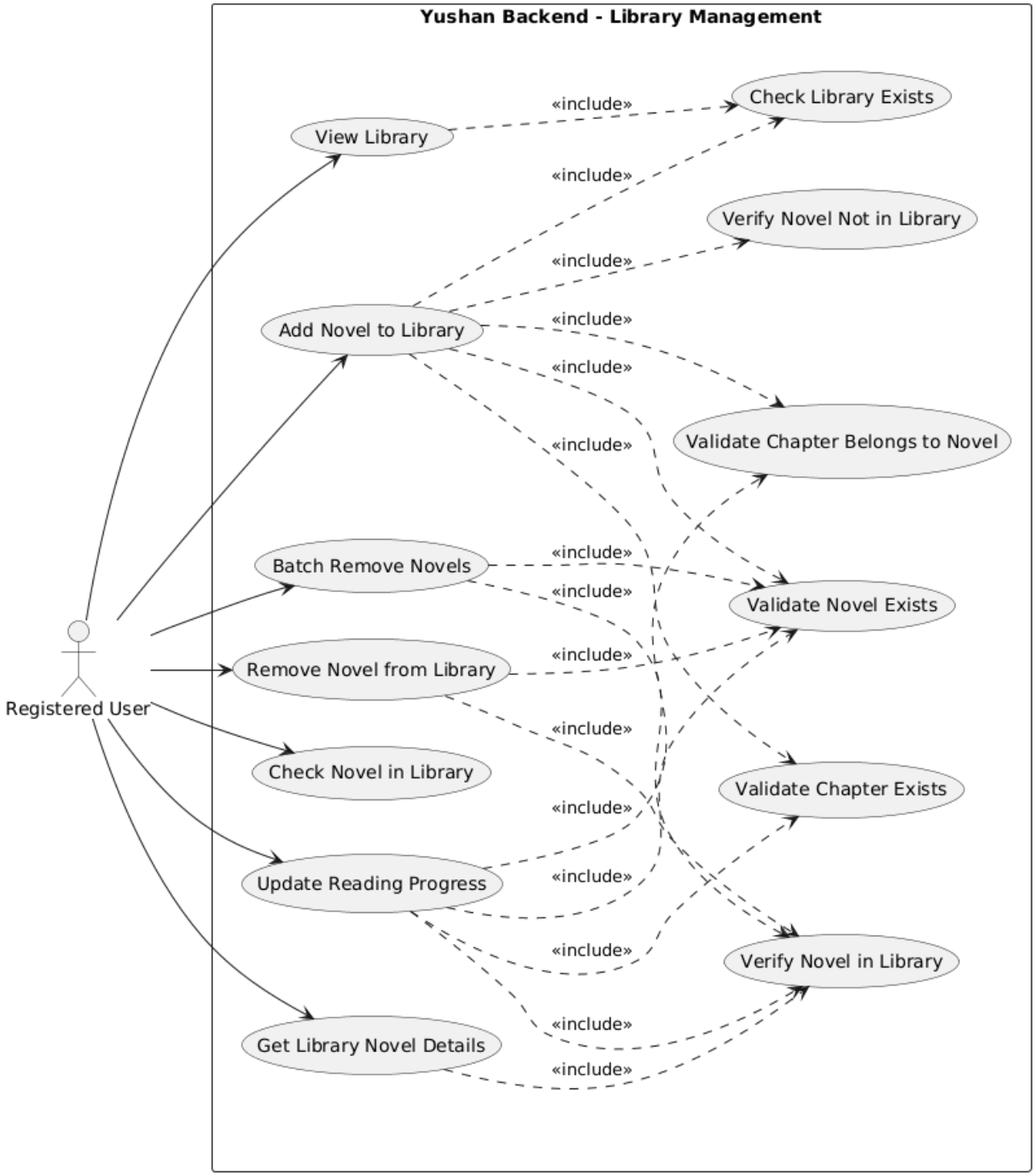
Use Case - Report Comment	
Use Case ID	**UC-05**
Description	On the comment section, the reader presses “Report comment”, provides the reason in the modal form, and submits; the backend validates the comment and stores the report for moderation review.
Primary Actors	Authenticated User
Secondary Actors	Admin
Main Flow of events	1. The user hovers over a specific comment and clicks “Report comment”. 2. Frontend collects reportType and reason, then submits. 3. Frontend sends POST /api/reports/comment/{commentId} to backend. 4. Controller extracts reporterId. 5. Service uses CommentService to load the comment entity and ensure it exists. 6. Service validates reportType and checks for duplicates. 7. Service builds and saves a Report with contentType="COMMENT", enriches the DTO and returns it. 8. Controller responds with a success ApiResponse; frontend notifies user that the report was received.
Alternative Flow of events	1a. User cancels the modal → no backend interaction, flow ends. 5a. Comment missing during validation → ResourceNotFoundException, return 404. 6a. Invalid reportType or reason blank → ValidationException,

	return 400. 6b. Duplicate report → ValidationException, return 400. 7a. Persistence failure → rollback and surface error, return 500.
Preconditions	1. The user is authenticated. 2. Target comment exists. 3. Request passes validation (reportType, reason).
Postconditions	1. Report record with contentType=COMMENT is saved and set to IN_REVIEW. 2. Admin dashboard and user report list reflect the new entry.

Use Case - Resolve Report	
Use Case ID	**UC-06**
Description	Through the admin dashboard, a moderator opens the report management screen, reviews the details, chooses resolve/dismiss with mandatory notes, and confirms; the system updates report status, applies the required moderation action, and notifies stakeholders.
Primary Actors	Admin
Secondary Actors	Reporter
Main Flow of events	1. Admin logs into the moderation dashboard and opens the "Reports" management screen. 2. Admin reviews report details, selects an action (RESOLVED or DISMISSED), enters required moderation notes, and submits. 3. UI sends PUT /api/reports/admin/{id}/resolve with ReportResolutionRequestDTO. 4. The controller authorizes the request and forwards it to the service. 5. Service loads the report; if missing, throws an error. 6. Service validates that action is either RESOLVED or DISMISSED. 7. Service ensures adminNotes are present; rejects if blank. 8. Service updates status via ReportMapper.updateReportStatus, and if the action is RESOLVED, applies domain effects: a. For novel reports, call NovelService to set the novel status to DRAFT. b. For comment reports, call CommentService to remove the comment. 9. Service reloads the report, maps to ReportResponseDTO. 10. Controller returns success via ApiResponse; dashboard reflects the updated status and notes.
Alternative Flow of events	2a. Admin aborts without submitting → no change applied. 5a. Report not found → ResourceNotFoundException, return 404. 6a. Action not in {RESOLVED, DISMISSED} → ValidationException, return 400.

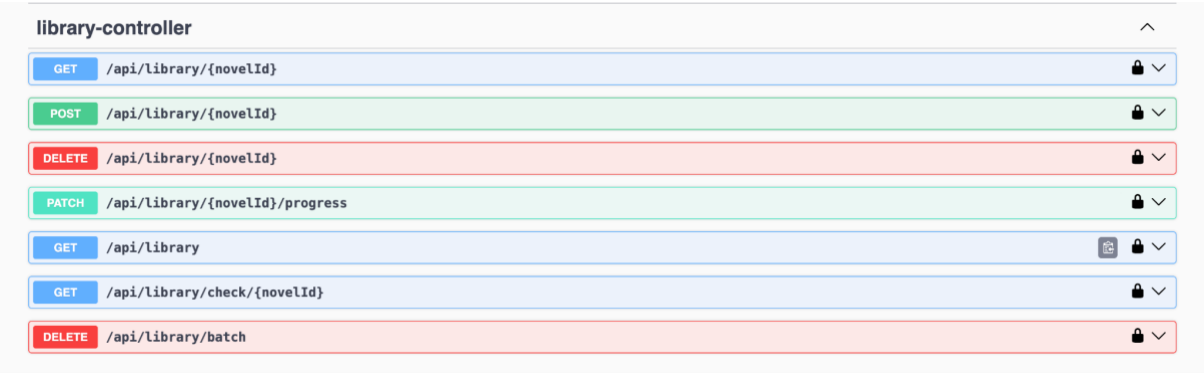
	7a. Notes missing → ValidationException, return 400. 8a. Database failure during update → rollback operation, return 500.
Preconditions	1. The admin has logged into the admin page 2. Report exists and is currently IN_REVIEW. 3. Request includes action ∈ {RESOLVED, DISMISSED}.
Postconditions	1. Report status updates accordingly; adminNotes, resolvedBy, and timestamps are stored. 2. Updated data appears in the admin dashboard and reporter history.

3.3.6 Use Case: User Add Novel to Library (Ahan Jaiswal)



Use Case - Add Novel to Library for a Registered User	
Use Case ID	**UC-07**
Description	A registered user adds a novel to their library with reading progress.
Primary Actors	Registered User
Secondary Actors	None
Main Flow of events	1. The user selects a novel and clicks the "Add to Library" button with optional progress information (chapter ID). 2. System extracts user ID from authentication token. 3. The system validates that the novel exists in the database. 4. The System validates if progress (chapter ID) is provided: 5. Validates that the chapter exists. 6. Validates that the chapter belongs to the specified novel. 7. The system retrieves the user's library by user ID. 8. The system checks if the novel already exists in the user's library. 9. System creates a new NovelLibrary entry with novel ID, library ID, and optional progress. 10. The system inserts the NovelLibrary record into the database. 11. The system returns a success response: "Added this novel to the library successfully".
Alternative Flow of events	1. If the user is not authenticated → return UnauthorizedException "Authentication required". 2. If user ID is not found in token → return ValidationException "User ID not found". 3. If the novel does not exist → return ResourceNotFoundException "novel not found: {novelId}". 4. If progress chapter does not exist → return ResourceNotFoundException "Chapter not found with id: {progress}". 5. If chapter does not belong to the novel → return ValidationException "Chapter don't belong with novel id: {novelId}". 6. If the user's library does not exist → return ResourceNotFoundException "User with ID {userId} does not have a library". 7. If the novel already exists in the library → return ValidationException "novel exists in the library".
Preconditions	1. The user must be registered and authenticated. 2. The user must have a library created (created during registration). 3. The novel must exist in the system. 4. If progress is specified, the chapter must exist and belong to

	the novel.
Postconditions	1. On success: A new NovelLibrary record is inserted into the novel_library table with the novel ID, library ID, progress (optional), and timestamps (createTime, updateTime). 2. On failure: No database side effects.



Additional Use Case Details:

API Endpoint: POST /api/library/{novelId}

Request Body (Optional)

```
{
  "progress": 7
}
```

Note: progress is optional and represents the chapter ID for reading progress

Success Response (HTTP 200)


```
{
  "code": 200,
  "message": "Add novel to library successfully",
  "data": null,
  "timestamp": "2025-11-11 07:36:08"
}
```

Business Rules

1. **Library Initialization**: Every user must have a library created during registration. If a library doesn't exist, the operation fails with a 404 error.
2. **Uniqueness Constraint**: A user cannot add the same novel to their library twice. The system checks for duplicates before insertion and returns a 400 validation error if the novel already exists.
3. **Progress Validation**: When reading progress is provided:
    - The progress field represents a chapter ID

- The chapter must exist in the database (404 error if not found)
- The chapter must belong to the specified novel (400 error if mismatch)
- Progress is optional; if omitted or null, no reading progress is tracked

4. **Authentication & Authorization**:
   - All library operations require valid JWT authentication
   - The user ID (UUID) is extracted from the authentication token's principal
   - Unauthenticated requests return 401 error
5. **Response Format**: All responses follow a standard ApiResponse structure with:
   - code: HTTP status code (200, 400, 401, 404)
   - message: Human-readable success or error message
   - data: Response payload (null for this endpoint)
   - timestamp: Server timestamp in "yyyy-MM-dd HH:mm:ss" format
6. **Data Integrity**: The system maintains referential integrity:
   - User → Library (one-to-one relationship)
   - Library → NovelLibrary (one-to-many relationship)
   - Novel → NovelLibrary (one-to-many relationship)
   - Chapter → NovelLibrary (optional via progress field)

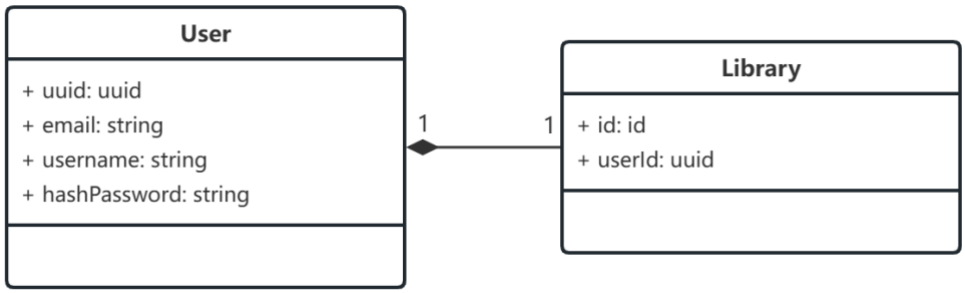
Related Use Cases

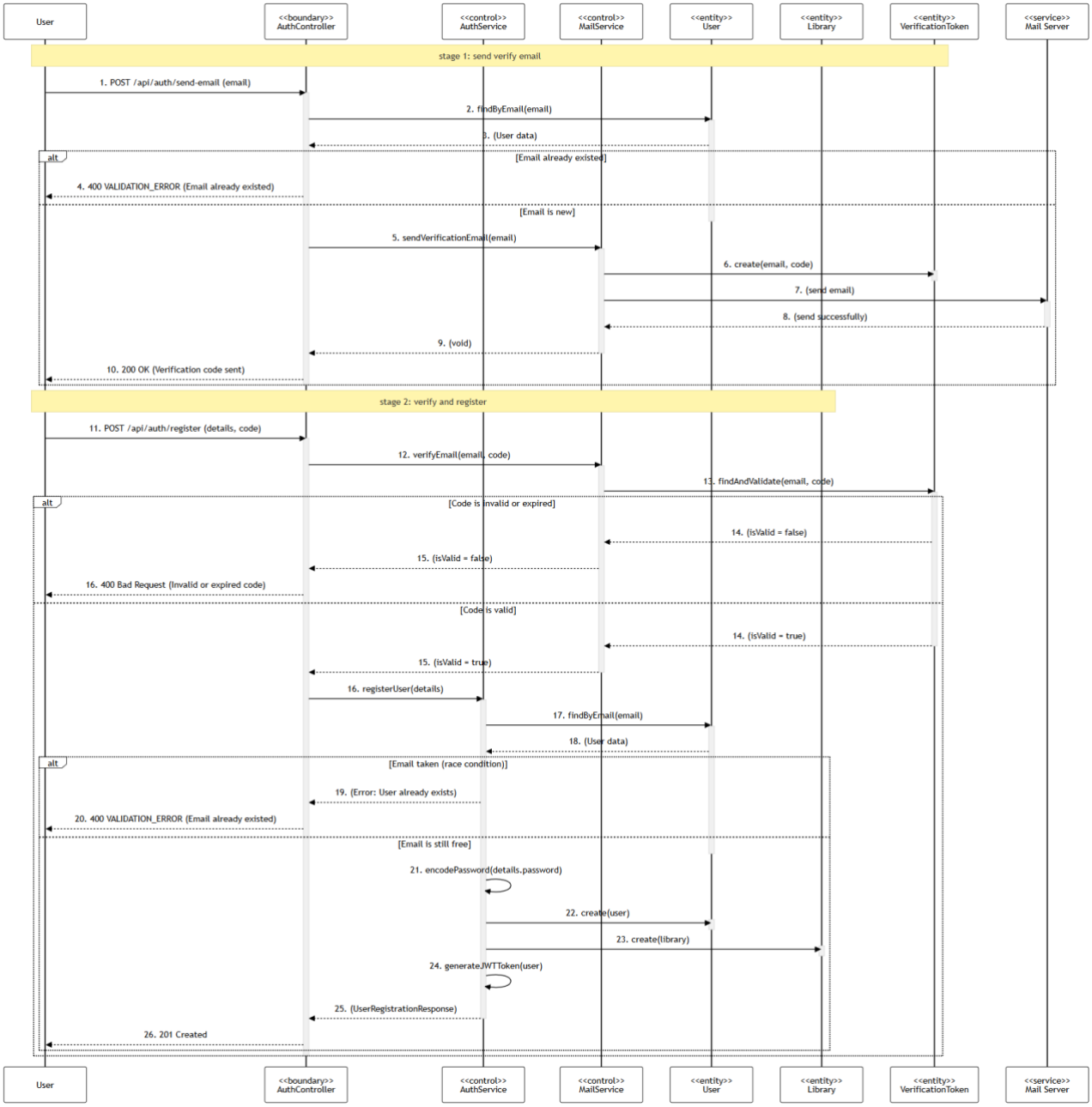
- **UC-08**: Remove Novel from Library
- **UC-09**: Batch Remove Novels from Library
- **UC-10**: View Library (with pagination)
- **UC-11**: Check Novel in Library
- **UC-12**: Get Library Novel Details
- **UC-13**: Update Reading Progress

3.4 Analysis & Design Models

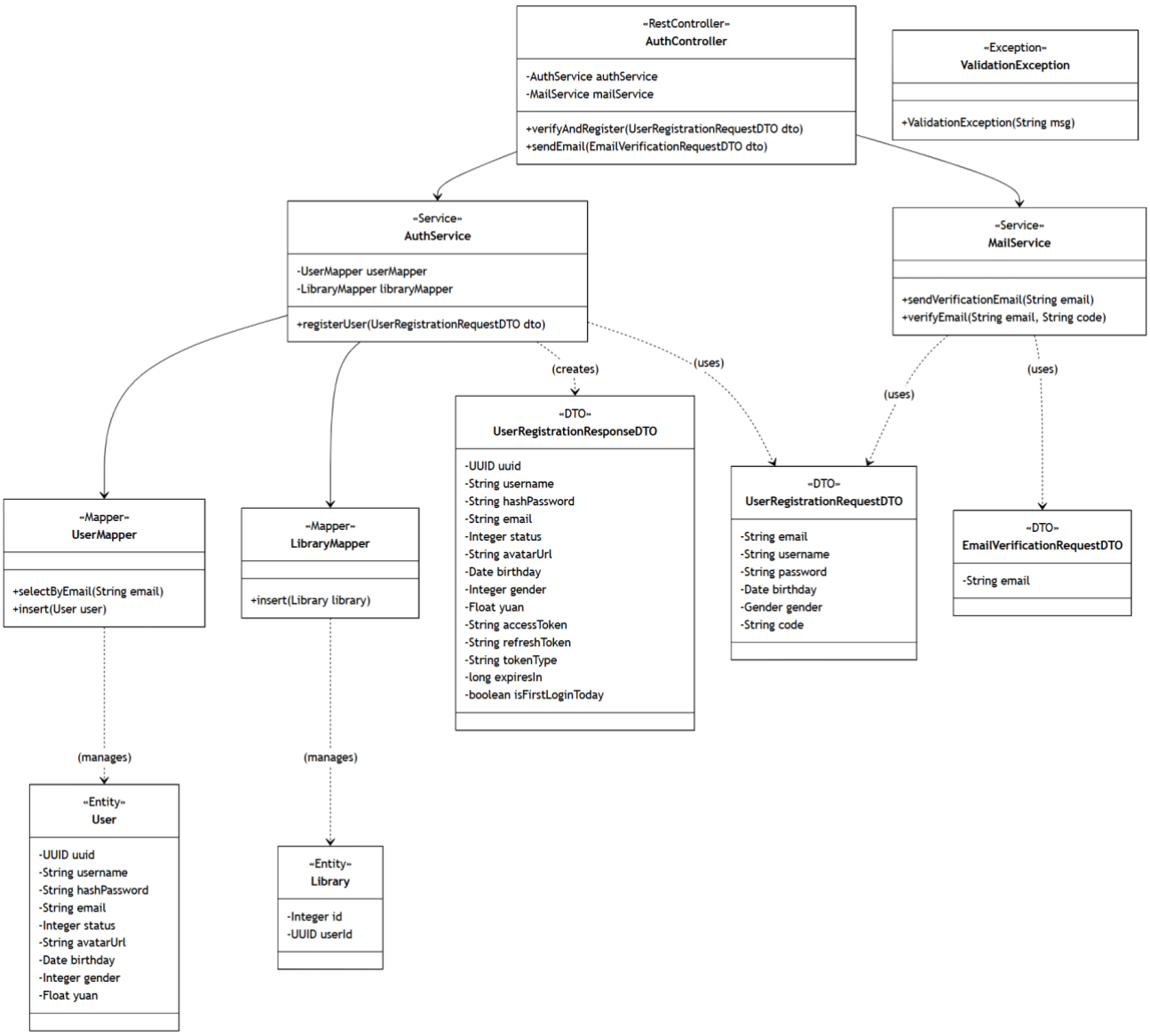
3.4.1 Use Case: User Register with Email Verification (Zhang Yan)

1. **Analysis Model**

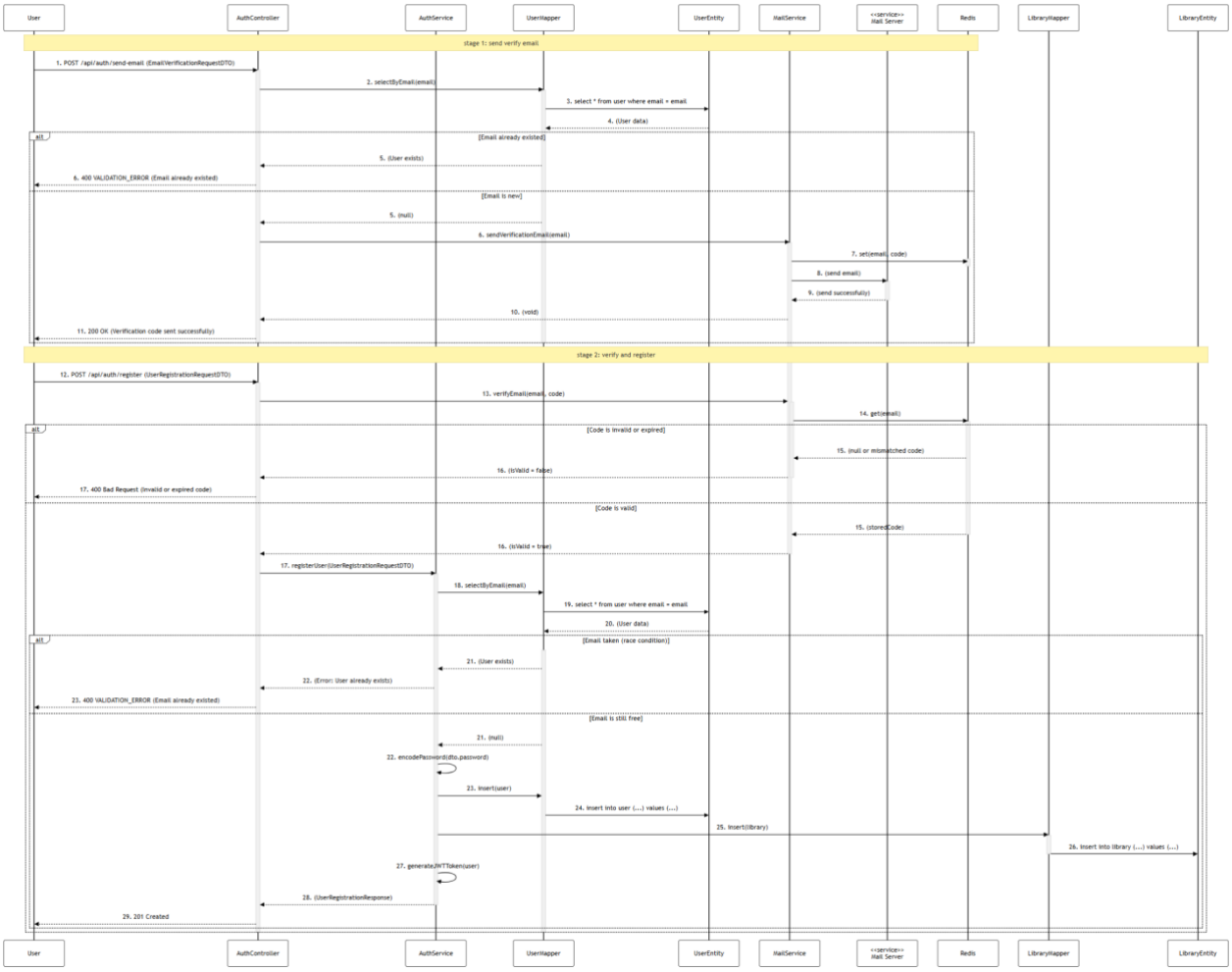



User
<<boundary>> AuthController
<<control>> AuthService
<<control>> MailService
<<entity>> User
<<entity>> Library
<<entity>> VerificationToken
<<service>> Mail Server
stage 1: send verify email
1. POST /api/auth/send-email (email)
2. findByEmail(email)
3. (User data)
alt
[Email already existed]
4. 400 VALIDATION_ERROR (Email already existed)
[Email is new]
5. sendVerificationEmail(email)
6. create(email, code)
7. (send email)
8. (send successfully)
9. (void)
10. 200 OK (Verification code sent)
stage 2: verify and register
11. POST /api/auth/register (details, code)
12. verifyEmail(email, code)
13. findAndValidate(email, code)
alt
[Code is invalid or expired]
14. (isValid = false)
15. (isValid = false)
16. 400 Bad Request (Invalid or expired code)
[Code is valid]
14. (isValid = true)
15. (isValid = true)
16. registerUser(details)
17. findByEmail(email)
18. (User data)
alt
[Email taken (race condition)]
19. (Error: User already exists)
20. 400 VALIDATION_ERROR (Email already existed)
[Email is still free]
21. encodePassword(details.password)
22. create(user)
23. create(library)
24. generateJWTToken(user)
25. (UserRegistrationResponse)
26. 201 Created
User
<<boundary>> AuthController
<<control>> AuthService
<<control>> MailService
<<entity>> User
<<entity>> Library
<<entity>> VerificationToken
<<service>> Mail Server

2. Design Model



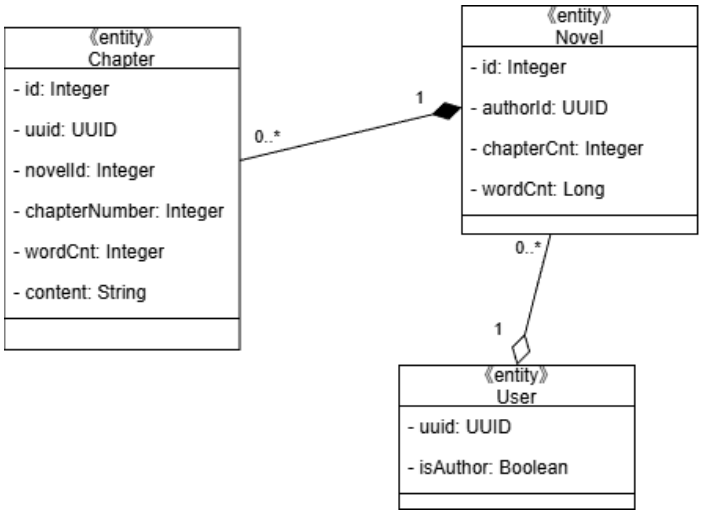
Class Diagram

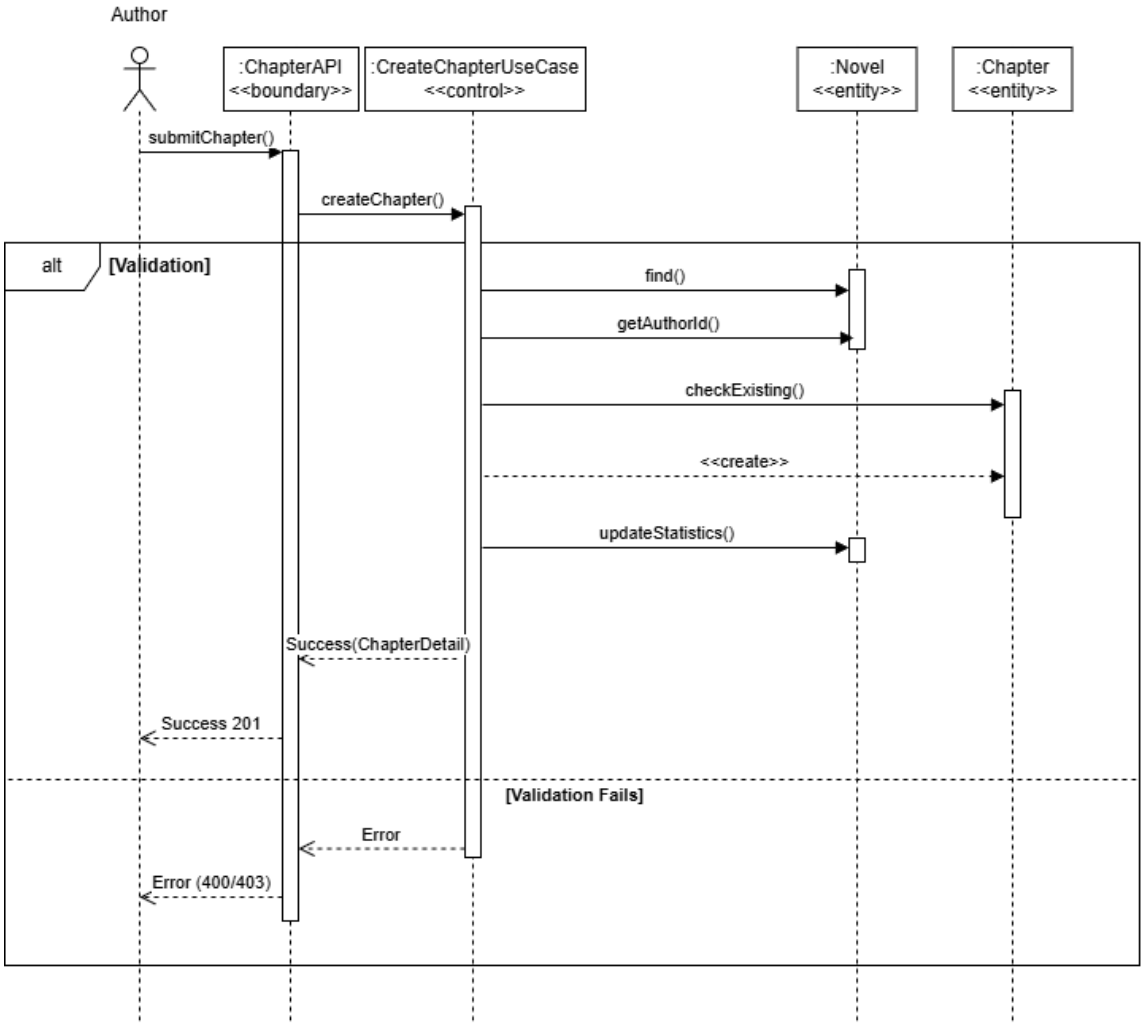


Sequence Diagram

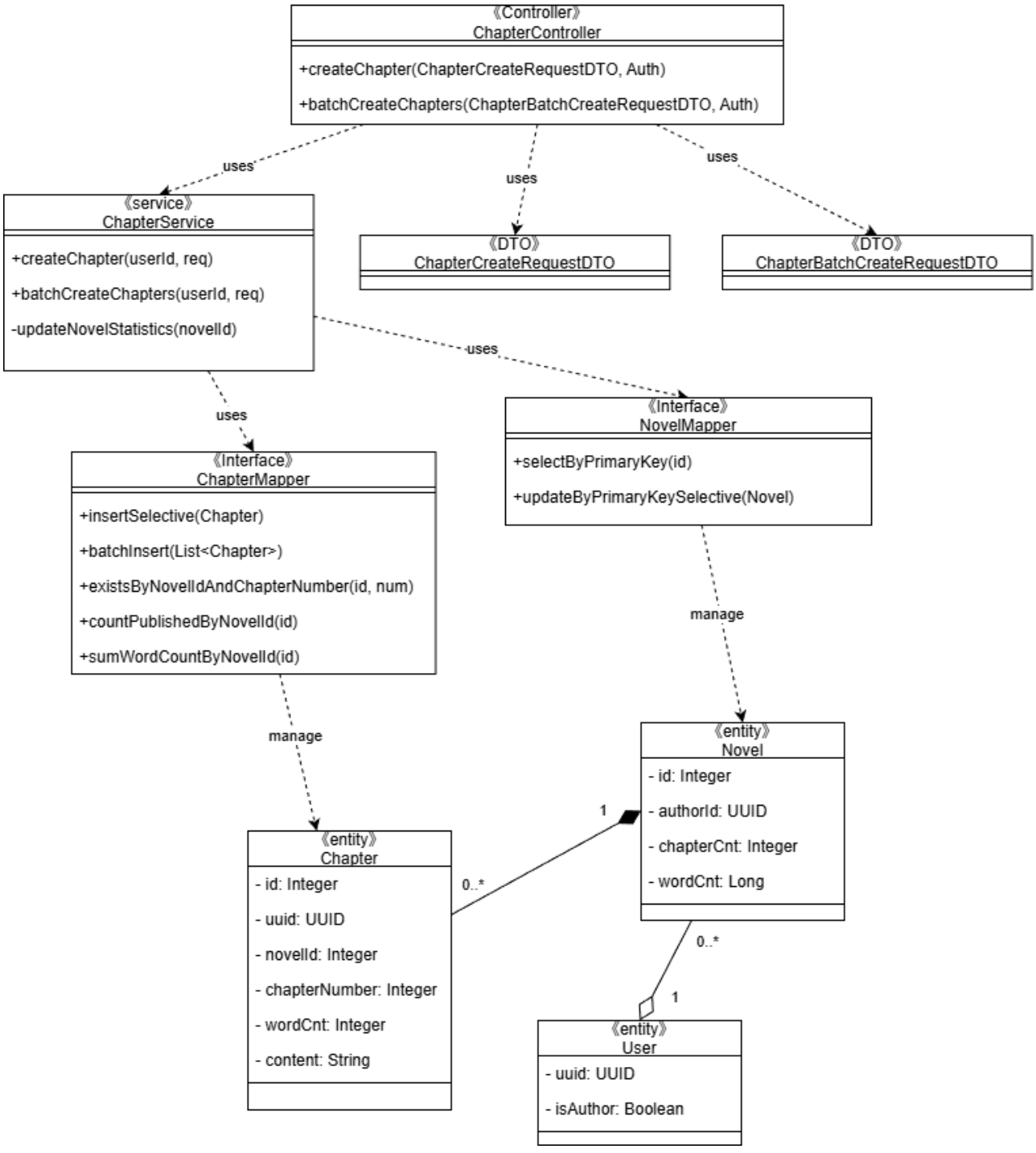
3.4.2 Use Case: Author Create Chapter (Yang Shuang)

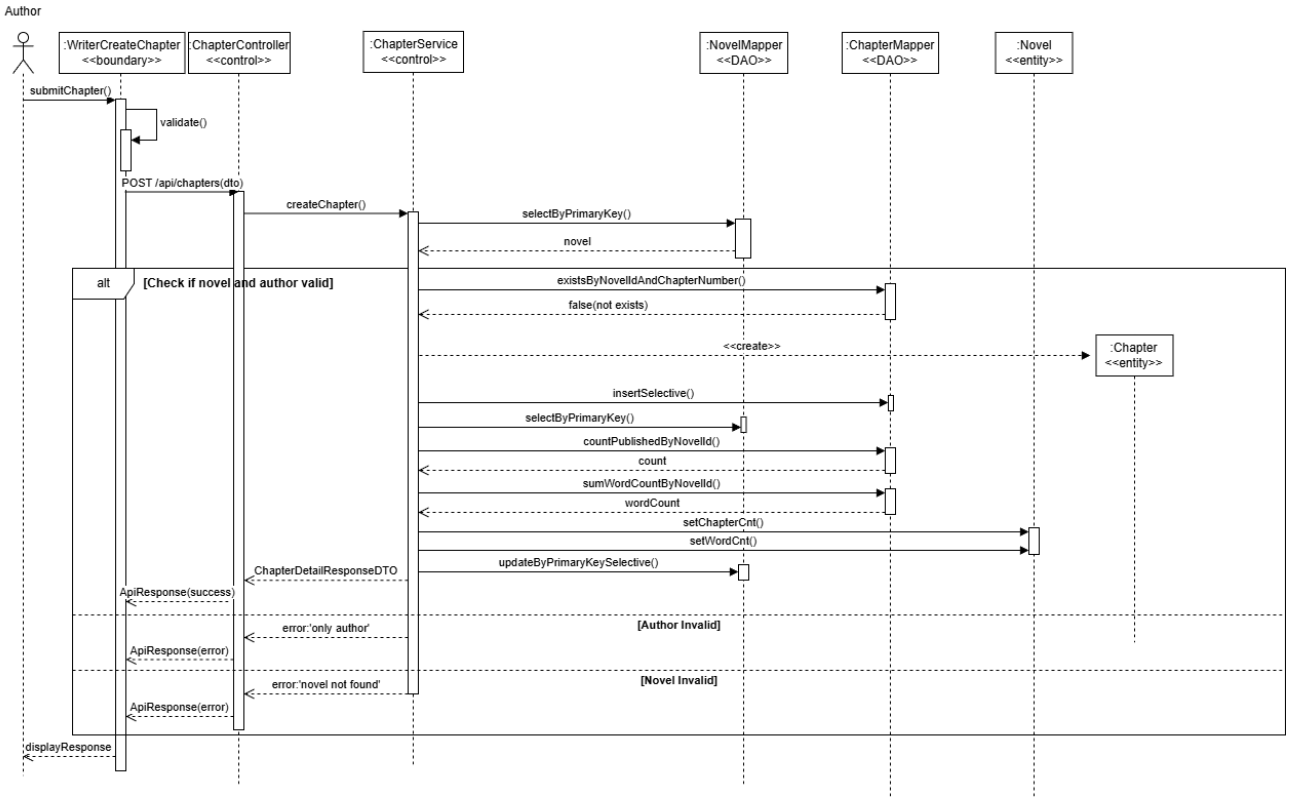
1. Analysis Model





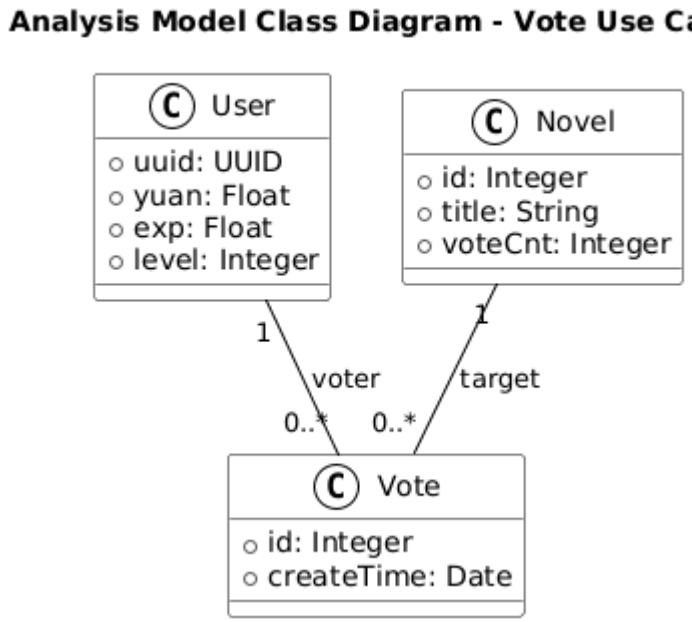
2. Design Model



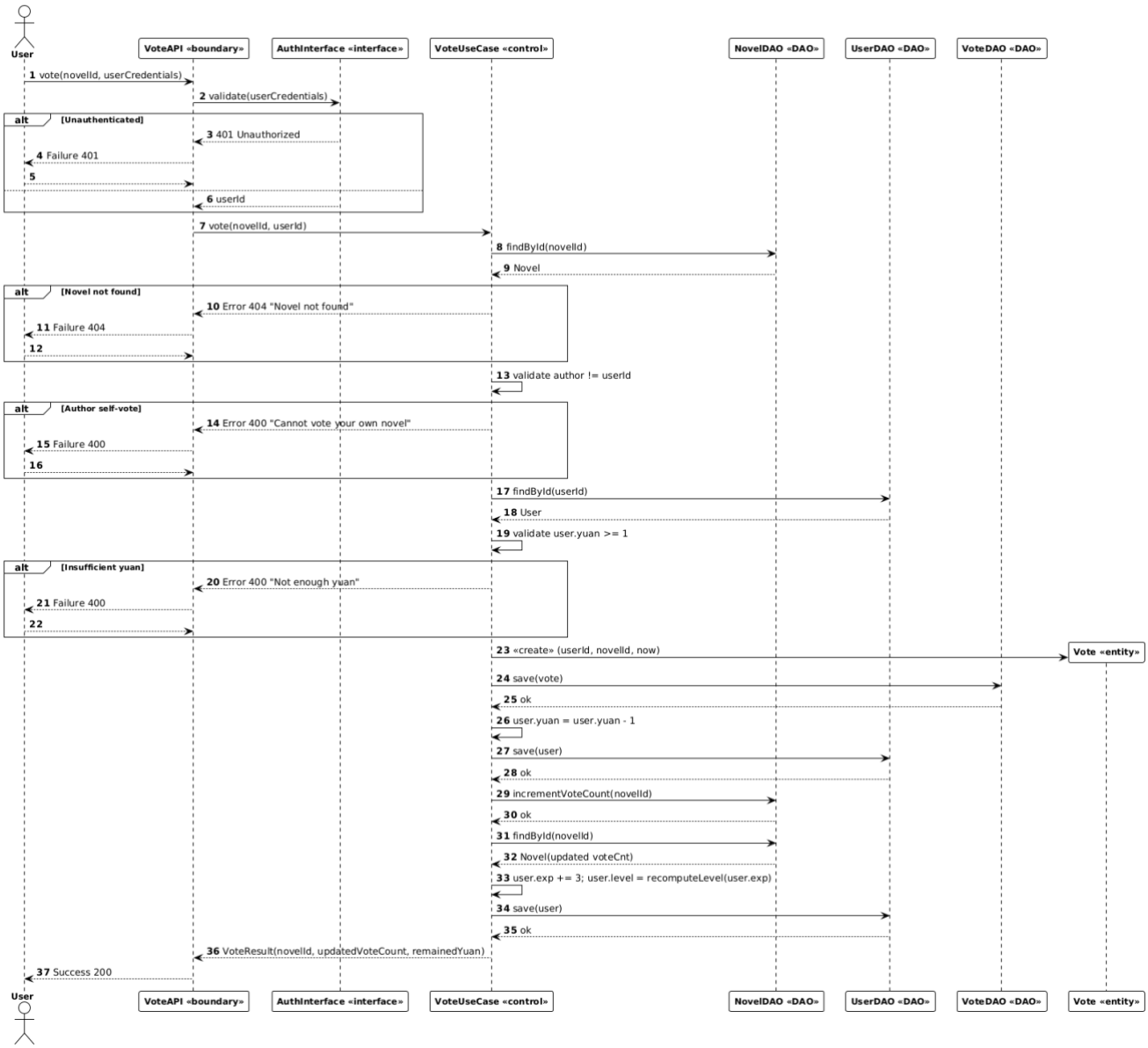


3.4.3 Use Case: User Vote for Novel (Zhu Yuhui)

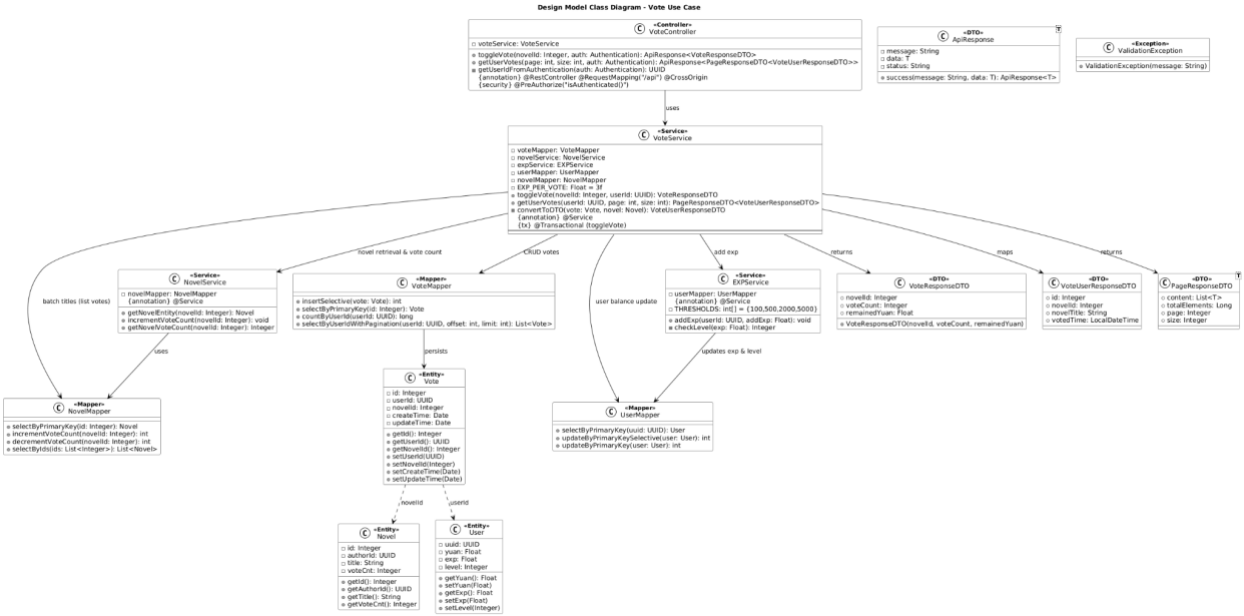
1. Analysis Model



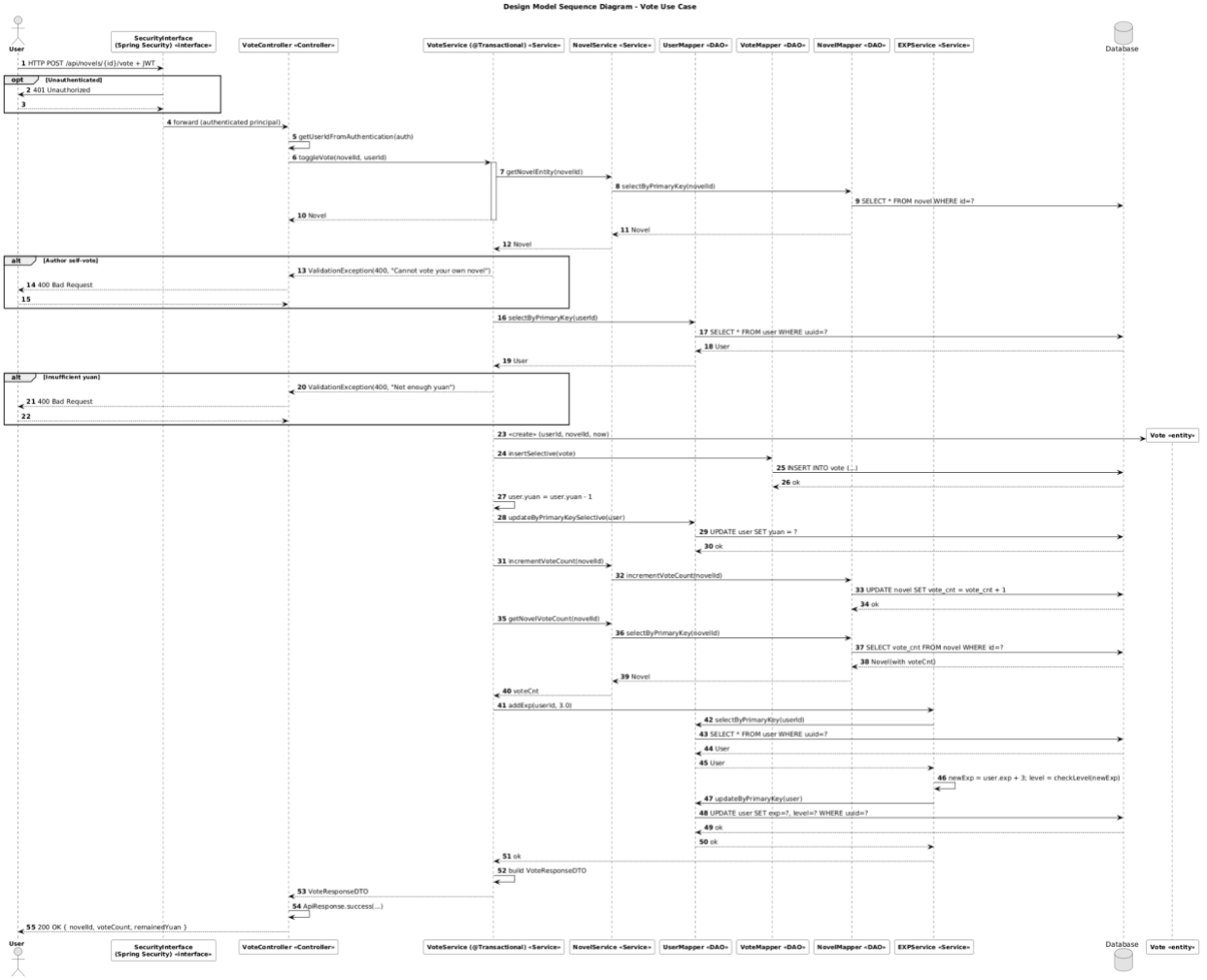
Analysis Model Sequence Diagram - Vote Use Case



2. Design Model

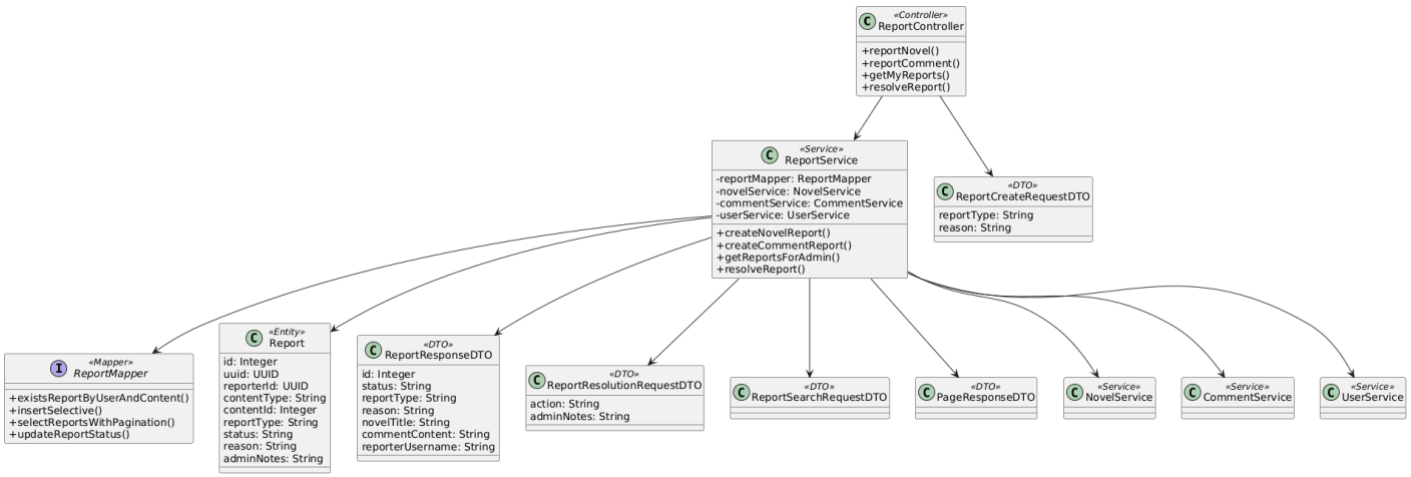


Class Diagram above / Sequence Diagram below

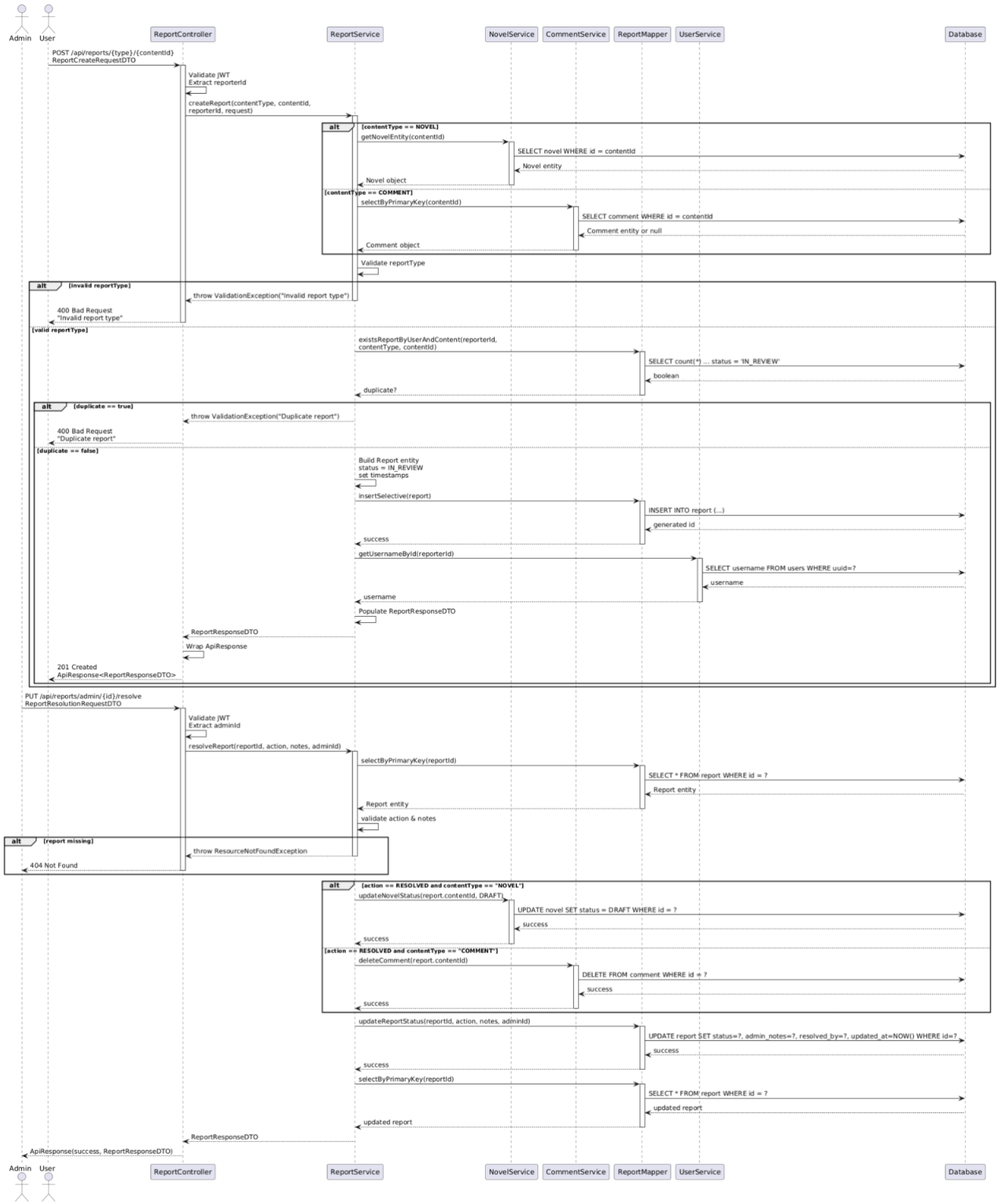


3.4.4 Use Case: User Report Novel & Comment (Nguyen Phu Truong)

1. Analysis Model

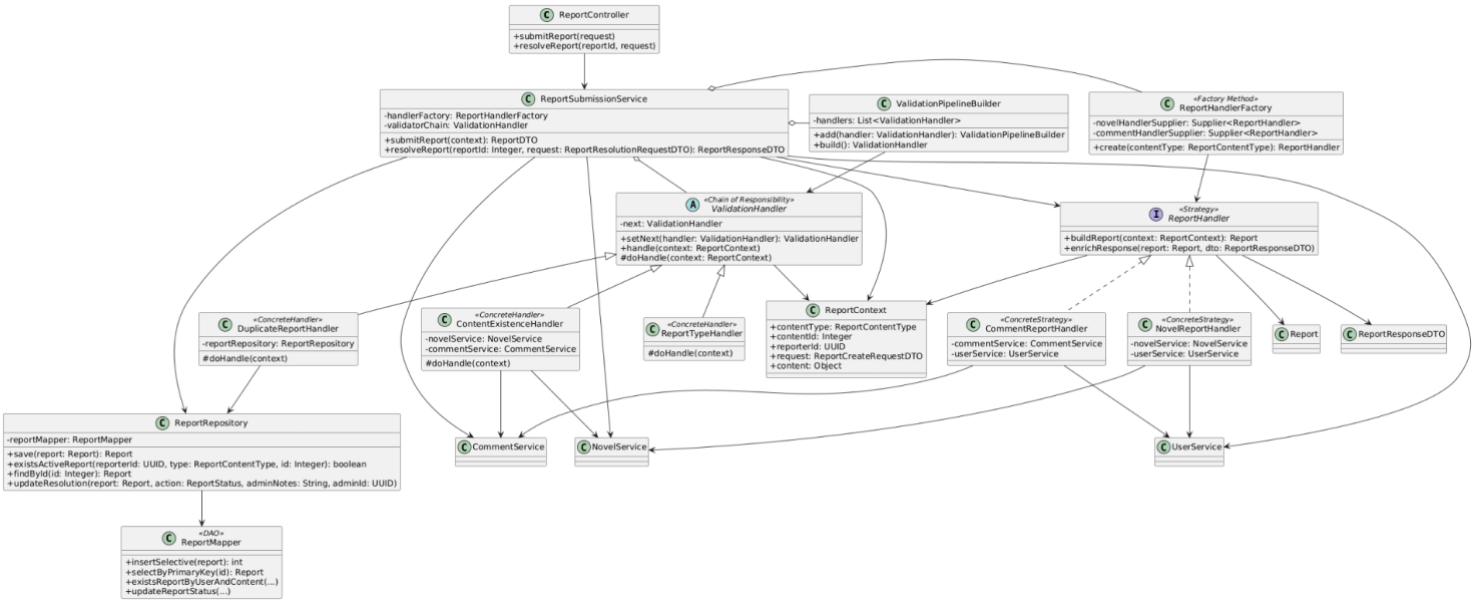


Class Diagram



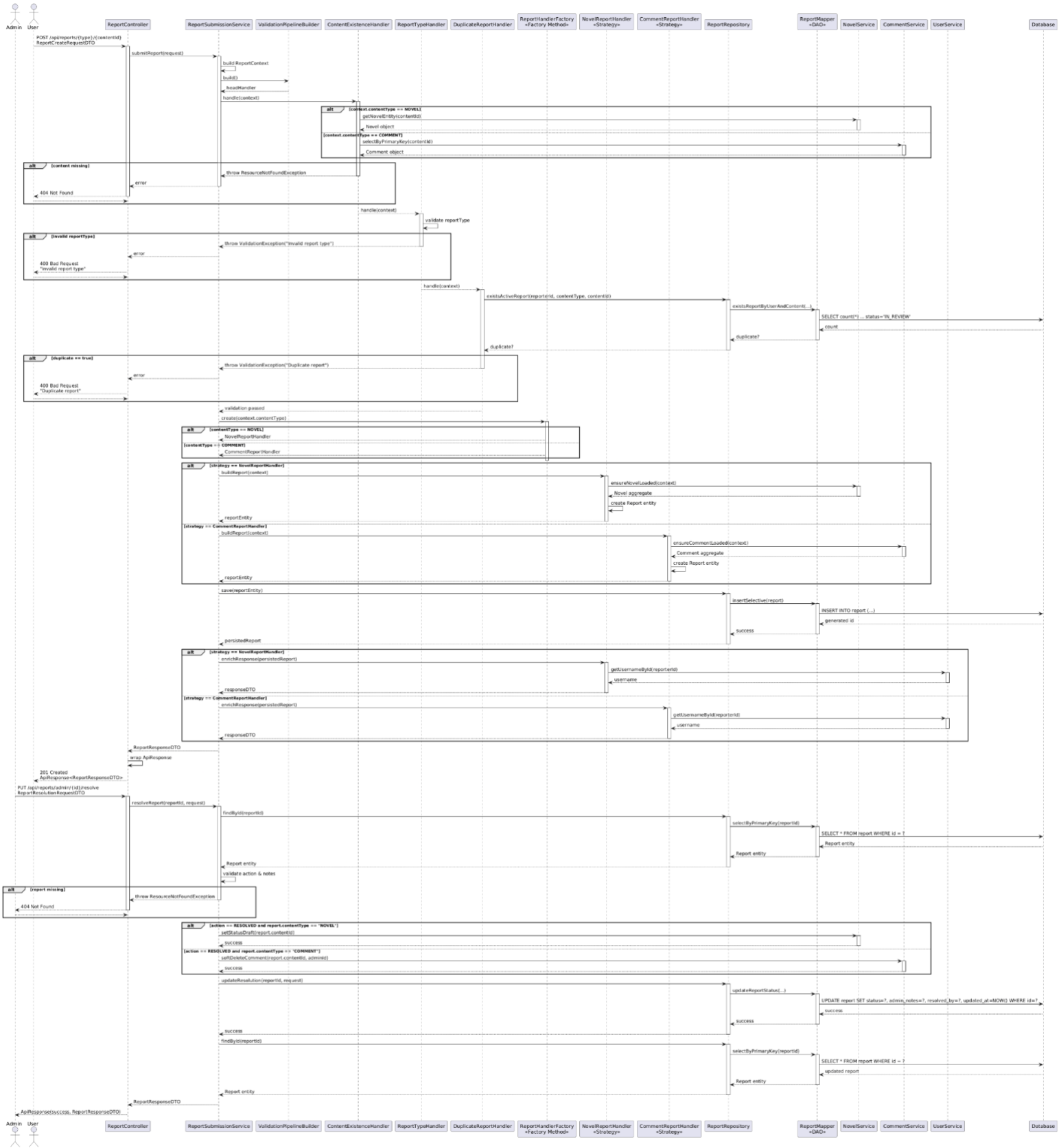
Sequence Diagram

2. Design Model



Class Diagram

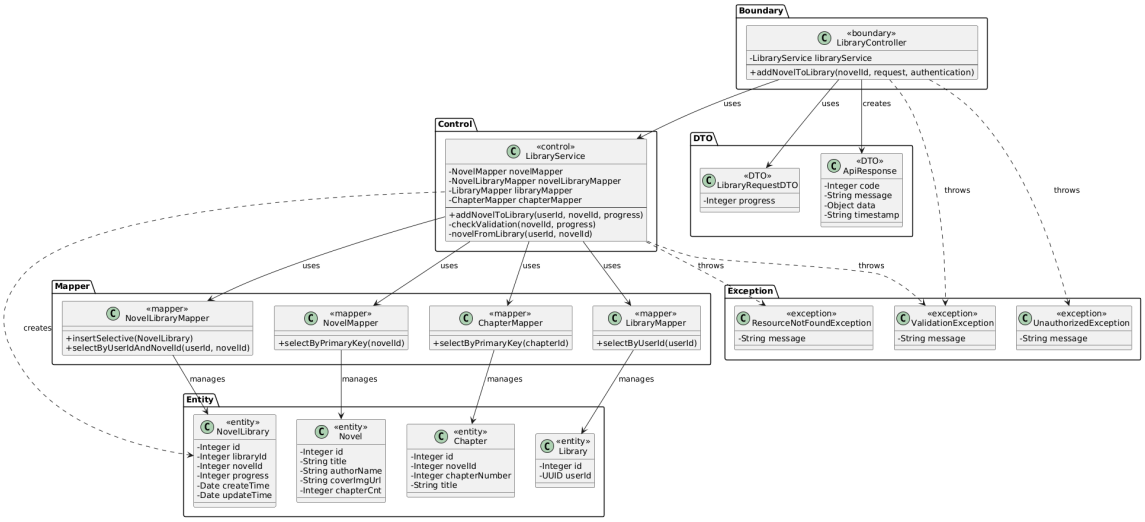
(after applying Chain of Responsibility for validation and Strategy & Factory Method for content handling)



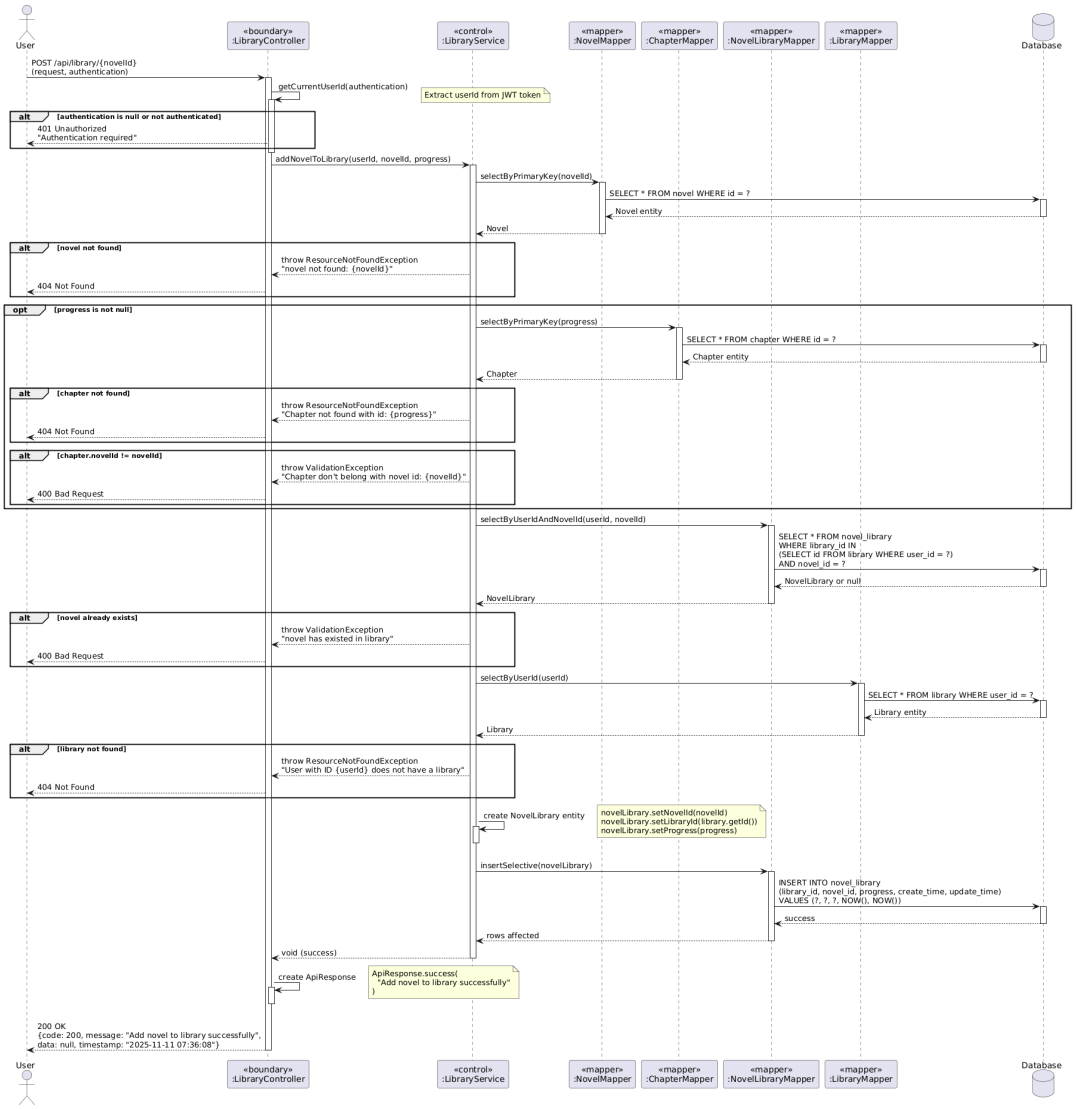
Sequence Diagram
(after applying Chain of Responsibility for validation and Strategy & Factory Method for content handling)

3.4.5 Use Case: User Add Novel to Library (Ahan Jaiswal)

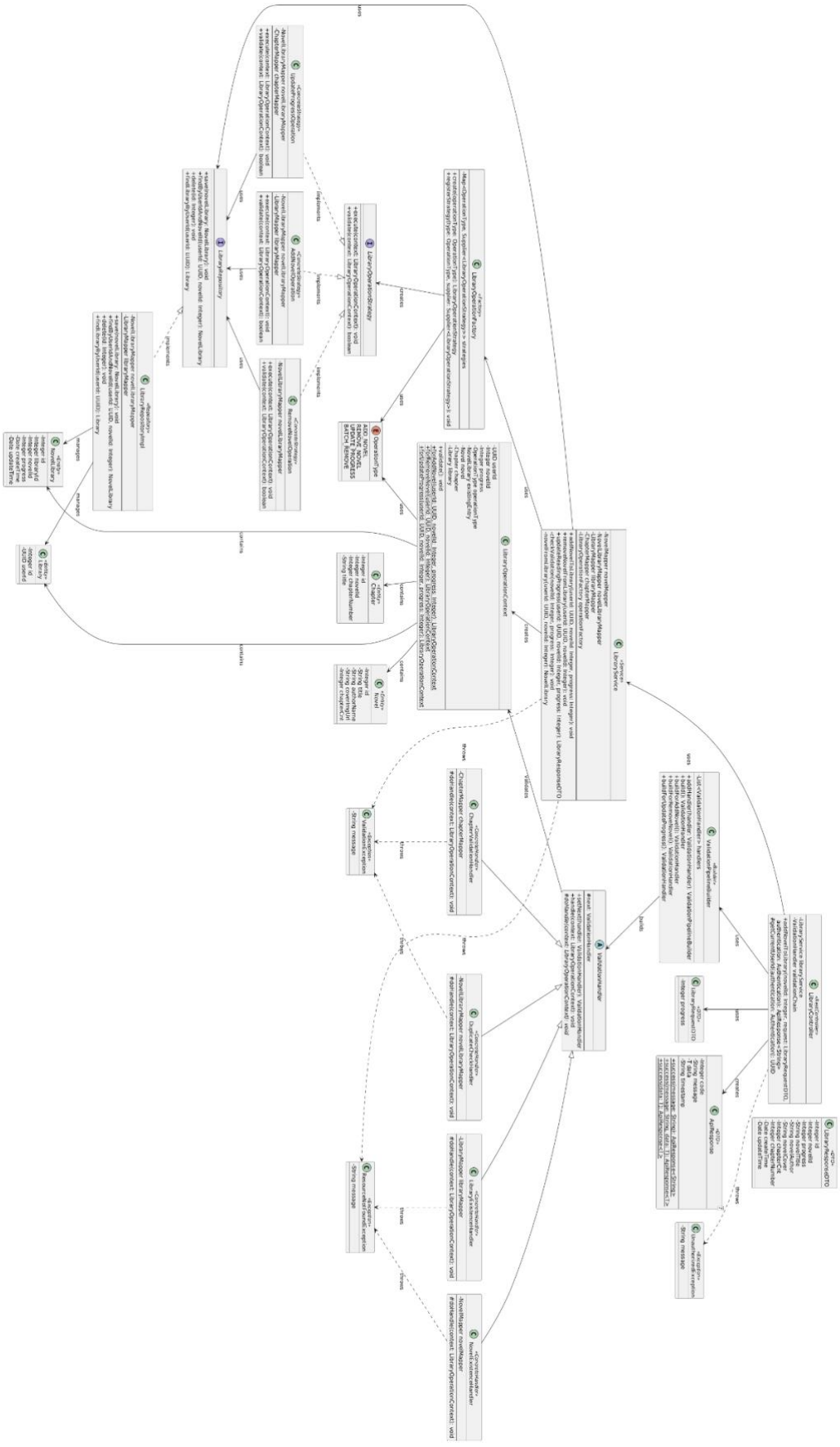
1. Analysis Model



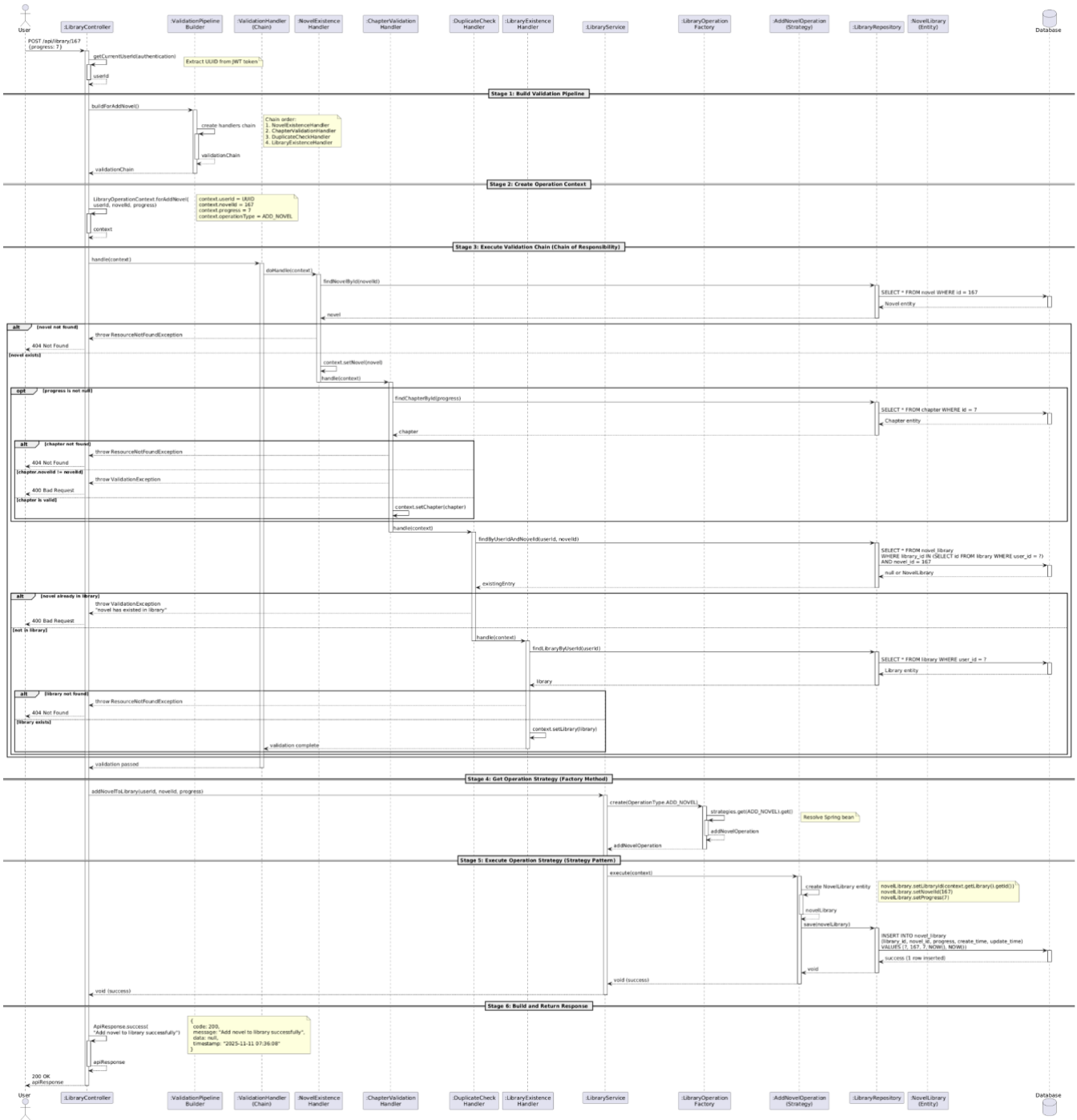
Class Diagram Above / Sequence Diagram Below



2. Design Model



Class Diagram in Previous Page (after applying Chain of Responsibility for validation and Strategy & Factory Method for library operations)



Sequence Diagram - DM above (after applying Chain of Responsibility for validation and Strategy & Factory Method for library operations)

3.5 Design Problems and Patterns

3.5.1 Design Problem & Pattern by Zhang Yan

Note: Design problems identified in this part are related to the 'User Register with Email Verification' use case.

1. **Description of design problem with class and sequence diagrams**

The core complexity in the designed flow occurs at the moment the user "clicks submit." At this point, the system must perform a complex operation. This problem can be broken down:

- The "Data Assembly" Problem: The data required to create a User comes from two different places: the email (from phase one, stored in Redis/Session) and the username, gender, etc. (from phase two's HTTP request).
- The "Atomicity of Creation" Problem: The business rules require that a User and a Library must be created together. A Library cannot exist on its own; it belongs to a User (the Composition relationship we discussed). This creation logic must be enforced.

2. **Candidate Design Patterns**

Strategy Pattern:

- Solves: The "OTP sending" mechanism.
- Application: Define an IOtpSender interface, then create EmailOtpSender and SmsOtpSender implementations. This allows you to switch the OTP delivery method (e.g., from email to SMS) without changing the main flow.
- (Note: This is a good pattern, but it solves a sub-problem (sending) rather than the core problem (creation).)

Factory Method Pattern:

- Solves: Hides the complexity of creating a User.
- Application: Create a UserFactory with a createUserWithLibrary(...) method. The service layer calls this factory instead of new User() and new Library() itself.

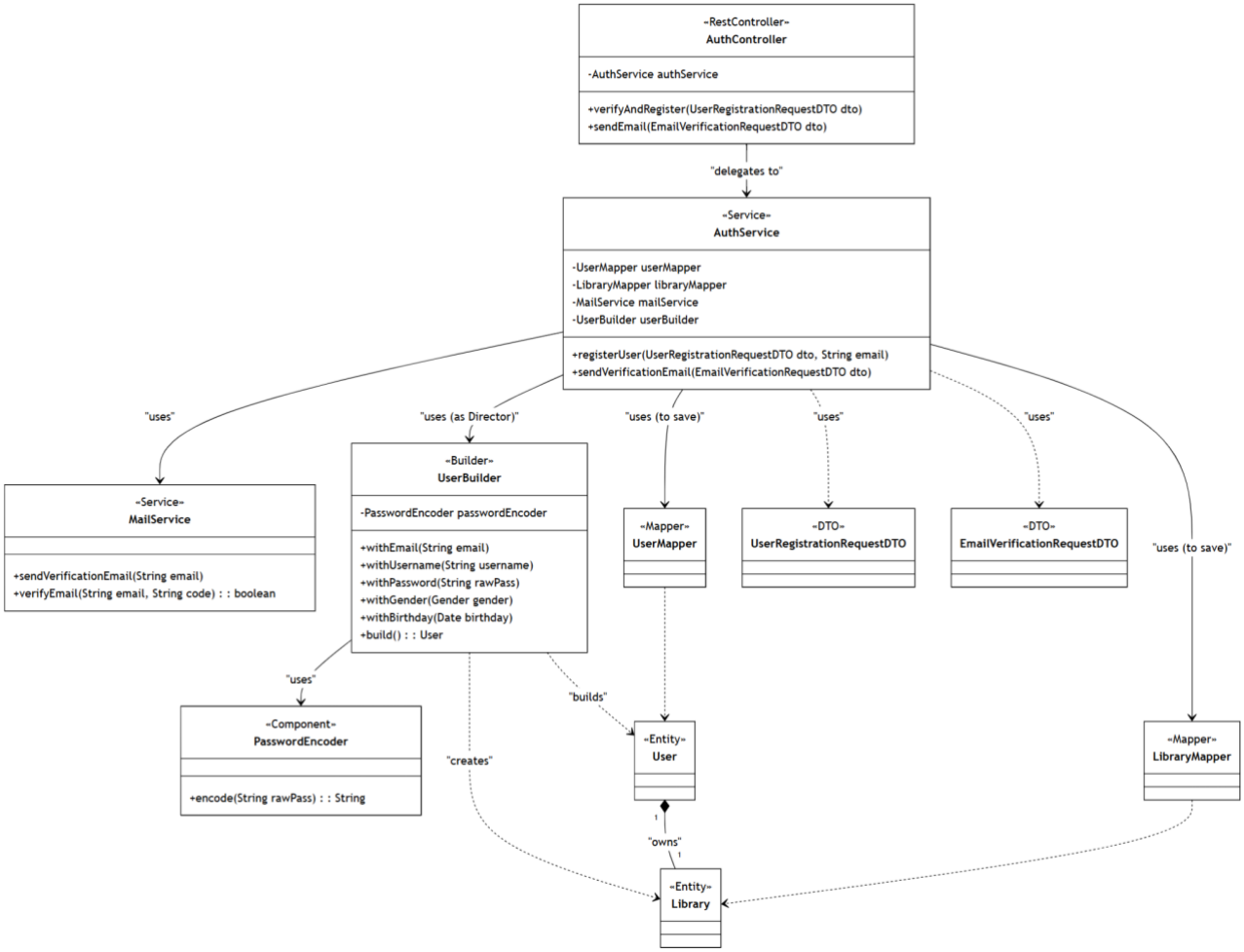
Builder Pattern:

- Solves: Gradually constructs a complex object and encapsulates its internal, complex creation logic (like the composition relationship).
- Application: Define a UserBuilder. The Service layer gathers data from Redis and the DTO, then calls builder.withEmail(...), builder.withUsername(...), etc. Finally, it calls builder.build(), and this build() method is responsible for creating both the `User` and `Library` and linking them.

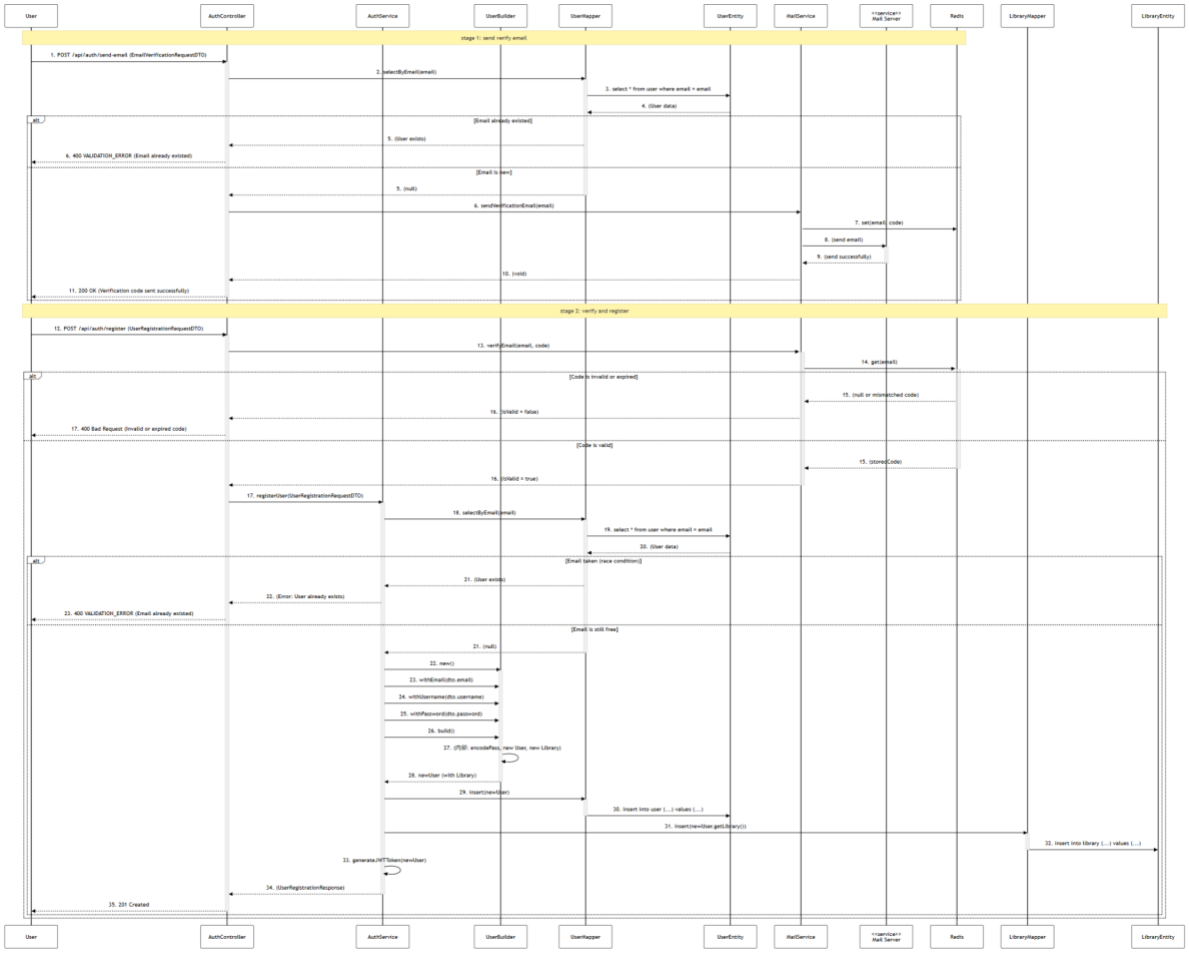
3. **Motivation to Choose Builder Pattern**

- Perfect Match for "Data Assembly": The essence of the Builder pattern is "step-by-step construction." Your service layer needs to "assemble" data from multiple sources (Redis, Request DTO). Using a Builder (e.g., builder.withEmail(...), builder.withUsername(...)) is the cleanest and most readable way to handle this.
- Encapsulates Complex Creation Logic: It perfectly solves the "atomicity" problem. All complex logic, including (1) hashing the password and (2) creating the Library and composing it with the User, can be encapsulated within the .build() method.
- High Extensibility: If you need to add dateOfBirth or address to the registration in the future, you only need to add a withAddress(...) method to the Builder. The service layer code remains almost unchanged. With a Factory, the parameter list for createUser(...) would grow longer and more unmanageable.

4. Class Diagram for the Design Solution



5. Sequence Diagram for the Design Solution



6. Implementation Decisions

1) Builder's Responsibility: The UserBuilder will be implemented as a separate class. Its .build() method will be solely responsible for:
   - Validating that all required fields have been provided.
   - Calling the PasswordEncoder to hash the password.
   - Instantiating the User object.
   - Instantiating the Library object.
   - Establishing the bi-directional association between User and Library (user.setLibrary(library) and library.setUser(user)) to fulfill the Composition relationship.
   - Setting default User status (status = ACTIVE).
2) Service's Responsibility (Director): The AuthService acts as the "Director" in this pattern. It is responsible for:
   - Coordinating with MailService and Redis to validate the OTP.
   - Collecting the data from Redis and the DTO.
   - Calling the UserBuilder to construct the final object.
   - Managing the database transaction.

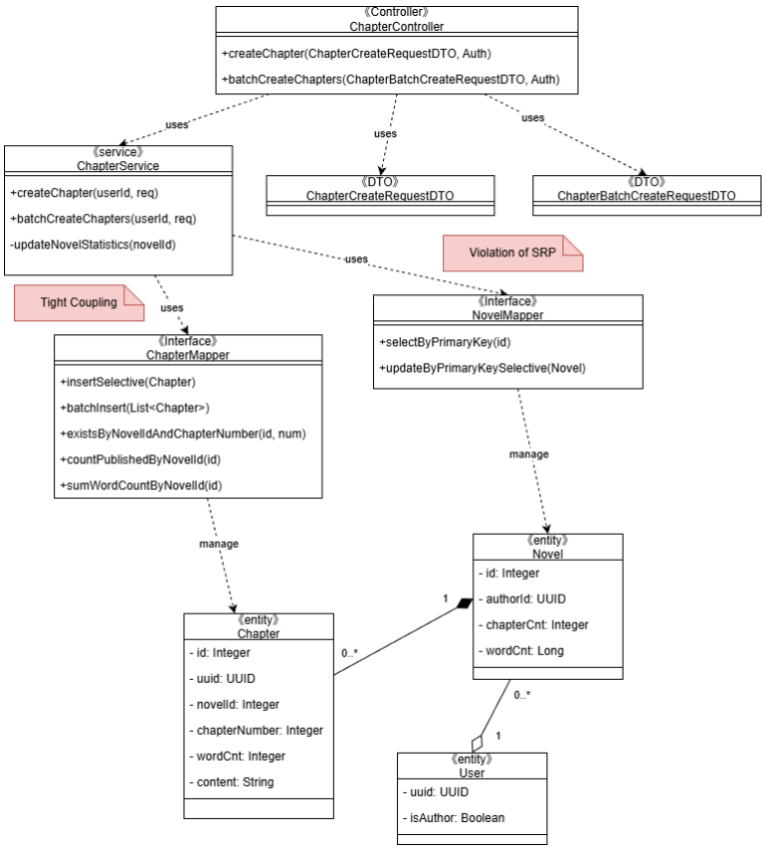
3) Transaction Management: The AuthService.registerUser() method must be annotated with @Transactional. This ensures that Step 16 (UserMapper.insert) and Step 17 (LibraryMapper.insert) either both succeed or both fail and roll back together. This is critical for maintaining data integrity.
4) State Management: Redis will be used to store the OTP and the email. The recommended approach is to use the email as the Key and the OTP as the Value, with a short expiration time (5 minutes).

3.5.2 Design Problem & Pattern by Yang Shuang

Note: Design problems identified in this part are related to the 'Author create chapter' use case.

1. Description of design problem with class and sequence diagrams

Before-refactoring diagram:



Problem summary:

- `ChapterService` directly calls `NovelMapper.selectByPrimaryKey`, `ChapterMapper.existsByNovelIdAndChapterNumber`, `ChapterMapper.insertSelective`, aggregate queries (`countPublishedByNovelId`, `sumWordCountByNovelId`), and `NovelMapper.updateByPrimaryKeySelective` in a single orchestration method.
- Business orchestration is coupled to MyBatis-specific DAOs, making technology substitutions and unit testing difficult; service mixes control logic with low-level persistence (SRP violation).

2. Candidate Design Patterns

1) Bridge Pattern

- Strength: Explicitly separates Abstraction (use case orchestration) from Implementor (persistence mechanism). Matches our exact variability axis: we foresee alternative persistence technologies and cross-cutting enhancements (e.g., cached persister) evolving independently from the orchestration logic.
- Fit: Direct mapping—`ChapterService` becomes the Abstraction; `IChapterPersister` and `INovelRepository` are Implementor interfaces; concrete classes (`MybatisChapterPersister`, `MybatisNovelRepository`) encapsulate mapper specifics.
- Outcome: Service code stops depending on MyBatis details, satisfying OCP & improving testability.

2) Adapter Pattern

- Considered for wrapping MyBatis mappers behind a uniform interface.
- Limitation: Adapter fixes API mismatches but does not establish the two-layer evolution model (it would still leave service depending on adapted concrete classes unless we introduce an additional abstraction).

3) Composite Pattern

- Typical domain fit would require hierarchical content (e.g., Volume → Chapter). Our current model (Novel → Chapters) is a simple one-level aggregation; there is no recursive structure or polymorphic "component" operations.
- Rejected: Does not address technology coupling; introduces abstraction without real hierarchical variability.

4) Decorator Pattern

- Useful for incremental "frills" (caching, metrics, auditing) around persistence operations.
- Not solving the core problem: Decorator augments an existing interface; it does not create the initial abstraction boundary we lack. We can still apply Decorator later on top of the Bridge interfaces—so Bridge is foundational, Decorator is optional.

5) Memento Pattern

- Pertains to capturing and restoring object state (e.g., undo chapter edits).
- Not relevant to chapter creation transactional persistence; no requirement for undo/rollback beyond standard DB transaction semantics. Adds complexity without benefit for current use case.

6) Observer Pattern

- Candidate for decoupling post-creation side effects (e.g., asynchronous statistic updates, notifications).
- Our immediate issue is synchronous structural coupling inside the service, not event dissemination.

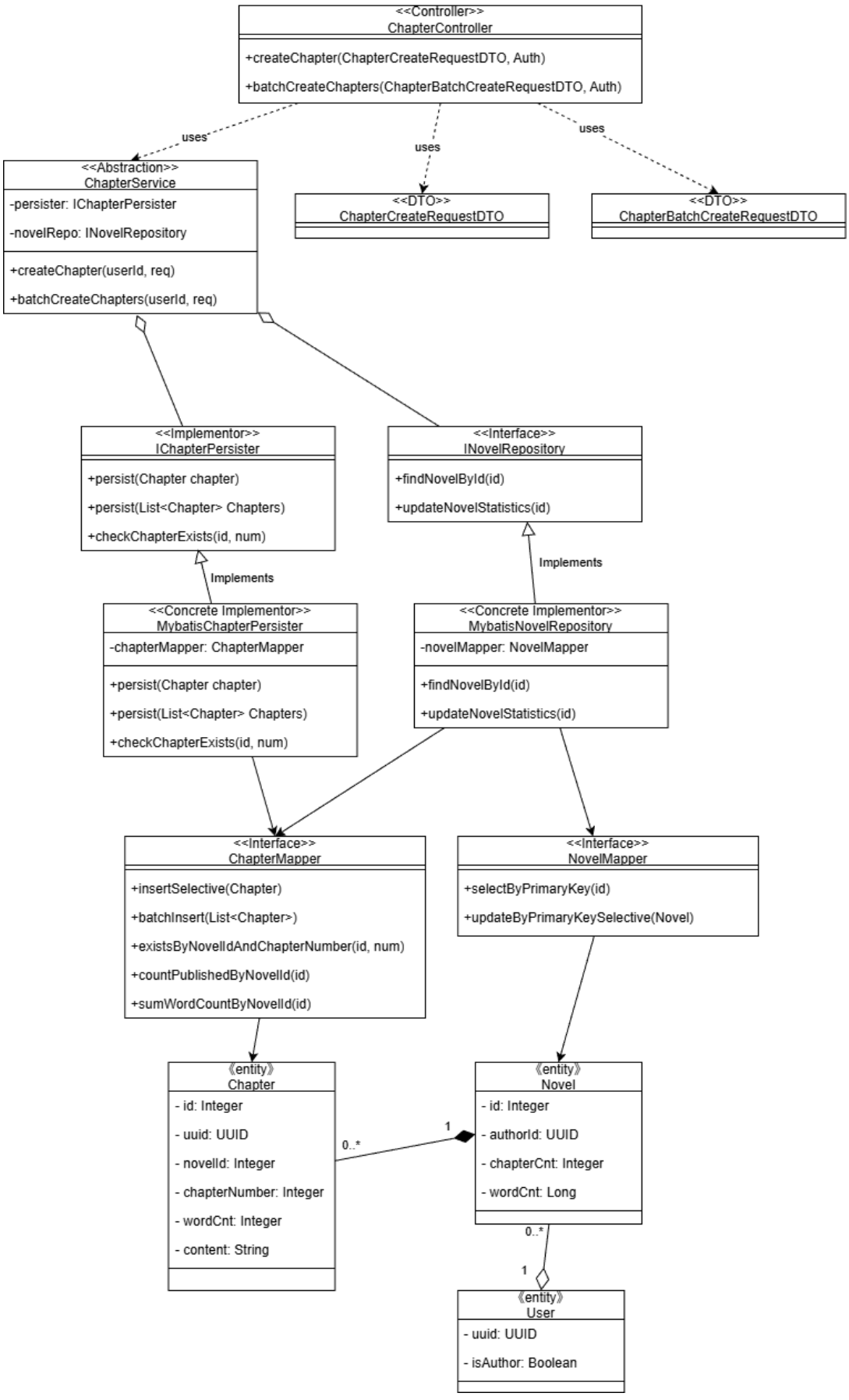
7) Command Pattern

- Encapsulates an action as an object (could queue, log, retry).
- Does not inherently solve persistence technology decoupling; we would still face the same coupling inside the command handler. Overhead unjustified for the current straightforward synchronous HTTP call.

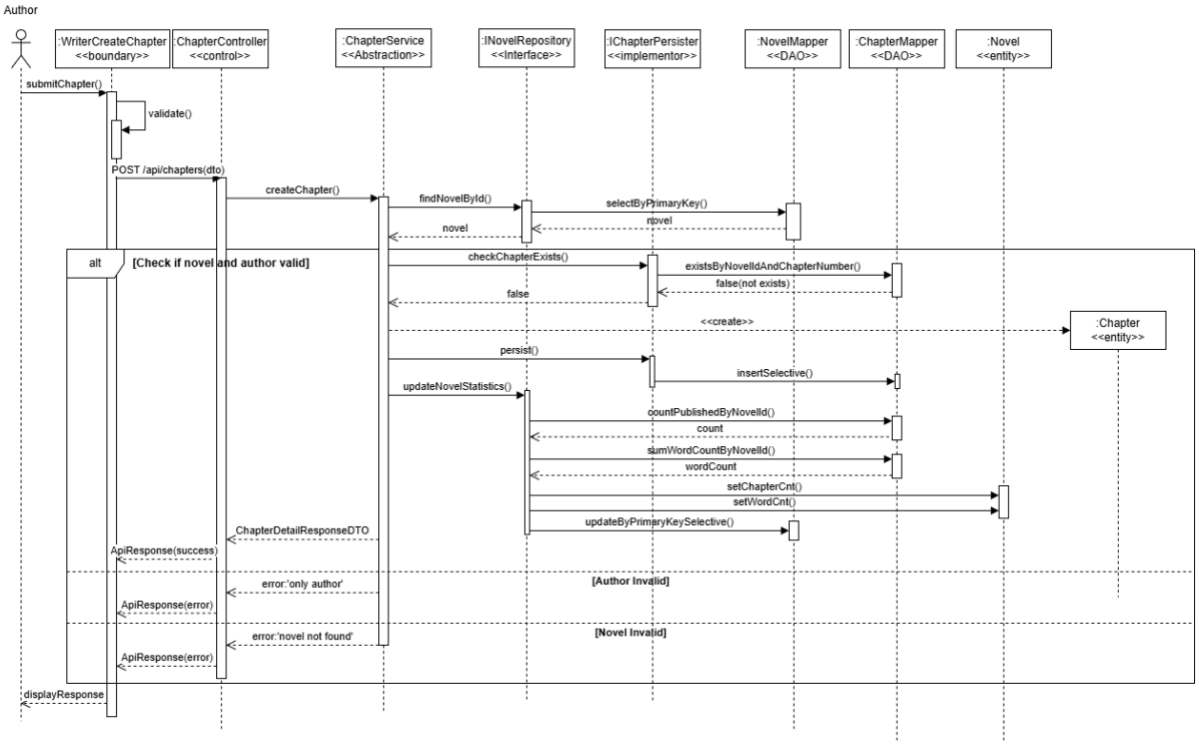
3. Motivation to Choose Bridge

It is the only pattern in this list whose primary intent directly matches our main structural deficiency—entanglement of abstraction (use case orchestration) with implementation (specific MyBatis mappers). Bridge cleanly introduces an indirection layer that supports independent evolution, unit testing without DB, and painless addition of persistence variants.

4. Class Diagram for the Design Solution



5. Sequence Diagram for the Design Solution



6. Implementation Decisions

1) Interface Definitions

- `IChapterPersister`: `persist(Chapter)`, `persist(List<Chapter>)`, `checkChapterExists(novelId, chapterNumber)`.
- `INovelRepository`: `findNovelById(id)`, `updateNovelStatistics(id)`.

2) Concrete Implementations

- `MybatisChapterPersister` uses `ChapterMapper.insertSelective`, `batchInsert`, and `existsByNovelIdAndChapterNumber`.
- `MybatisNovelRepository` uses `NovelMapper.selectByPrimaryKey` and `NovelMapper.updateByPrimaryKeySelective`; also uses `ChapterMapper.countPublishedByNovelId` and `ChapterMapper.sumWordCountByNovelId` to compute aggregates before updating Novel.

3) Dependency Injection & Packaging

- `ChapterService` (application/service layer) receives `IChapterPersister` and `INovelRepository` via constructor injection (Spring).
- Concrete implementors and mappers live in infrastructure/persistence packages.

4) Transaction Boundary

- `@Transactional` on `ChapterService.createChapter` and `batchCreateChapters` so that “persist chapter(s) + statistics update” is atomic.

5) Error Handling

- Novel not found → 404; author invalid → 403; request validation failures → 400.
- Controller uses global exception handling to translate domain exceptions to HTTP responses; sequence diagrams already include alt lanes for error cases.

6) Backward Compatibility

- With only MyBatis implementors wired, behavior matches the “before” sequence ; the service API to controllers and DTOs remains unchanged.

7) Testability

- Unit tests mock `IChapterPersister` and `INovelRepository` and assert orchestration and alt branches (novel missing, unauthorized author, duplicate chapter).
- Integration tests run MyBatis implementors with an in-memory database to verify transactional consistency and aggregate updates.

8) Extensibility

- New persistence technology (e.g., an alternative repository) is added as new concrete implementors without touching `ChapterService`.
- Optional cross-cutting concerns (e.g., metrics/logging) can wrap implementors if needed, without changing the service interface.

3.5.3 Design Problem & Pattern by Zhu Yuhui

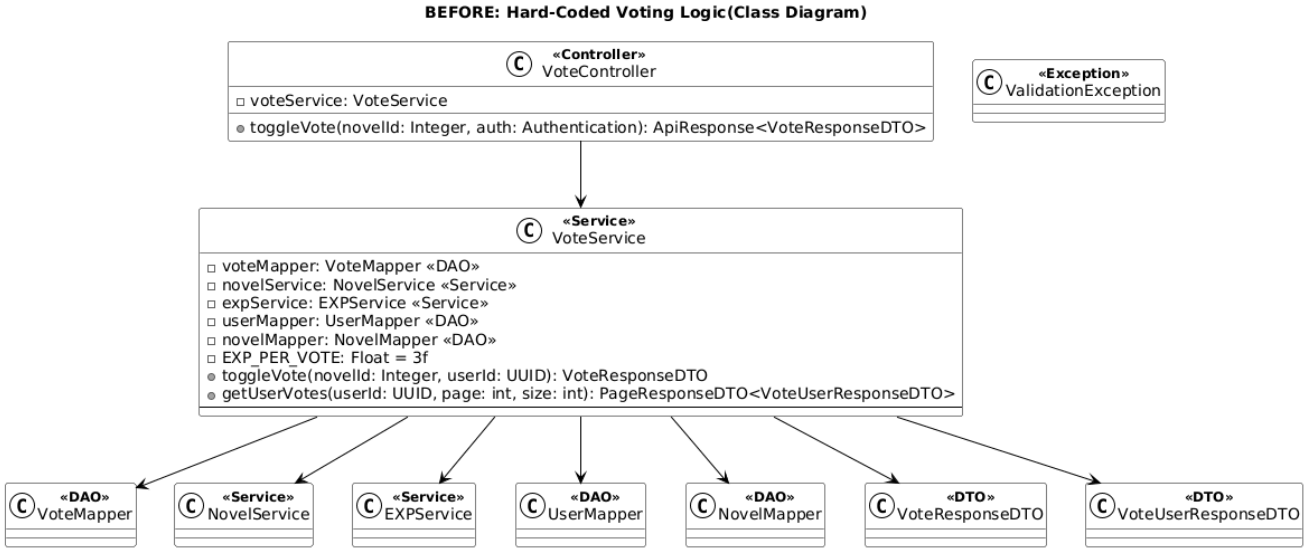
Note: Design problems identified in this part are related to the 'User Vote for Novel' use case.

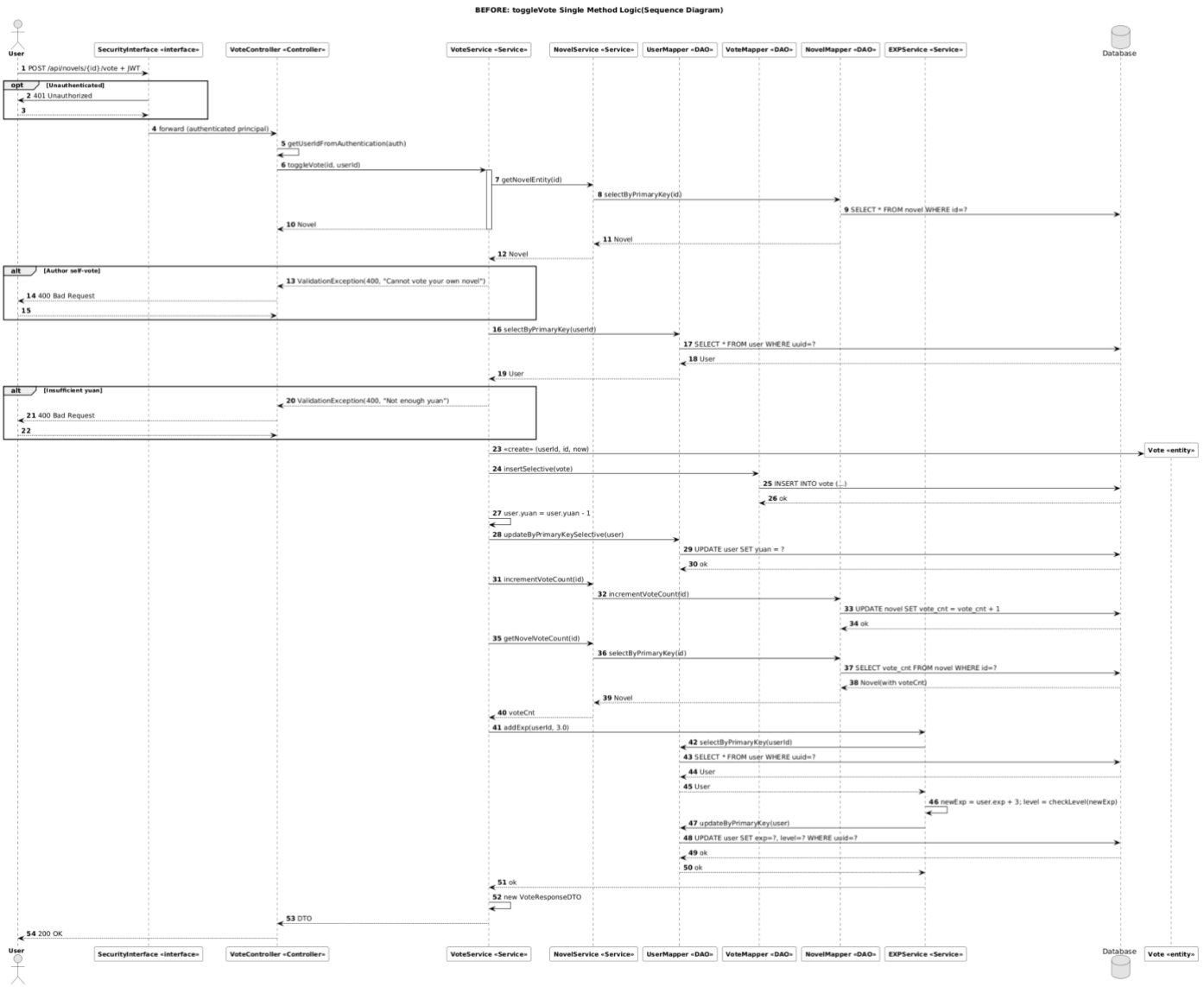
1. Description of design problem with class and sequence diagrams

`VoteService.toggleVote` hard-codes all rules:

- author self-vote check, user balance check
- fixed cost (1 yuan), fixed reward (3 EXP) and level recomputation
- persistence sequencing in one transactional block

This makes it hard to change cost/reward/validation without editing `VoteService`, violating OCP and harming testability.





2. Candidate Design Patterns

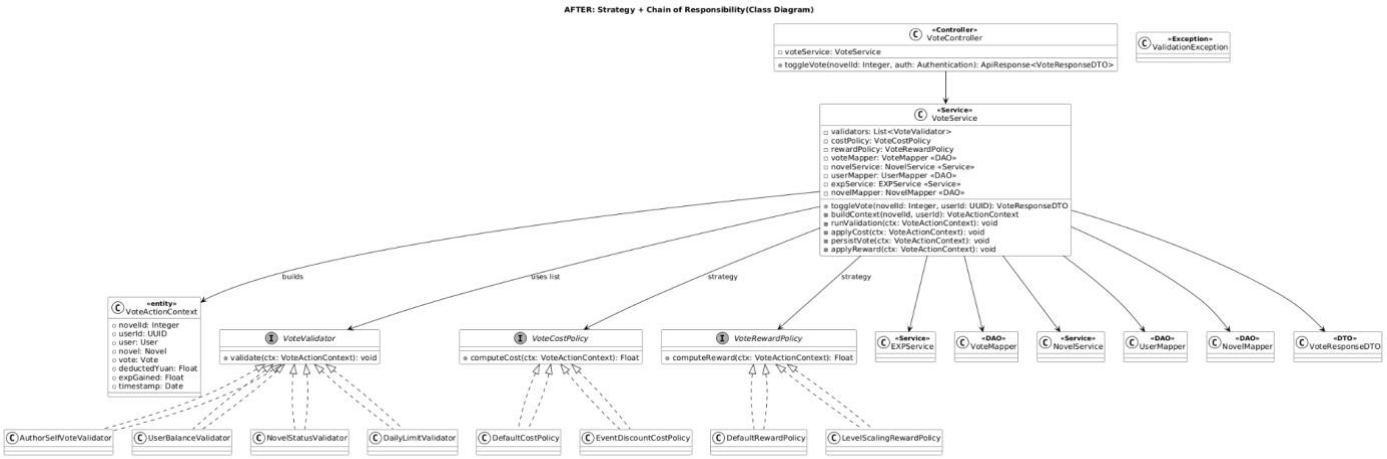
- Strategy Pattern: For pluggable cost and reward policies (vote cost, EXP calculation).
- Chain of Responsibility Pattern: For an extensible pipeline of validation rules executed before applying cost & reward.
- Template Method Pattern: To define a generic “vote action” skeleton with hook methods (less flexible when combining multiple orthogonal variations).
- Decorator Pattern: For stacking cross-cutting responsibilities (e.g., logging, metrics), but not ideal for domain rule branching.
- Command Pattern: To encapsulate the vote action; helpful for queuing/retry, but does not directly solve rule variability.
- Observer Pattern: For emitting domain events (e.g., VoteCreatedEvent) to decouple ranking/achievement updates.

3. Selected Patterns & Motivation

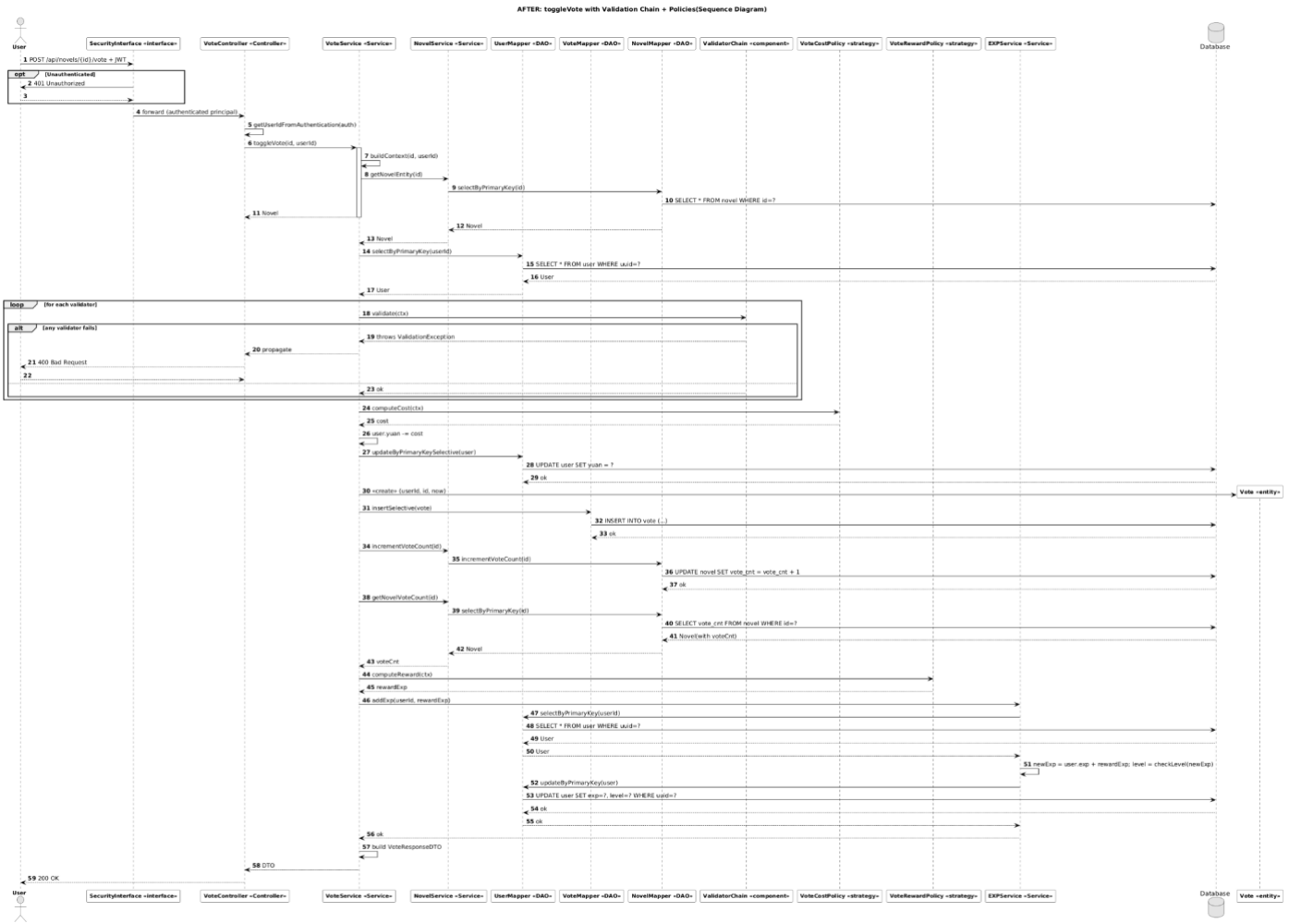
Chosen combination: Strategy + Chain of Responsibility. Motivation:

- Strategy separates variation points (cost, reward) and enables runtime swap or conditional selection.
- Chain of Responsibility composes validations and allows adding/removing validators without touching `VoteService`.
- This combination scales better than inheritance (Template Method) and is clearer than stacking many decorators.
    - Template Method declined because different combinations of validators and policies scale better with composition than inheritance.
    - Decorator declined for domain rule orchestration (would become convoluted for conditional abort).

4. Class diagram for the design solution



5. Sequence diagram for the design solution



6. Implementation decisions

1) Bean Assembly

- Validators: Each validator is a Spring `@Component` that implements `VoteValidator` with a single method `validate(VoteActionContext ctx)`.
- Ordering: Use `@Order` (or `Ordered`) so Spring injects `List<VoteValidator>` in a deterministic pipeline order.
- Policies: `VoteCostPolicy` and `VoteRewardPolicy` are Spring `@Component`s. Use `@Primary`/`@Qualifier` or a simple `PolicyRegistry` to select the active policy implementations.

2) Strategy Resolution

- Default strategy: Cost policy returns `1f`, reward policy returns `3f` to reproduce current behavior.
- Extensibility: Future selection can be made dynamic (e.g., by user role, event flags, time windows) via a registry or conditional beans.

3) Transaction Scope

- Keep `@Transactional` on `VoteService.toggleVote`.
- Validators and policies must be side-effect free (no direct persistence or external commits). All writes are coordinated by `VoteService` (and `EXPService` for EXP/level updates) to maintain atomicity.

4) Exception Semantics

- Business rule failures in validators throw `ValidationException` → mapped to HTTP 400.
- Missing resources (e.g., novel not found) throw `ResourceNotFoundException` → mapped to HTTP 404.
- The controller does not catch these; they are handled by global exception handling to produce proper HTTP responses.

5) Testability

- Unit tests: Test each `VoteValidator`, `VoteCostPolicy`, and `VoteRewardPolicy` in isolation using mocked DAOs, with a prebuilt `VoteActionContext`.
- Integration tests: Verify validator ordering, policy selection, and that vote creation, balance deduction, vote count increment, and EXP update occur atomically within one transaction.

6) Backward Compatibility

- With only `DefaultCostPolicy(1f)` and `DefaultRewardPolicy(3f)` registered, the system exactly replicates current repository behavior. This enables a safe migration to the “after” design.

7) Performance

- Build `VoteActionContext` once at the start of `toggleVote`, loading `User` and `Novel` upfront. Validators and policies read from the context to avoid extra DAO round-trips.
- Use incremental DB updates for aggregate values (e.g., `NovelMapper.incrementVoteCount`) as the current code does.

8) Extensibility

- Adding a new rule requires only adding a new `VoteValidator` bean; `VoteService` does not change.
- Changing the reward computation requires only swapping or adding a `VoteRewardPolicy` implementation; no changes to `VoteService`.
- The pattern keeps domain rules pluggable and the orchestration stable.

3.5.4 Design Problem & Pattern by Nguyen Phu Truong

Note: Design problems identified in this part are related to the ‘User Report Novel & Comment’ use case.

1. Decoupling Validation Rules (Chain of Responsibility)

A. Design Problem Description

- Context: In the analysis model, ReportService performs all validation inline (check novel/comment existence, validate report type, detect duplicates).

- Pain points:
    - Validation rules are tightly coupled to the service and appear as sequential if blocks.
    - Adding/removing rules requires editing the service directly, risking regressions.
    - Hard to unit-test or reuse individual rules.
- Class Diagram and Sequence Diagram Reference in Analysis Model in 3.4.4.

B. Candidate Design Patterns

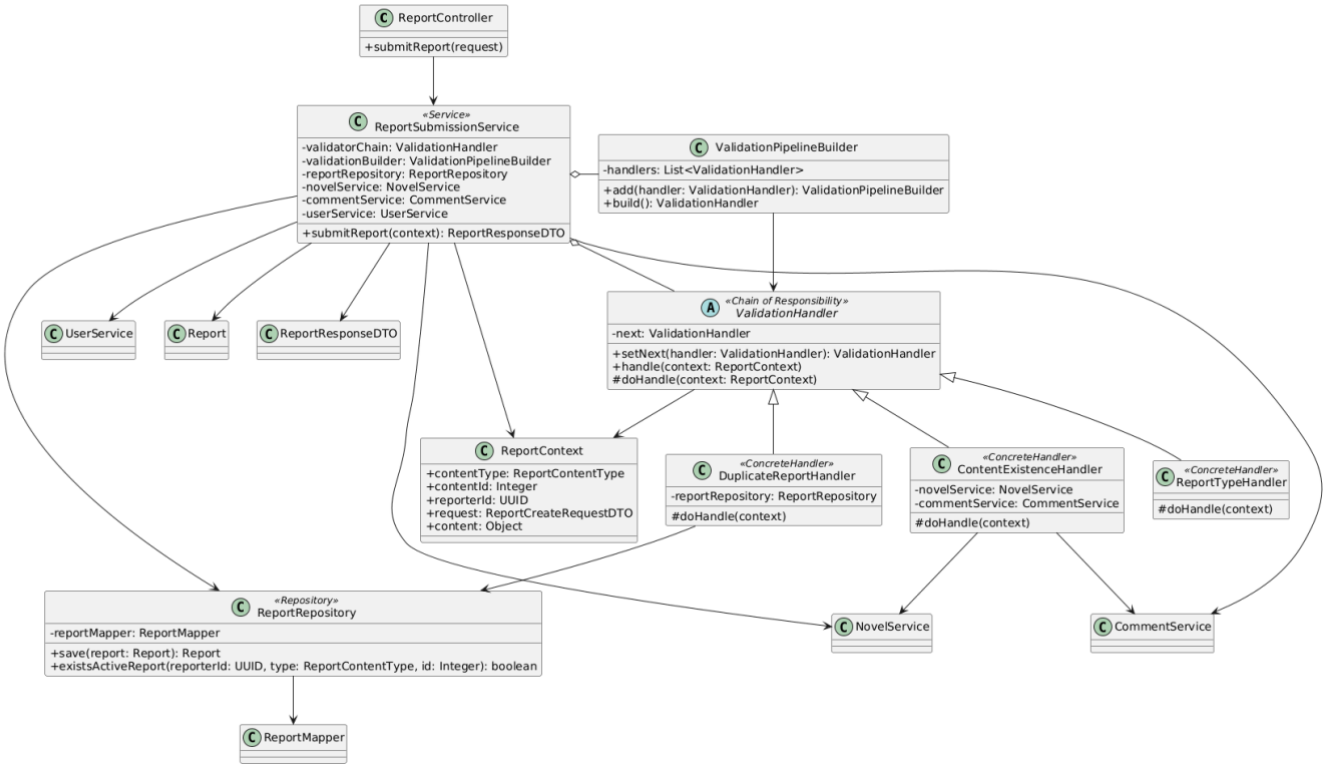
- Template Method (encapsulate validation skeleton in abstract base class).
- Strategy (one strategy per validation policy).
- Chain of Responsibility (link handlers dynamically and stop processing on failure).

C. Motivation for Chosen Pattern

- Validation rules must run in order and be able to short-circuit when a rule fails → Chain of Responsibility matches this requirement naturally.
- Allows new handlers to be inserted without changing existing ones.
- Keeps the service focused on orchestration rather than individual checks.

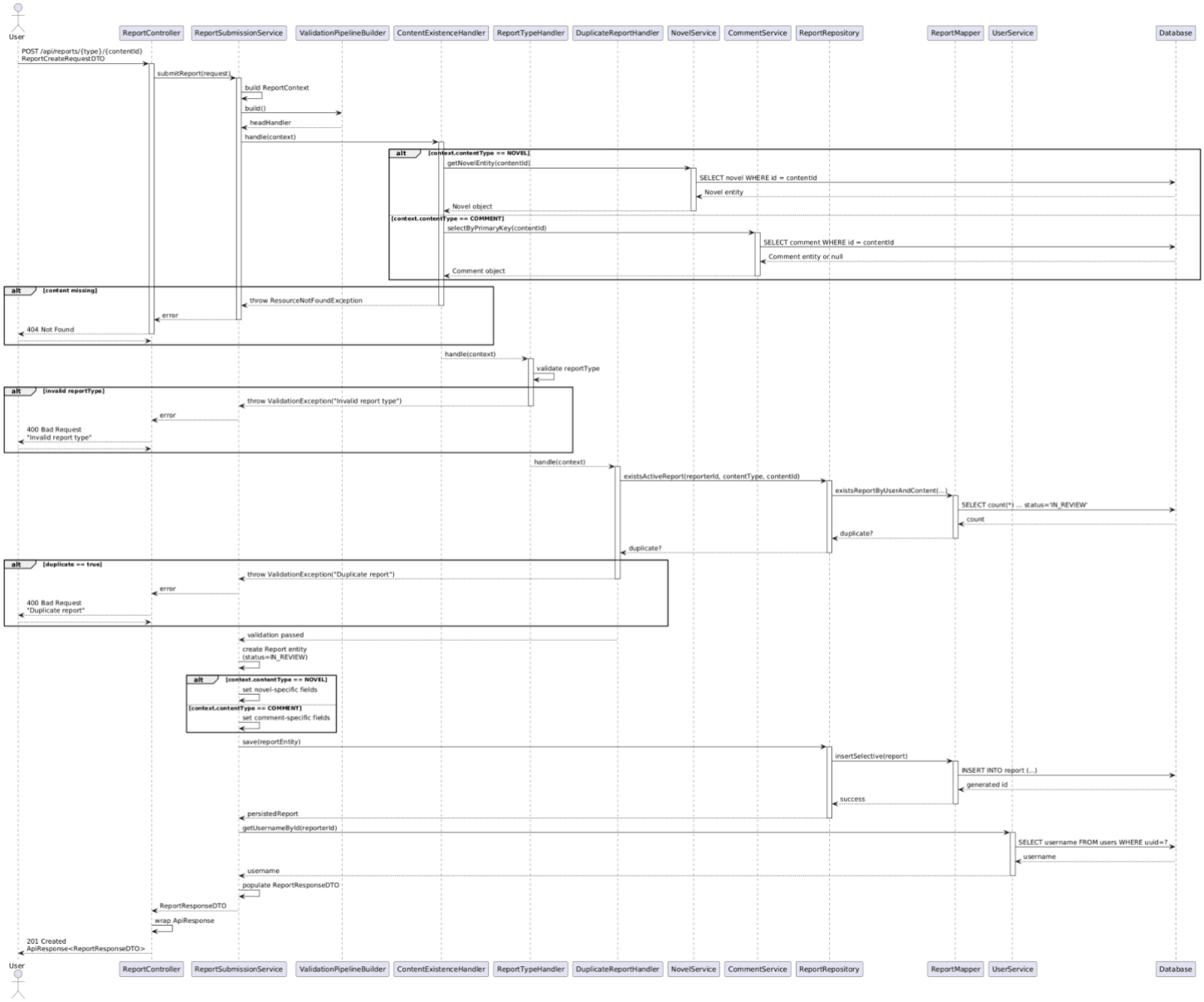
D. Class Diagram for the design solution

- Introduce ValidationHandler abstract class with handle() delegating to doHandle() and setNext().
- Use ValidationPipelineBuilder to assemble the handler chain (ContentExistenceHandler, ReportTypeHandler, DuplicateReportHandler).
- ReportSubmissionService obtains the root handler from the builder and executes validation before constructing the report.



E. Sequence Diagram for the design solution

- Key flow: Controller → Service → Builder → Handler chain → Repository.



F. Implementation Decisions

- Create new classes: ValidationHandler (abstract), ContentExistenceHandler, ReportTypeHandler, DuplicateReportHandler, ValidationPipelineBuilder.
- Extract persistence-related checks into ReportRepository (wraps ReportMapper) so handlers avoid direct mapper access.
- Refactor service to build a ReportContext and call validatorChain.handle(context) before proceeding.

2. Eliminating Content-Type Branching (Strategy Pattern)

A. Design Problem Description

- Context: ReportSubmissionService still contains if (NOVEL) vs if (COMMENT) branches to build Report objects and enrich DTOs.
- Pain points:
    - Duplicated logic per content type.

- Difficult to add new content types (e.g., chapter) without modifying the service.
- Testing individual behaviours requires service-level tests.

- Class Diagram and Sequence Diagram Reference in Analysis Model in 3.4.4.

B. Candidate Design Patterns

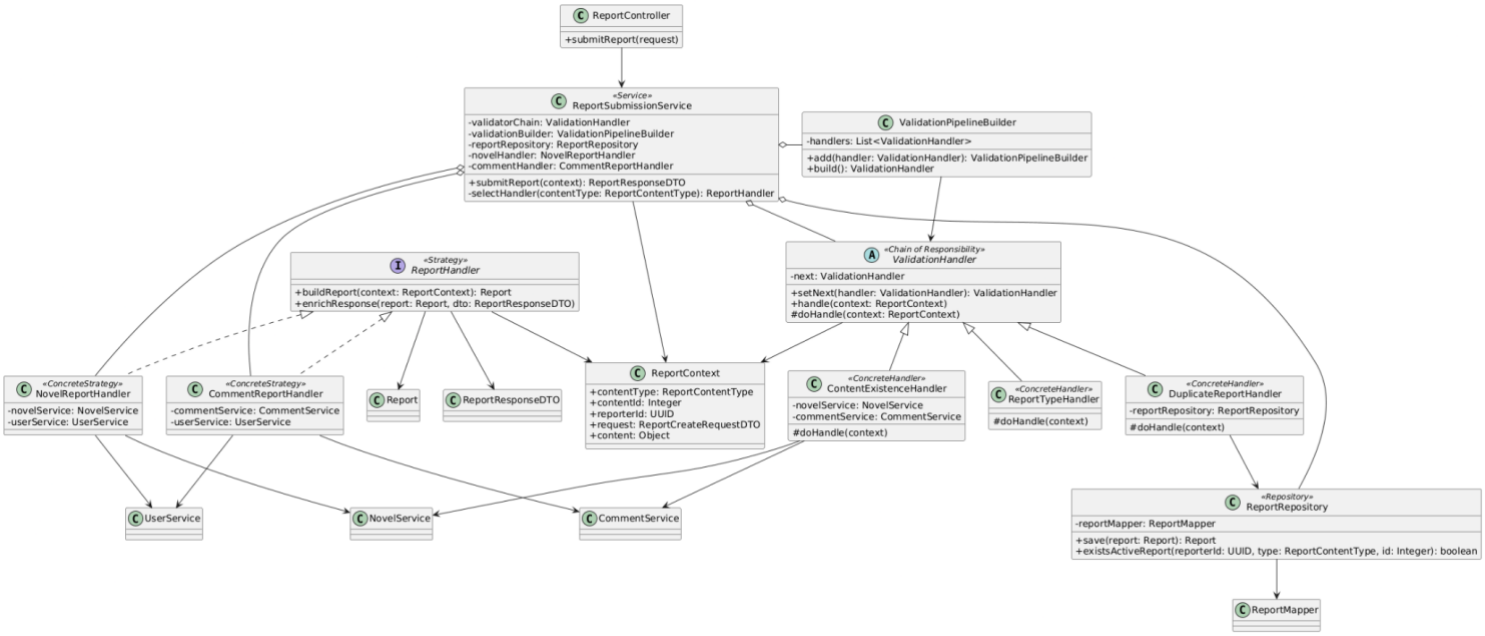
- Template Method (base class with hook for content-specific steps).
- Strategy (separate object per content type).

C. Motivation for Chosen Pattern

- Strategy keeps the service ignorant of concrete content types while allowing each strategy to fetch dependencies (NovelService, CommentService) freely.
- Swapping or adding strategies (e.g., ChapterReportHandler) requires no changes to the service core.
- Template Method would still require inheritance hierarchy inside the service, less flexible for injection.

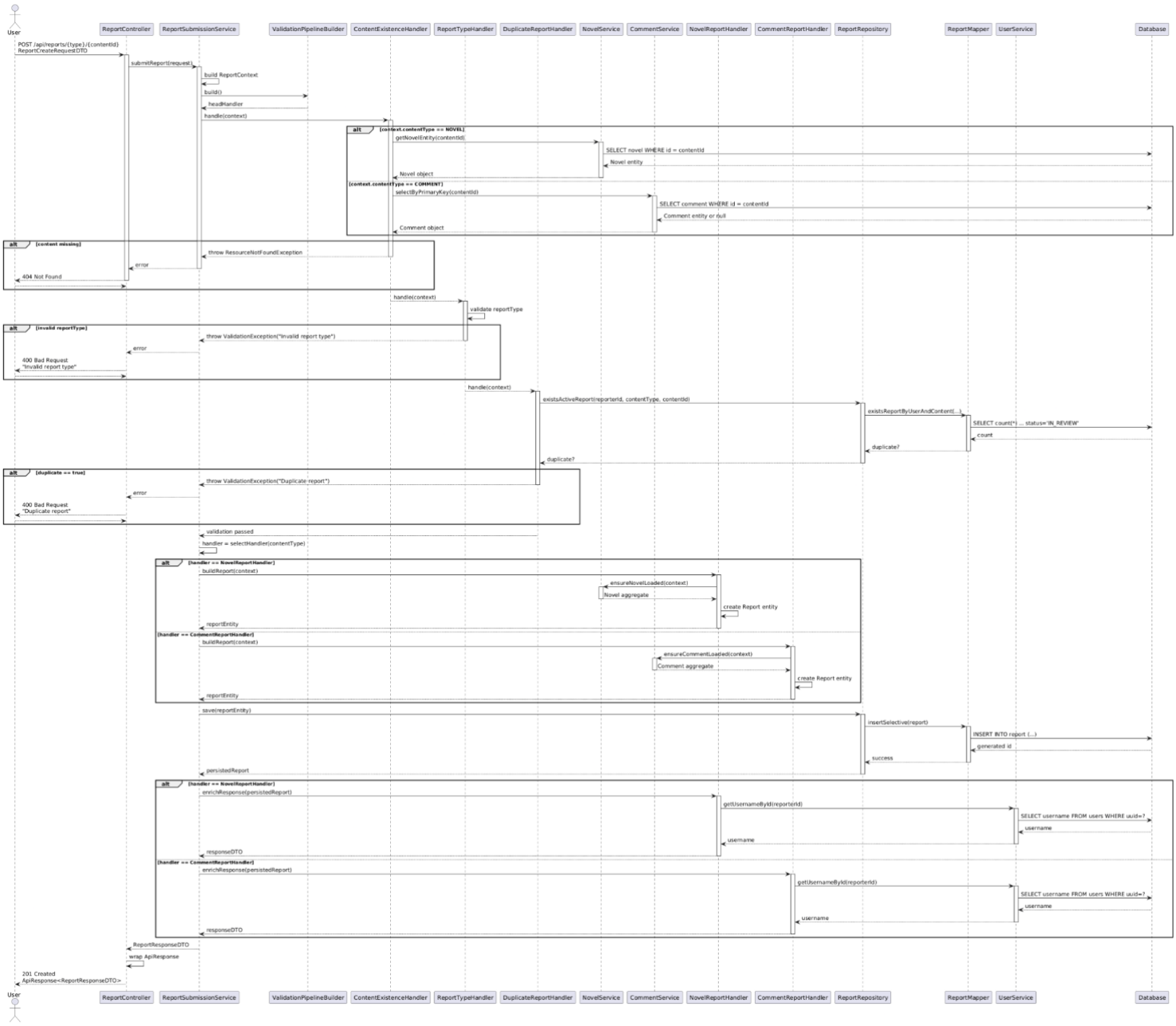
D. Class Diagram for the design solution

- Define ReportHandler interface with buildReport(context) and enrichResponse(report, DTO).
- Implement NovelReportHandler and CommentReportHandler, each injecting required services.
- ReportSubmissionService uses a selectHandler method (later replaced by factory) to choose the strategy after validation.



E. Sequence Diagram for the design solution

- Service delegates report creation and response enrichment to the chosen handler.



F. Implementation Decisions

- Register strategy beans in the Spring container and inject them into the service.
- Ensure handlers reuse shared DTO/entity mapping logic.
- Keep validation pipeline unchanged; strategies are invoked only after successful validation.

3. Centralizing Handler Instantiation (Factory Method)

A. Design Problem Description

- Context: ReportSubmissionService still holds direct references to concrete strategies and a selectHandler method with conditional logic.
- Pain points:
    - Service constructors continue to grow as more strategies are added.
    - Strategy instantiation logic is scattered; no central place to manage new handler registrations.
    - Harder to support variations (e.g., caching, lazy creation).
- Class Diagram and Sequence Diagram Reference in Analysis Model in 3.4.4.

B. Candidate Design Patterns

- Simple Factory (static helper returning handler instances).
- Abstract Factory (if multiple families of handlers were needed).
- Factory Method (inject suppliers and delegate creation through an interface).

C. Motivation for Chosen Pattern

- Factory Method allows the service to depend on an interface and makes it easy to inject new handler suppliers without modifying service code.
- Keeps alignment with dependency-injection style; each handler can still be a Spring bean.
- Abstract Factory would be overkill; Simple Factory would centralize logic but remain static and less testable.

D. Class Diagram for the design solution

- Introduce ReportHandlerFactory with create(contentType); internally stores suppliers for each supported type.
- Refactor ReportSubmissionService to request handlers from the factory instead of owning them.
- Maintain Chain of Responsibility and Strategy from previous steps.
- Class Diagram Reference in Design Model in 3.4.4.

E. Sequence Diagram for the design solution

- Sequence Diagram Reference in Design Model in 3.4.4.

F. Implementation Decisions

- Provide factory configuration in Spring (e.g., map ReportContentType to suppliers that resolve existing strategy beans).
- Update existing unit tests to mock the factory instead of individual handlers.
- Ensure ReportSubmissionService.resolveReport orchestrates moderation side-effects (novel reverted to DRAFT, comment deleted) before persisting status updates.

3.5.5 Design Problem & Pattern by Ahan Jaiswal

Note: Design problems identified in this part are related to the 'User Add Novel to Library' use case.

1. Decoupling Validation Rules (Chain of Responsibility)

Design Problem Description

Context: In the analysis model, LibraryService performs all validation inline (check novel existence, chapter existence, chapter-novel relationship, library existence, duplicate check).

Pain points:

- Validation rules are tightly coupled to the service as sequential if blocks
- Adding/removing validation rules requires editing the service directly, risking regressions
- Hard to unit-test or reuse individual validation rules
- Difficult to change validation order without modifying service code

Class Diagram and Sequence Diagram Reference: Analysis Model in 3.4.5.

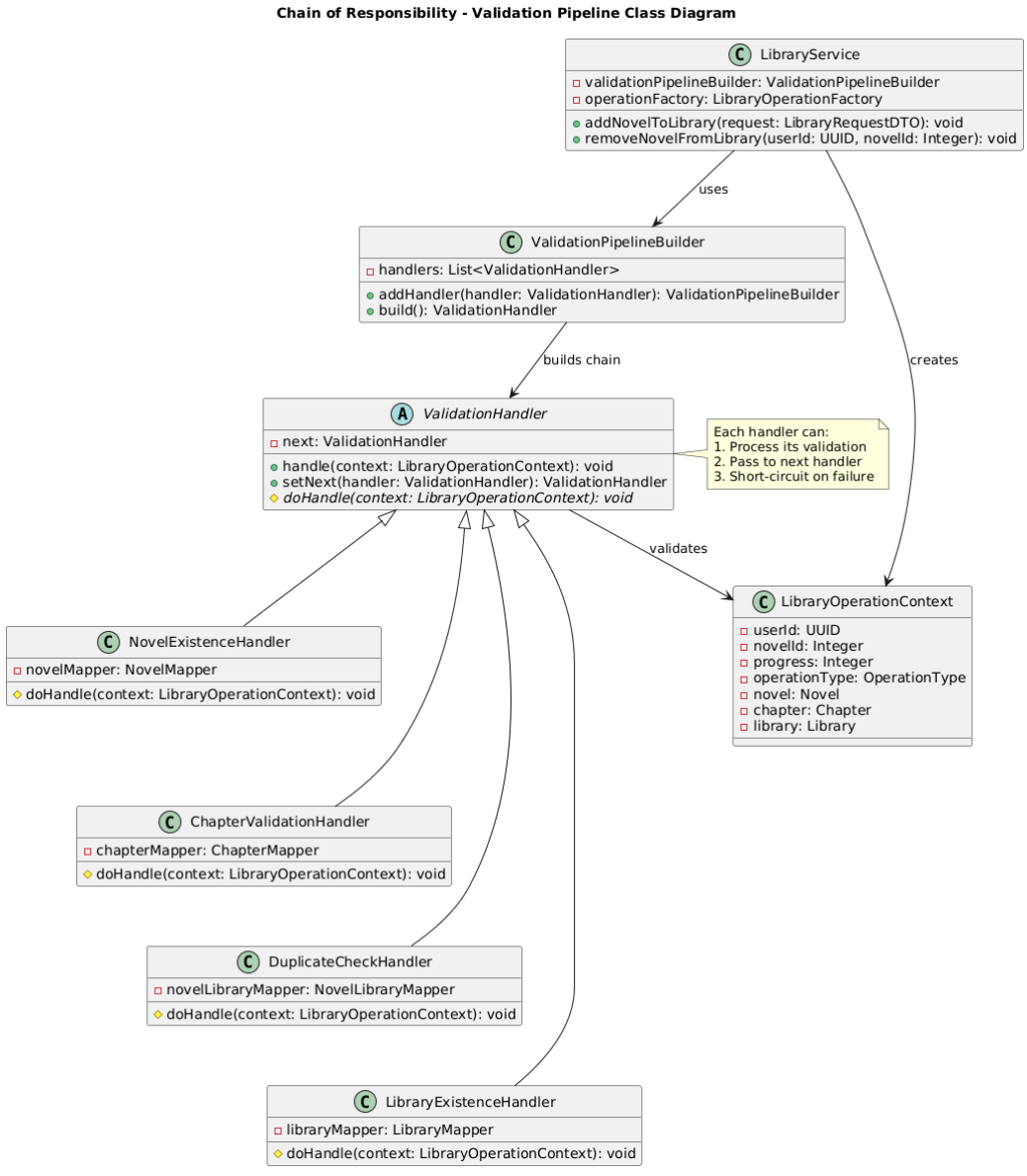
Candidate Design Patterns

- **Template Method** (encapsulate validation skeleton in abstract base class)
- **Strategy** (one strategy per validation policy)
- **Chain of Responsibility** (link handlers dynamically and stop processing on failure)

Motivation for Chosen Pattern

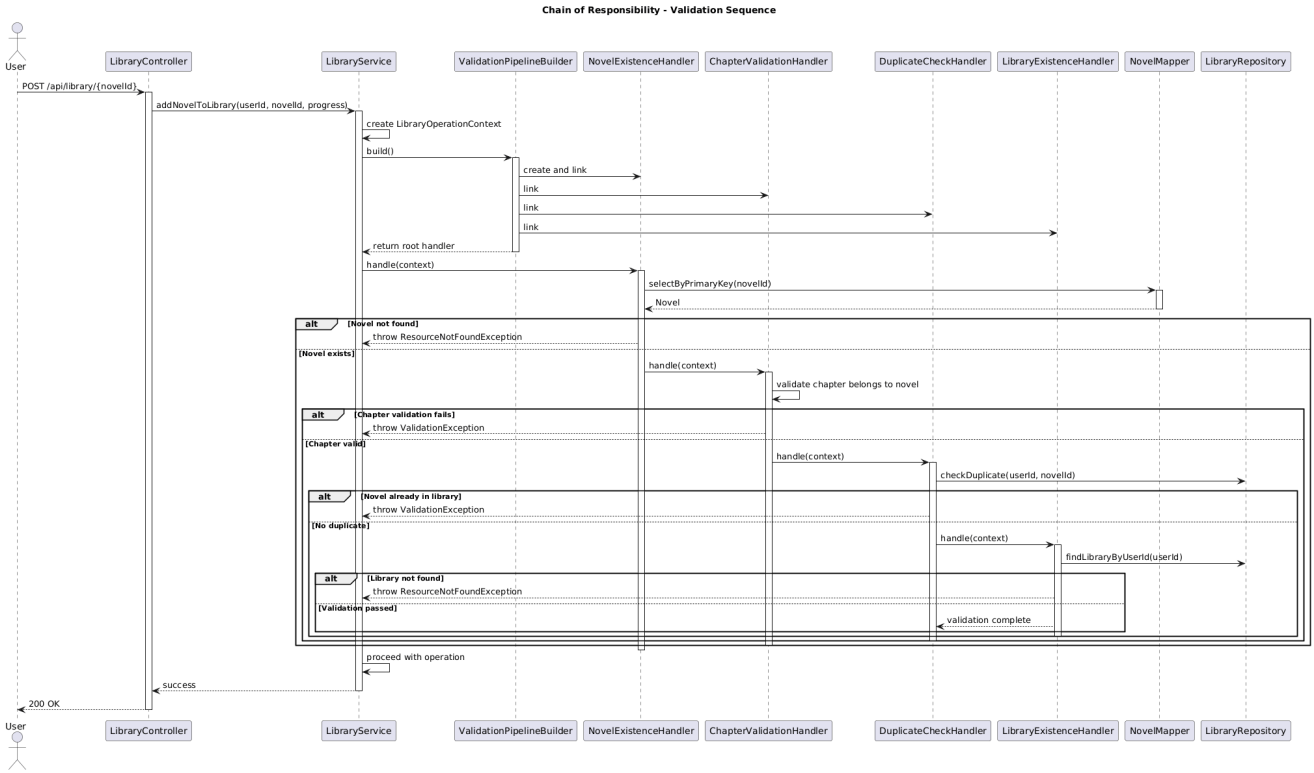
- Validation rules must run in a specific order and be able to short-circuit when a rule fails → **Chain of Responsibility** matches this requirement naturally
- Allows new handlers to be inserted without changing existing ones
- Keeps the service focused on orchestration rather than individual validation checks
- Each validator can be independently tested and reused

Class Diagram for the Design Solution



Introduce ValidationHandler abstract class with handle() delegating to doHandle() and setNext(). Use ValidationPipelineBuilder to assemble the handler chain (NovelExistenceHandler, ChapterValidationHandler, DuplicateCheckHandler, LibraryExistenceHandler). LibraryService obtains the root handler from the builder and executes validation before performing library operations.

Sequence Diagram for the Design Solution



Key flow: Controller → ValidationPipelineBuilder → Handler chain → Repository → Service.

Implementation Decisions

- Create new classes: ValidationHandler (abstract), NovelExistenceHandler, ChapterValidationHandler, DuplicateCheckHandler, LibraryExistenceHandler, ValidationPipelineBuilder
- Extract persistence-related checks into LibraryRepository (wraps LibraryMapper and NovelLibraryMapper) so handlers avoid direct mapper access
- Refactor service to build a LibraryOperationContext and call validatorChain.handle(context) before proceeding with library operations

---

2. Eliminating Operation-Type Branching (Strategy Pattern)

Design Problem Description

Context: LibraryService contains different logic branches for various library operations (add novel, remove novel, update progress, batch remove) within the same service class.

Pain points:

- Duplicated validation and execution logic per operation type

- Difficult to add new library operations (e.g., move to folder, mark as favorite) without modifying the service
- Testing individual operation behaviors requires service-level tests
- Service class grows larger with each new operation type

Class Diagram and Sequence Diagram Reference: Analysis Model in 3.4.5.

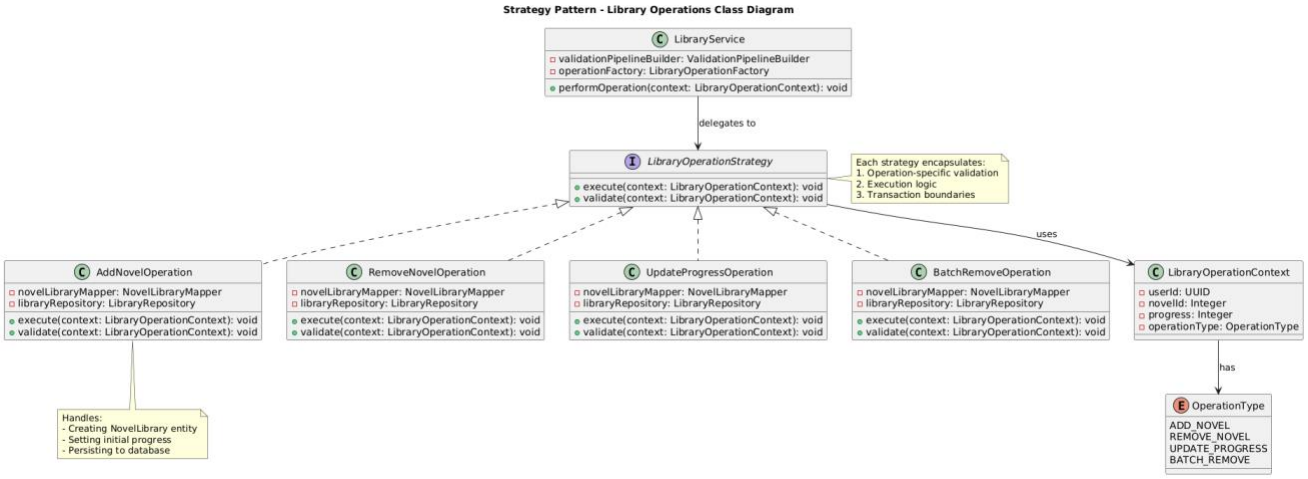
Candidate Design Patterns

- **Template Method** (base class with hooks for operation-specific steps)
- **Strategy** (separate object per operation type)

Motivation for Chosen Pattern

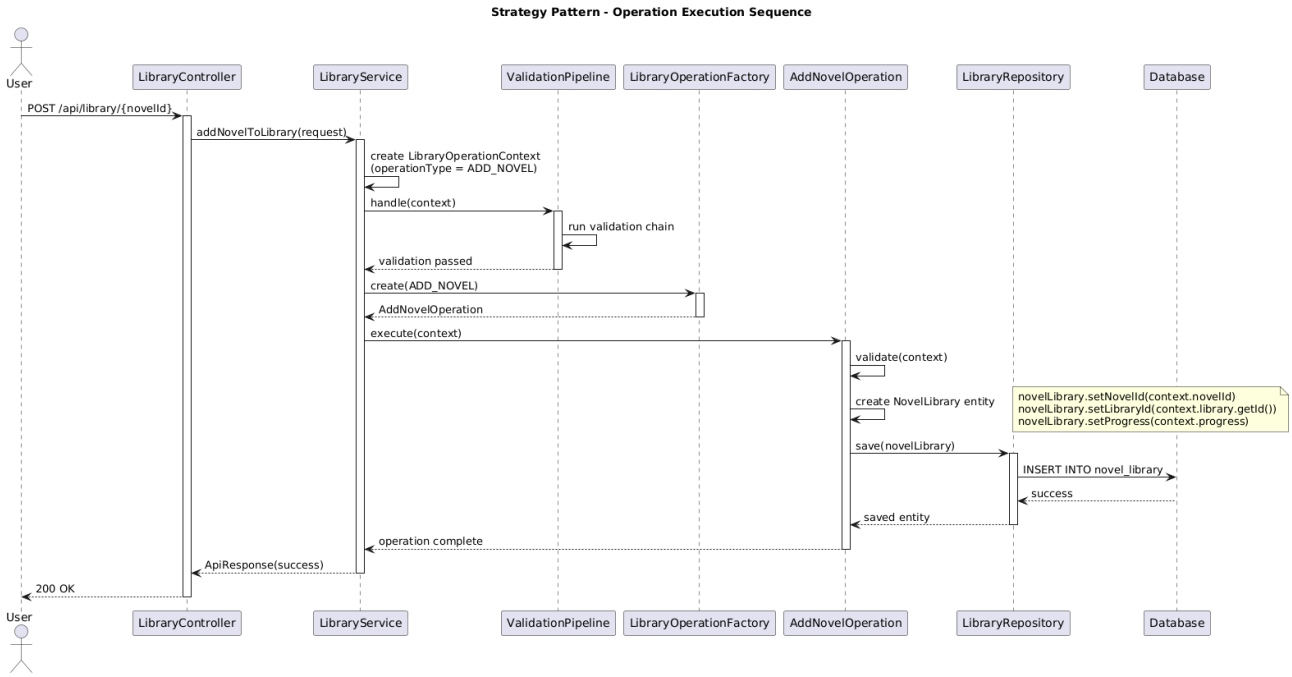
- **Strategy** keeps the service ignorant of concrete operation types while allowing each strategy to encapsulate its specific execution logic
- Swapping or adding strategies (e.g., BatchRemoveOperation, UpdateProgressOperation) requires no changes to the service core
- Template Method would still require inheritance hierarchy inside the service, less flexible for dependency injection
- Each strategy can have different dependencies injected independently

Class Diagram for the Design Solution



Define LibraryOperationStrategy interface with execute(context) and validate(context). Implement AddNovelOperation, RemoveNovelOperation, UpdateProgressOperation, and BatchRemoveOperation, each injecting required repositories. LibraryService uses LibraryOperationFactory to select the appropriate strategy based on operation type.

Sequence Diagram for the Design Solution



Service delegates operation execution to the chosen strategy after validation chain passes.

Implementation Decisions

- Register strategy beans in the Spring container and inject them into the factory
- Ensure strategies reuse shared validation pipeline through Chain of Responsibility
- Keep validation pipeline unchanged; strategies are invoked only after successful validation
- Each strategy manages its own transactional boundaries if needed

---

3. Centralizing Strategy Instantiation (Factory Method)

Design Problem Description

Context: LibraryService would need direct references to all concrete operation strategies, requiring conditional logic to select the appropriate strategy.

Pain points:

- Service constructors grow as more operation strategies are added
- Strategy selection logic is scattered; no central place to manage new operation registrations
- Harder to support variations (e.g., cached operations, async operations)
- Violates Open-Closed Principle when adding new operation types

Class Diagram and Sequence Diagram Reference: Analysis Model in 3.4.5.

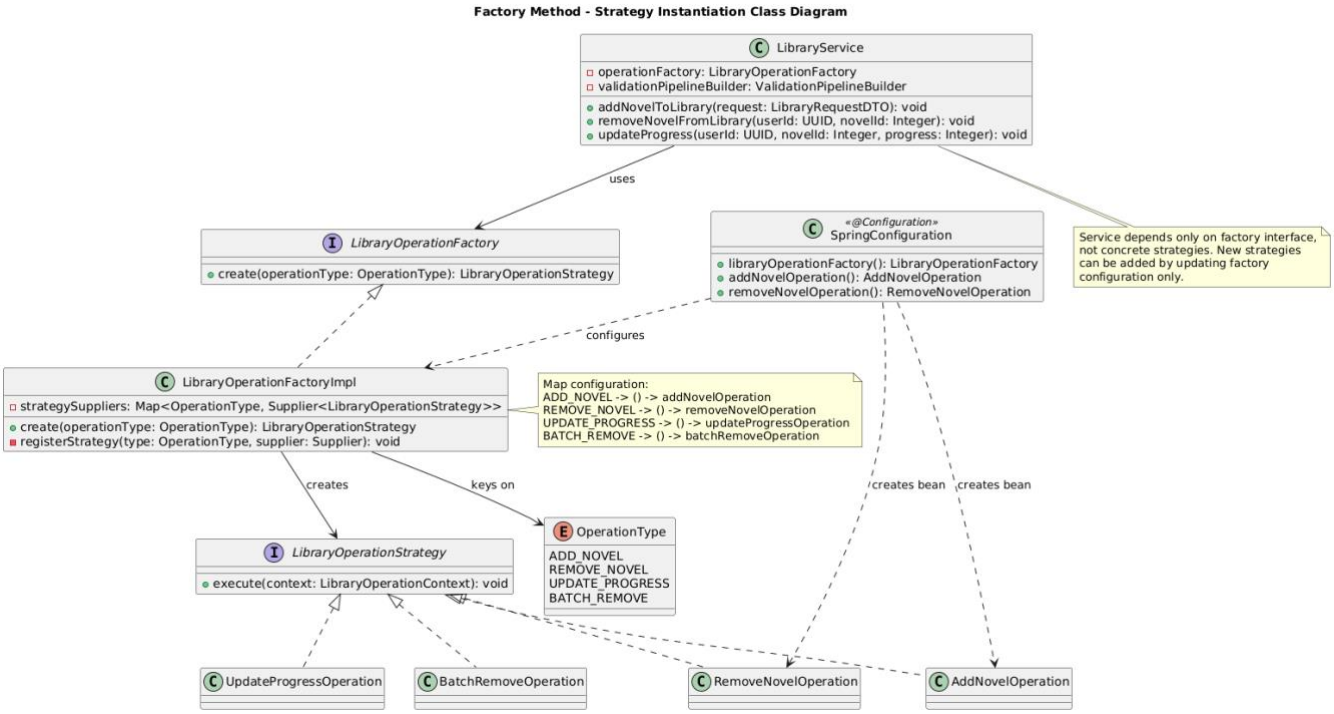
Candidate Design Patterns

- **Simple Factory** (static helper returning strategy instances)
- **Abstract Factory** (if multiple families of strategies were needed)
- **Factory Method** (inject suppliers and delegate creation through an interface)

Motivation for Chosen Pattern

- **Factory Method** allows the service to depend on an interface and makes it easy to inject new strategy suppliers without modifying service code
- Keeps alignment with dependency-injection style; each strategy can still be a Spring bean
- Abstract Factory would be overkill for single-family strategies
- Simple Factory would centralize logic but remain static and less testable

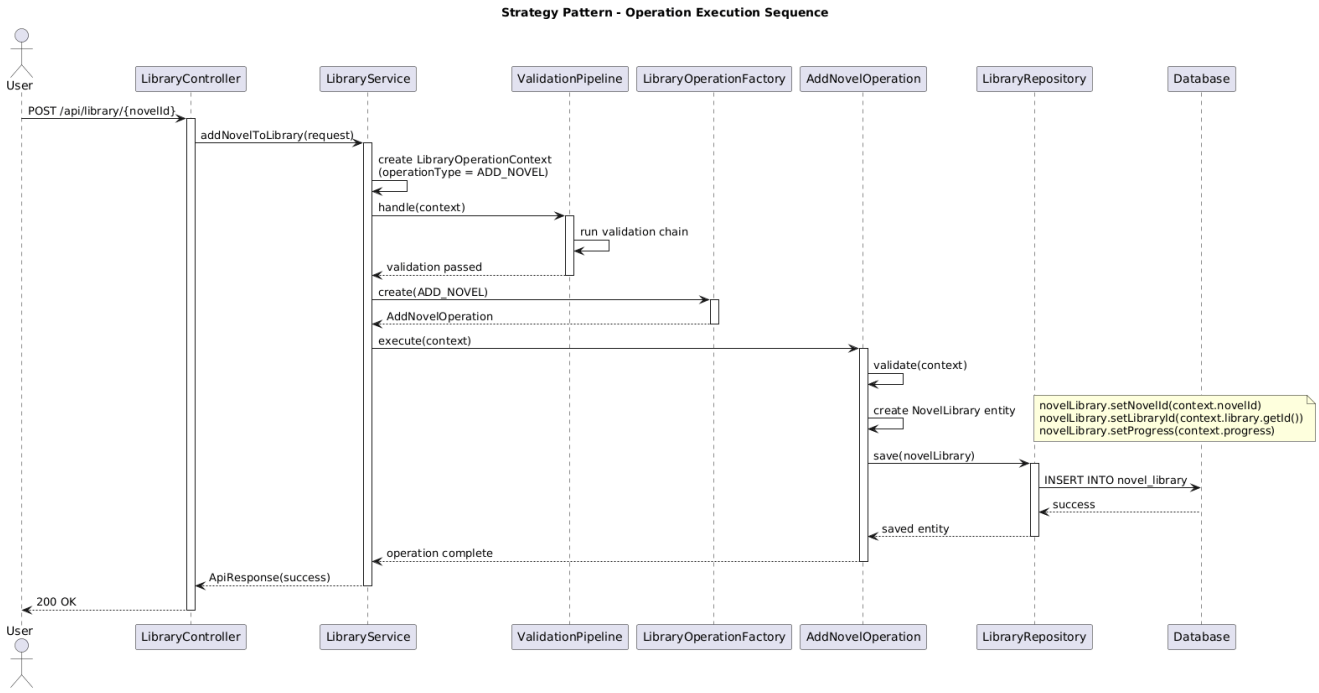
Class Diagram for the Design Solution



Introduce LibraryOperationFactory with create(operationType: OperationType) method; internally stores suppliers for each supported type (ADD_NOVEL, REMOVE_NOVEL, UPDATE_PROGRESS, BATCH_REMOVE). Refactor LibraryService to request strategies from the factory instead of owning concrete implementations. Maintain Chain of Responsibility for validation and Strategy pattern for operation execution.

Class Diagram Reference: Design Model in 3.4.5.

Sequence Diagram for the Design Solution

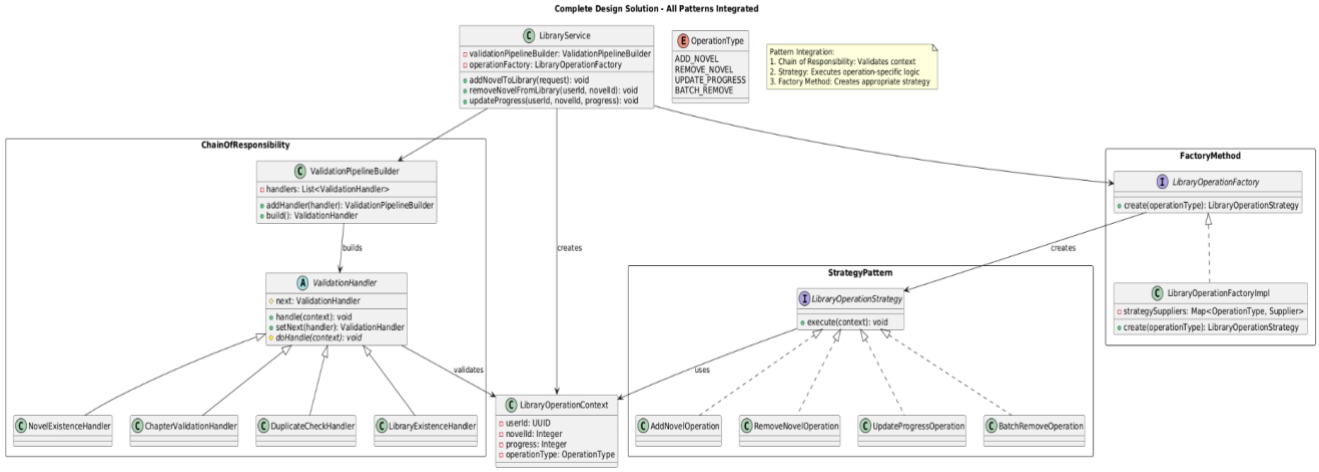


Controller → Service → ValidationPipelineBuilder → ValidationChain → Service → Factory → Strategy → Repository.

Sequence Diagram Reference: Design Model in 3.4.5.

Implementation Decisions

- Provide factory configuration in Spring using Map<OperationType, Supplier<LibraryOperationStrategy>> to map operation types to strategy bean suppliers
- Update existing unit tests to mock the factory instead of individual strategies
- Use @Transactional on LibraryService methods to ensure atomicity across validation and execution
- Factory selects strategy based on OperationType enum, making the system easily extensible



3.4.5.4 Complete Design Model

3.6 Database Schemas

3.6.1 Technology Stack & Design Principles

- Platform: PostgreSQL across development, staging, and production. UUID support is enabled via the pgcrypto extension (gen_random_uuid()).
- Schema management: Flyway provides automated versioned migrations. Scripts in classpath:db/migration are treated as the single source of truth and run automatically during application startup and CI/CD deployments. Settings (baseline-on-migrate=true, clean-disabled=true) guarantee safe rollouts and prevent accidental destructive operations.
- Normalization: The logical model adheres to Third Normal Form (3NF) to eliminate redundancy and make relationships explicit. Limited denormalisations (e.g., cached counters on novel) are maintained transactionally by service code to keep reads fast while preserving consistency.
- ORM / data mapper: MyBatis XML mappers under src/main/resources/mapper pair with entity POJOs in com.yushan.backend.entity. The shared configuration (config/mybatis-config.xml) enables automatic snake_case → camelCase conversion and registers a custom UUIDTypeHandler so PostgreSQL UUID columns map cleanly to java.util.UUID.
- Testing database: In-memory H2 (src/test/resources/schema.sql) mirrors the production schema for automated tests.

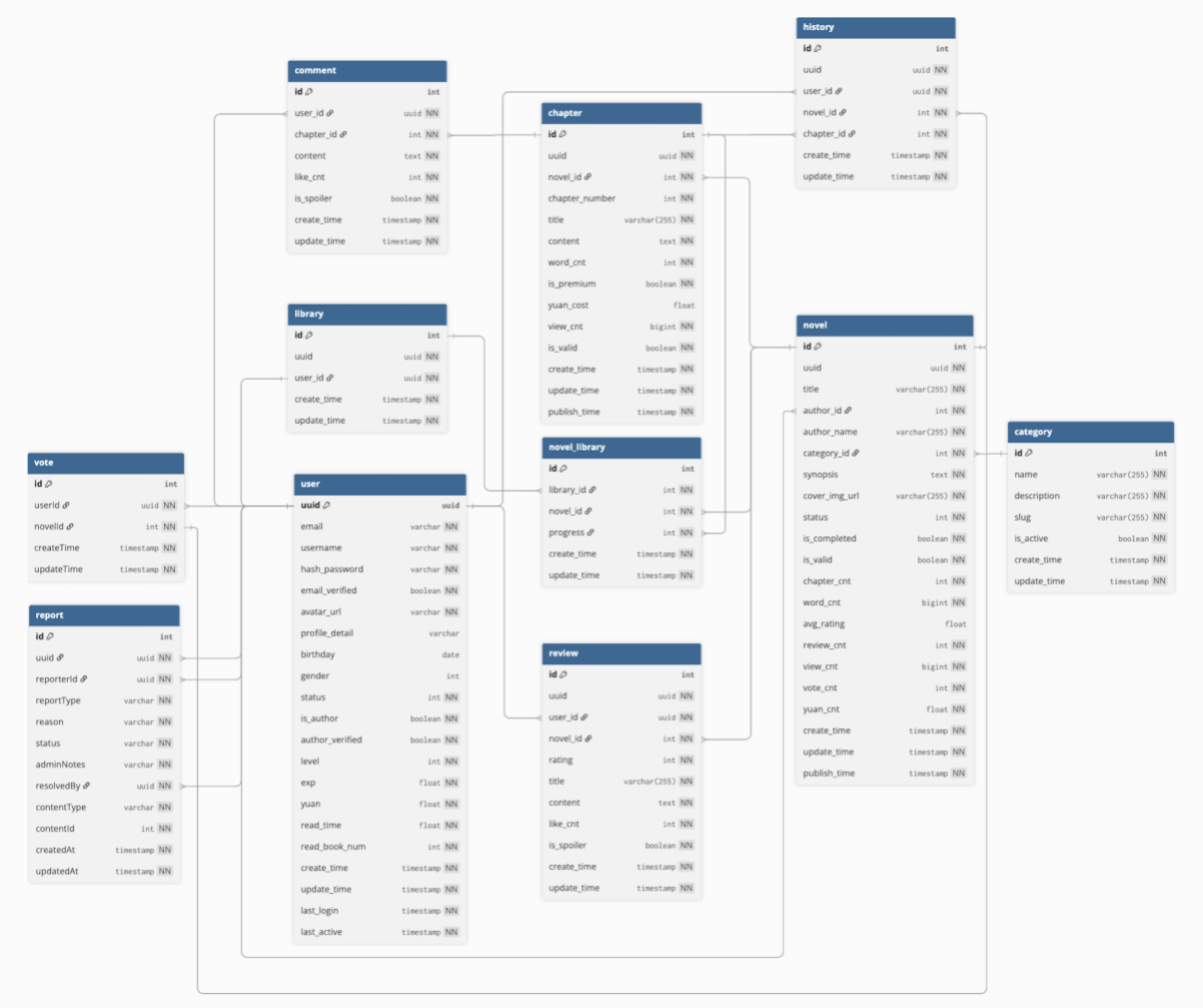
3.6.2 Migration Timeline (Flyway)

Version	File	Purpose
V1	V1__Initial_schema.sql	Creates the core tables (users, category, novel, chapter, comment, review, vote, library, novel_library).
V2	V2__Add_indexes.sql	Adds Postgres partial indexes optimised for novel browsing.
V3 & V6	V3__Seed_data.sql, V6__Enhanced_seed_data.sql	Populate base reference data (categories, novels, demo users) for dev/test.
V4	V4__Create_history.sql	Adds the history table for reader progress tracking.
V5	V5__Create_report_system.sql	Creates report_type & report_status enums plus the report table, rebuilds vote.
V7	V7__Update_novel_status_and_user_library.sql	Data quality adjustments: novel statuses, synthetic user libraries.
V8–V10	V8__Update_novel_images.sql	Content tuning migrations

	V9__Fix_all_data_inconsistencies.sql V10__Fix_chapter_numbers_and_titles.sql	(cover images, chapter titles, numbering).
V11	V11__Alter_novel_library_progress_nullable.sql	Allows novel_library.progress to be NULL (meaning "not started").

- Flyway runs with baseline-on-migrate=true, so new environments start at the current schema state even if initial tables already exist.

3.6.3 Schema Overview



- The schema centres around the novel domain: users author and consume novels broken into chapters; reviews, votes, comments, libraries, and reports hang off those core objects. UUIDs are used for public identifiers while integer SERIAL keys support efficient joins.
- Below, each table is documented with its columns, constraints, relationships, and notable usage in the code base.

1. Users

- Purpose: Accounts for readers, authors, and admins.

- Key columns:
- uuid UUID (PK) – generated via gen_random_uuid().
    - Authentication & profile fields: email (unique), hash_password, avatar_url, profile_detail, gender, birthday.
    - Engagement metrics: level, exp, yuan, read_time, read_book_num.
    - Flags: email_verified, is_author, is_admin, status.
    - Audit timestamps: create_time, update_time, last_login, last_active.
- Indexes/constraints: email unique constraint; several default values to avoid NULL handling.
- Code references: mapped by User.java and UserMapper.xml; services like UserService fetch author names and reader metadata.

2. Category

- Purpose: Taxonomy for novels.
- Columns: id SERIAL (PK), name, description, slug (unique), is_active, timestamps.
- Usage: category_id foreign key in novel; CategoryMapper handles CRUD.

3. Novel

- Purpose: Core published content.
- Columns: id SERIAL (PK), uuid, title, author_id (FK → users.uuid), category_id (FK → category.id), synopsis, cover_img_url, lifecycle/status fields (status, is_completed, is_valid, publish_time), analytics counters (chapter_cnt, word_cnt, avg_rating, review_cnt, view_cnt, vote_cnt, yuan_cnt), audit timestamps.
- Indexes (V2__Add_indexes.sql):
    - idx_novel_valid_category_status (partial on (is_valid, category_id, status)) powering list filters.
    - idx_novel_create_time_desc (partial on create_time DESC for recency sorting).
    - idx_novel_valid_author (partial on (is_valid, author_id) for dashboards).
- Code references: NovelMapper.xml, NovelService for queries and updates, ReviewService updates avg_rating & review_cnt.

4. Chapter

- Purpose: Chapter-level content under a novel.
- Columns: id SERIAL (PK), uuid, novel_id (FK → novel.id with ON DELETE CASCADE), chapter_number, title, content, word_cnt, monetisation flags (is_premium, yuan_cost), analytics (view_cnt), lifecycle flags (is_valid, publish_time), timestamps.
- Constraints: (novel_id, chapter_number) unique to prevent duplicates. V10__Fix_chapter_numbers_and_titles.sql renumbers chapters sequentially, updates titles, and recalculates chapter_cnt/word_cnt in novel.
- Code references: ChapterMapper.xml, ChapterService for CRUD and sequencing logic.

5. Comment

- Purpose: Reader comments on specific chapters.
- Columns: id SERIAL (PK), user_id (FK → users), chapter_id (FK → chapter, cascades on delete), content, like_cnt, is_spoiler, timestamps.

- Usage: CommentService handles creation, moderation (ReportService references comments when resolving reports).

6. Review

- Purpose: Star ratings and critiques at the novel level.
- Columns: id SERIAL (PK), uuid, user_id (FK → users), novel_id (FK → novel), rating (1–5 enforced by CHECK), optional title & content, like_cnt, is_spoiler, timestamps.
- Constraints: unique_user_novel_review ensures one review per user/novel pair.
- Code references: ReviewService orchestrates creation, ensuring duplicates are blocked and novel metrics update; ReviewMapper.xml provides queries.

7. Vote

- Purpose: Reader upvotes for novels (daily vote mechanic).
- Columns: id SERIAL (PK), user_id, novel_id, timestamps.
- History: Originally included an is_active soft-delete flag; V5__Create_report_system.sql recreated the table without it to simplify toggle logic.
- Constraints: (re-)enforced programmatically in VoteService to prevent duplicates; earlier unique index dropped when table recreated (service now toggles by deletion/reinsert).

8. Library

- Purpose: Personal library container per user.
- Columns: id SERIAL (PK), uuid (manual assignment in services), user_id (owner), timestamps.
- Usage: LibraryMapper.xml attaches libraries to users; V7 ensures every user has one.

9. Novel_Library

- Purpose: Join table mapping libraries to novels with reading progress.
- Columns: id SERIAL (PK), library_id (FK conceptually pointing to library.id), novel_id, progress (0–100 percentage, now nullable after V11), timestamps.
- Data shaping: V7 seeds realistic progress patterns; progress=NULL now signals "not started".

10. History

- Purpose: Track the most recently read chapter per user/novel pair.
- Columns: id SERIAL (PK), uuid, user_id, novel_id, chapter_id, timestamps.
- Constraints: No explicit foreign keys defined in V4, but entity services (HistoryMapper.xml) treat these IDs as references to the respective tables.

11. Report

- Purpose: Moderation reports for novels and comments.
- Structure:
    - Enum types report_type (PORNOGRAPHIC, HATE_BULLYING, PERSONAL_INFO, INAPPROPRIATE, SPAM) and report_status (IN_REVIEW, RESOLVED, DISMISSED) created in V5.

- Columns: id SERIAL (PK), uuid, reporter_id (FK → users), report_type, reason, status, admin_notes, resolved_by (FK → users), content_type (NOVEL/COMMENT), content_id, timestamps.

- Business rules: ReportService enforces uniqueness per reporter/content, captures moderation actions (e.g., reverting novel to DRAFT, soft-deleting comments) and requires admin_notes when resolving.

12. Other supporting structures

- Enums: report_type, report_status (Postgres native enums).
- Sequences/serials: Implicit for tables using SERIAL (Postgres sequences).
- Audit columns: Most tables follow the create_time / update_time convention, updated via SQL defaults and service-layer updates.

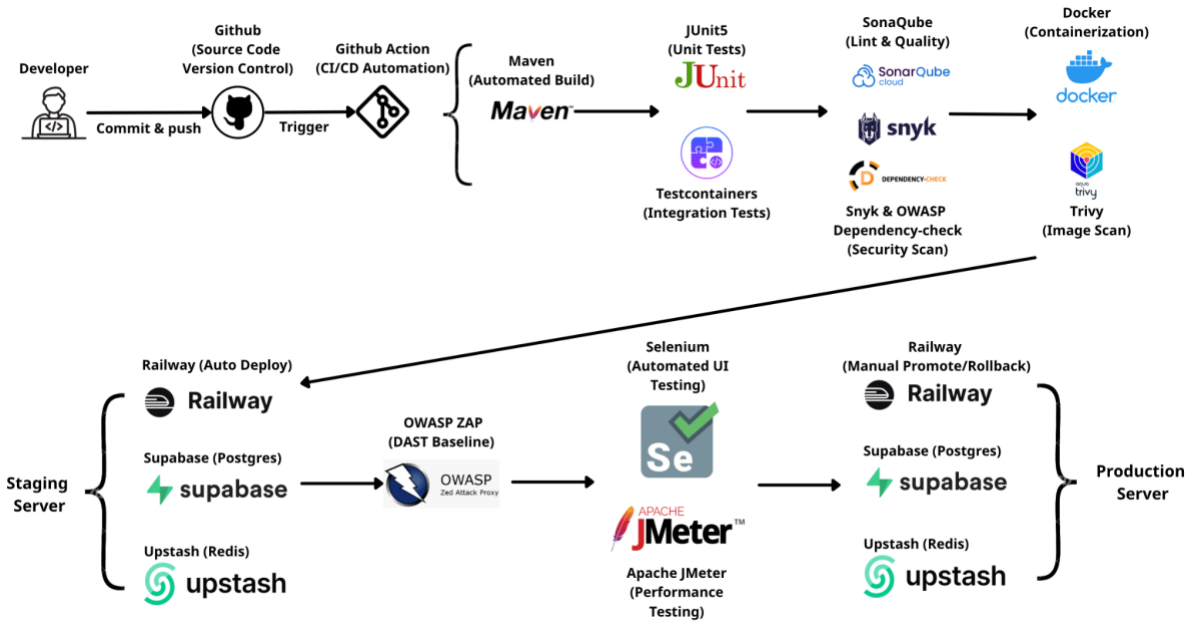
3.6.4 Core Relationships

- user → novel: authorship (novel.author_id FK), plus user-generated content via review, comment, vote, history, library.
- category → novel: novels inherit taxonomy via category_id.
- novel → chapter, review, vote, history, novel_library: novels own their chapters and aggregate engagement counters.
- chapter → comment, history: comments are scoped to chapters; history tracks reading progress at chapter granularity.
- library ↔ novel_library ↔ novel: per-user shelves reference novels with progress metadata; novel_library.progress (nullable) records completion percentage.
- report: polymorphic references to either novel or comment via (content_type, content_id) alongside reporter/assignee UUIDs.

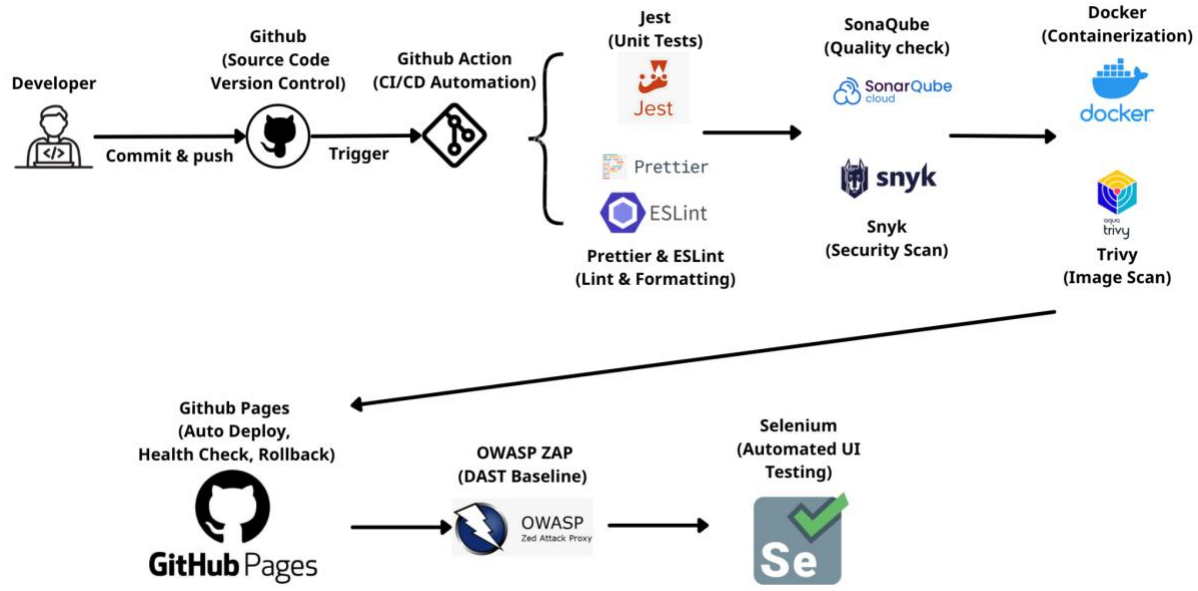
3.6.5 Data Access Layer Notes

- MyBatis mappers provide CRUD plus custom queries (e.g., ReportMapper.existsReportByUserAndContent, NovelMapper.selectByPrimaryKey). The configuration maps snake_case columns to camelCase Java properties automatically.
- DTOs in com.yushan.backend.dto (e.g., NovelDetailResponseDTO, ReportResponseDTO) aggregate multiple mapper calls; documentation in this file can be cross-referenced when understanding service orchestrations (NovelService, ReportService, LibraryService).
- The UUIDTypeHandler keeps PostgreSQL UUID columns from being coerced to strings when mapping to Java.

4. DevSecOps and Development Lifecycle



DevOps Pipeline Design for Backend



DevOps Pipeline Design for Frontend

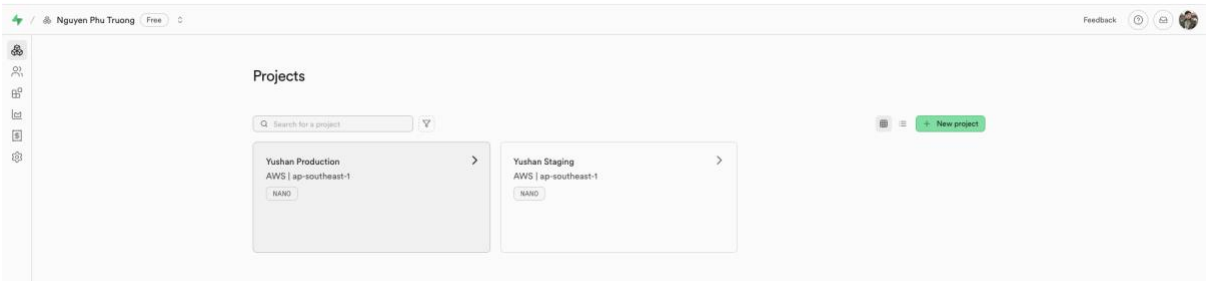
4.1 Source Control Strategy (Backend)

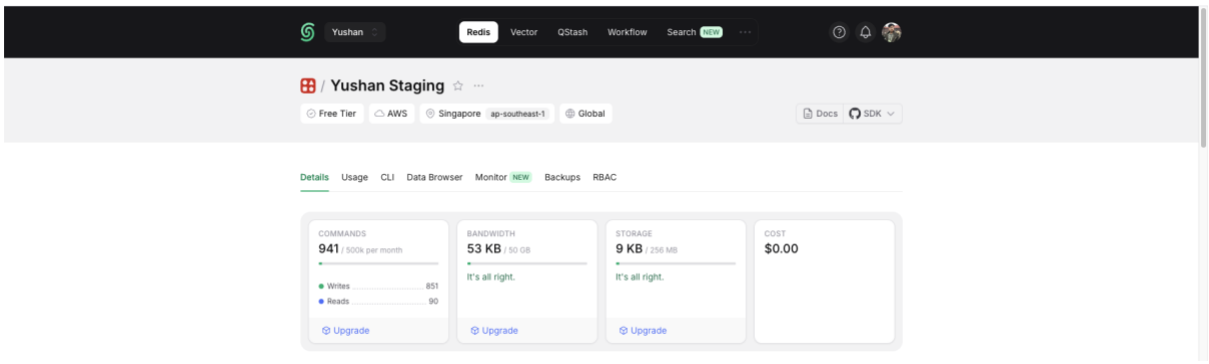
- Repository layout: the backend lives in a mono-repo (yushan-backend) managed on GitHub.
- Branching model: all production code sits on main. Developers create short-lived feature branches and raise Pull Requests into main.
- Branch protection:
    - Direct pushes to main are blocked in Github settings; PR review is mandatory.

- GitHub Actions status checks (unit, integration, lint, security, build) must pass before a PR can merge.
- GitHub Environments (yushan-staging, yushan-production) enforce additional approval for deployments (prevent_self_review is enabled, reviewer lists can be configured in the environment yaml).

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

- Secrets & access:
  - Build and deployment credentials (Railway token, Sonar token, dependency scan tokens) and infrastructure endpoints (Supabase Postgres URL/credentials, Upstash Redis URL) are stored as GitHub secrets or environment secrets (.github/environments/*.yml documents what is required).
  - Supabase provides the managed PostgreSQL for both staging and production; Upstash supplies Redis instances. Access keys are injected via environment secrets during deployment.



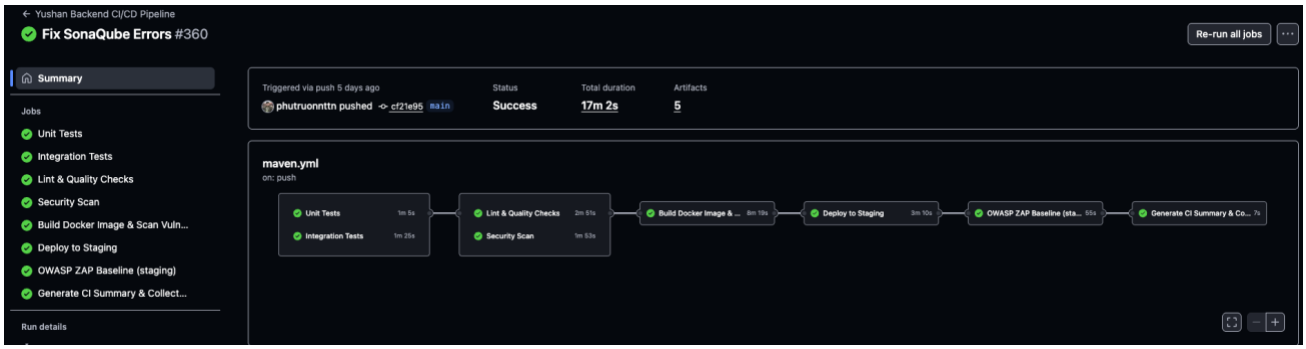


  - Action jobs use scoped GITHUB_TOKEN permissions (contents read, packages write, etc.). Repository access follows GitHub organisation policies (2FA required).
- Artefact strategy:
  - Docker images are pushed to GitHub Container Registry (ghcr.io/<repo>). Production tags are managed by promotion workflows (see CD below).
  - Reports (SpotBugs SARIF, dependency-check, ZAP, etc.) are uploaded as workflow artifacts for later inspection.

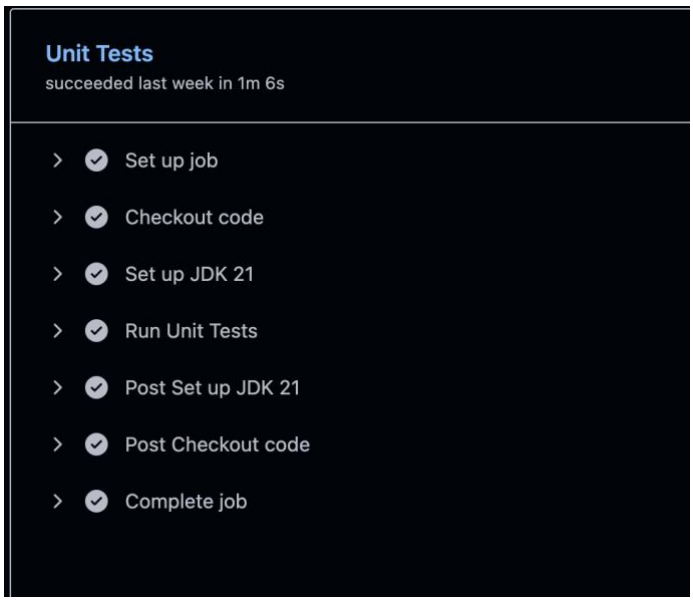
Name	Size	Digest
ci-reports	212 KB	sha256:73fdb57b2ff05bcc83e00d22f873afbc577808c48dbd0c6336945296…
image-digest	208 Bytes	sha256:954f62a24ebf2e1033f3318d683a5302b9879a1e0cdcaf3ebcad54bb…
owasp-dependency-check-report	192 KB	sha256:ce4d5aad2664bf491eee02b94a027f249152f87c11ef89590037289b…
trivy-sarif	1.5 KB	sha256:86298c6b5f574e2ba3e97528dd982206d56220badf3df00fea31c3bd…
zap-staging	18.6 KB	sha256:4beb000c0f2eb1fa2b0f7eb8edfe058e261731b3513994945790 83e9…

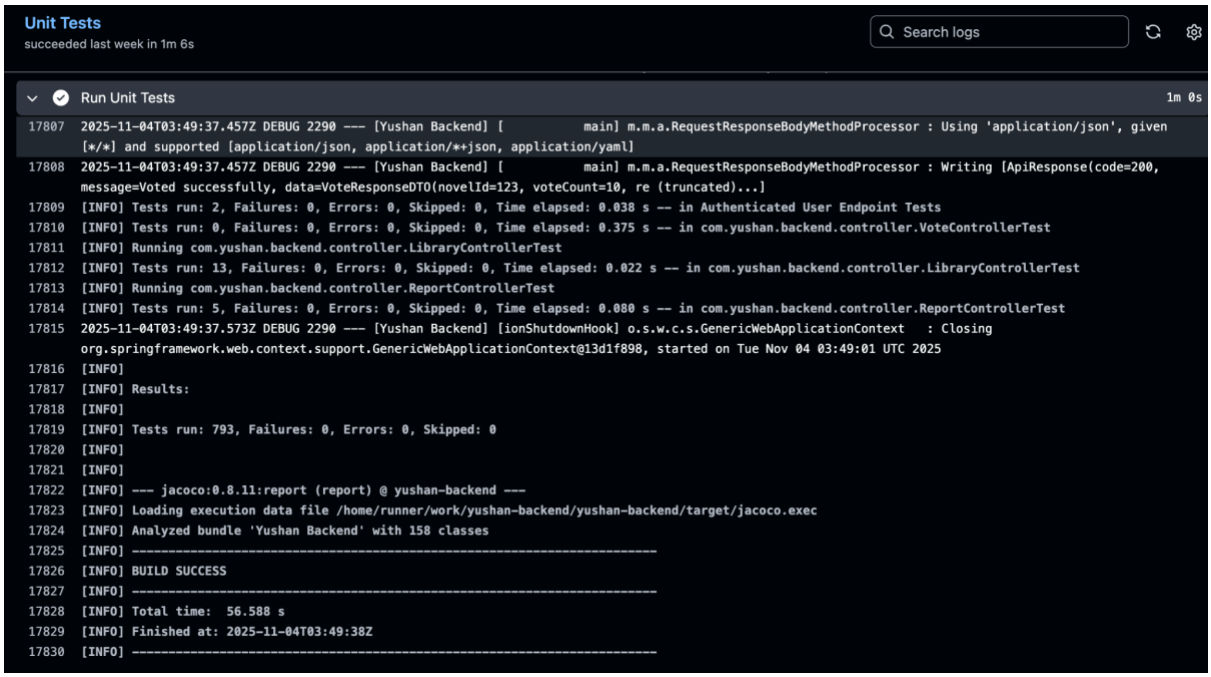
4.2 Continuous Integration (Backend)

- All CI jobs are defined in .github/workflows/maven.yml, triggered on pushes and PR updates to main.



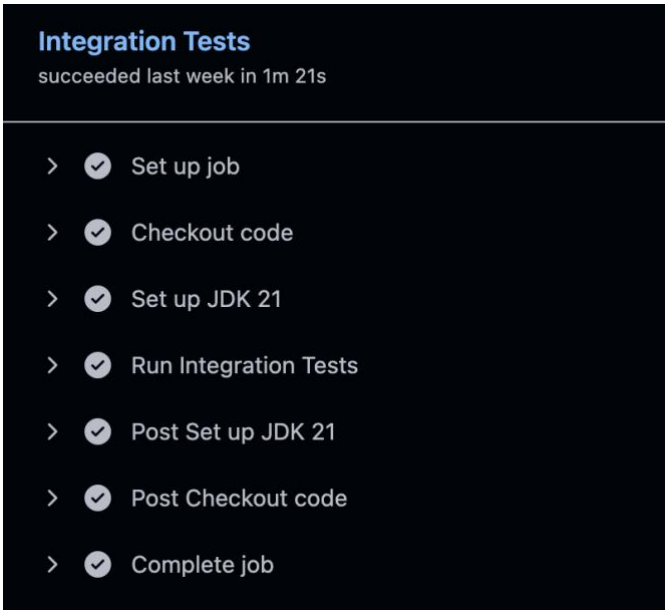
- Unit Tests (unit-tests job)
  - Runs on ubuntu-latest.
  - Checks out code (actions/checkout@v4), sets up Temurin JDK 21 (actions/setup-java@v4), executes ./mvnw test excluding the integration package with Spring profile test.

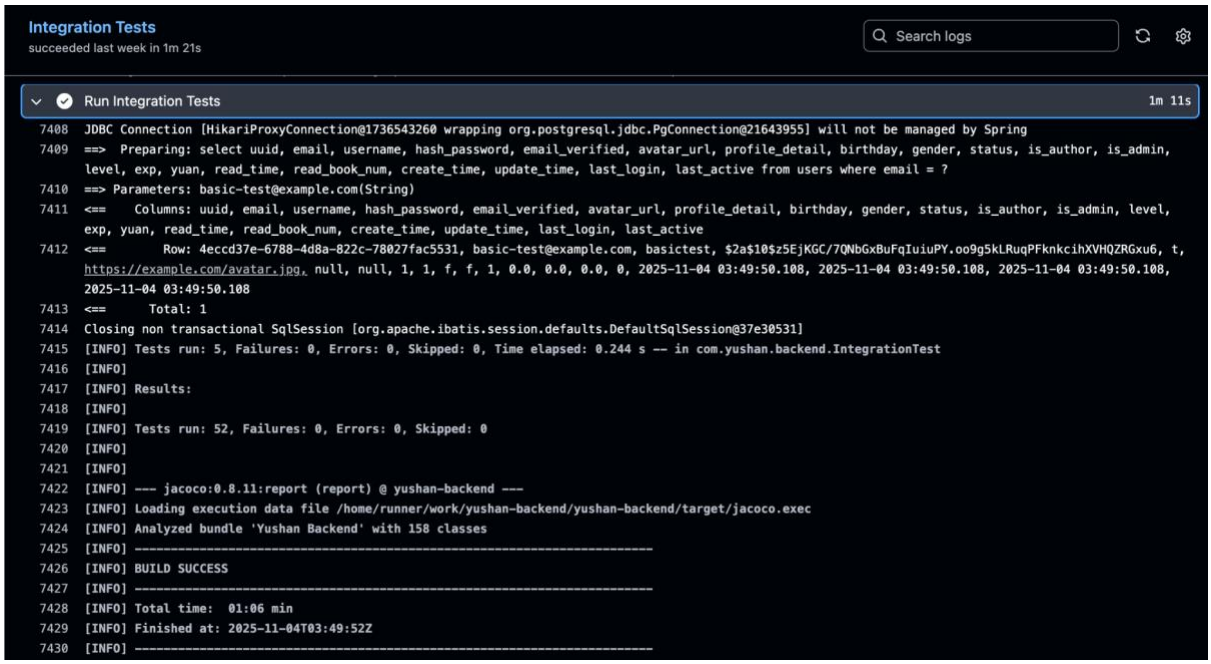




Unit Test Execution Output (GitHub Actions)

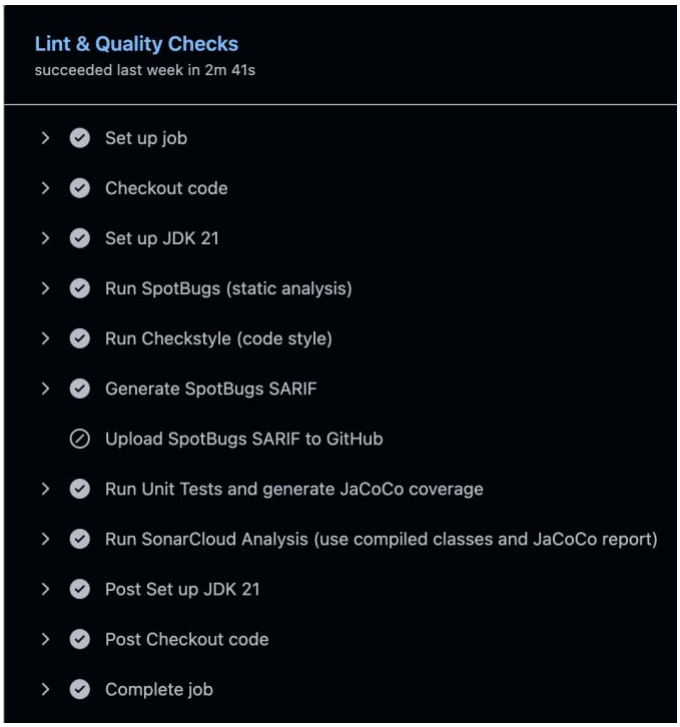
- Integration Tests (integration-tests job)
  - Same setup; runs only integration-scoped tests with Spring profile integration-test.
  - Exports CI=true for tooling compatibility.





Integration Test Execution Output (GitHub Actions)

- Lint & Quality (lint-quality job, needs previous two)
  - Runs SpotBugs (./mvnw spotbugs:check), Checkstyle (./mvnw checkstyle:check).
  - Generates SpotBugs SARIF and uploads via github/codeql-action/upload-sarif.
  - Re-runs unit tests with JaCoCo coverage (./mvnw test jacoco:report).
  - Performs SonarCloud analysis using the generated binaries and coverage (SONAR_TOKEN secret).



Yushan Backend

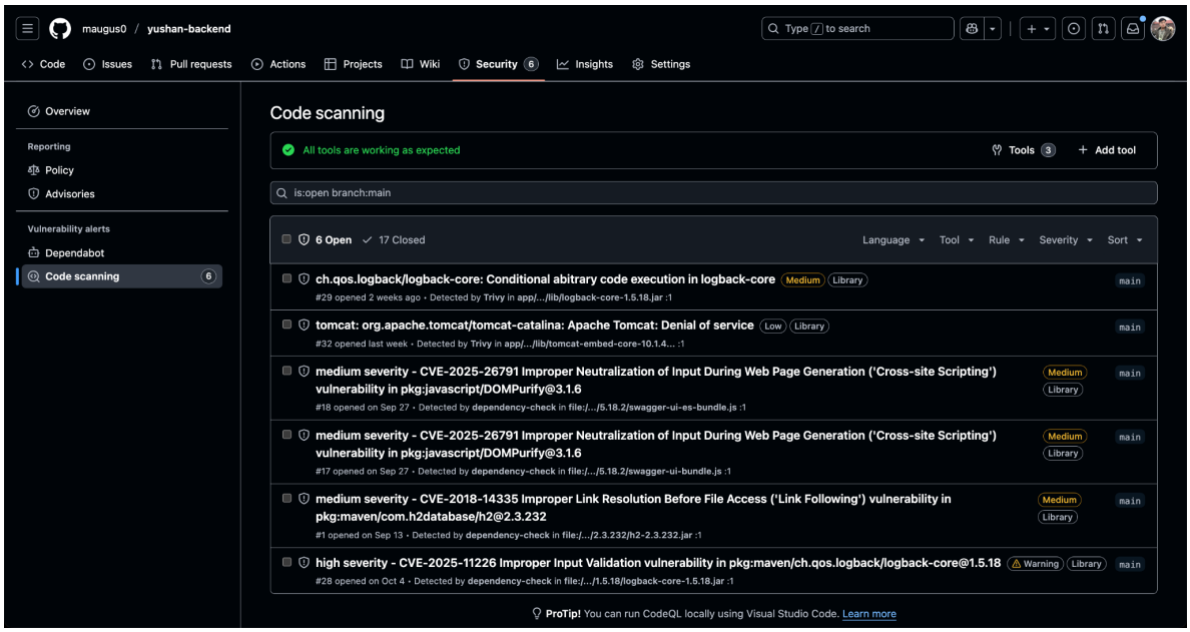
Yushan Backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.yushan.backend.dto		83%		50%	1,550	3,329	39	1,330	67	1,501	0	80
com.yushan.backend.service		78%		54%	404	760	417	2,186	48	236	0	19
com.yushan.backend.controller		79%		54%	101	271	130	692	19	168	0	17
com.yushan.backend.entity		83%		56%	85	357	57	508	6	254	0	11
com.yushan.backend.security		85%		62%	68	166	39	239	13	67	0	7
com.yushan.backend.enums		87%		58%	15	45	16	98	3	27	0	6
com.yushan.backend.util		87%		71%	13	49	11	102	9	42	0	3
com.yushan.backend.exception		67%		50%	5	21	15	52	4	19	0	5
com.yushan.backend.config		97%		83%	6	35	6	106	3	26	3	8
com.yushan.backend.interceptor		95%		87%	2	14	2	45	0	6	0	1
com.yushan.backend		37%		n/a	1	2	2	3	1	2	0	1
Total	6,979 of 39,027	82%	2,574 of 5,382	52%	2,250	5,049	734	5,361	173	2,348	3	158

Coverage report (JaCoCo)

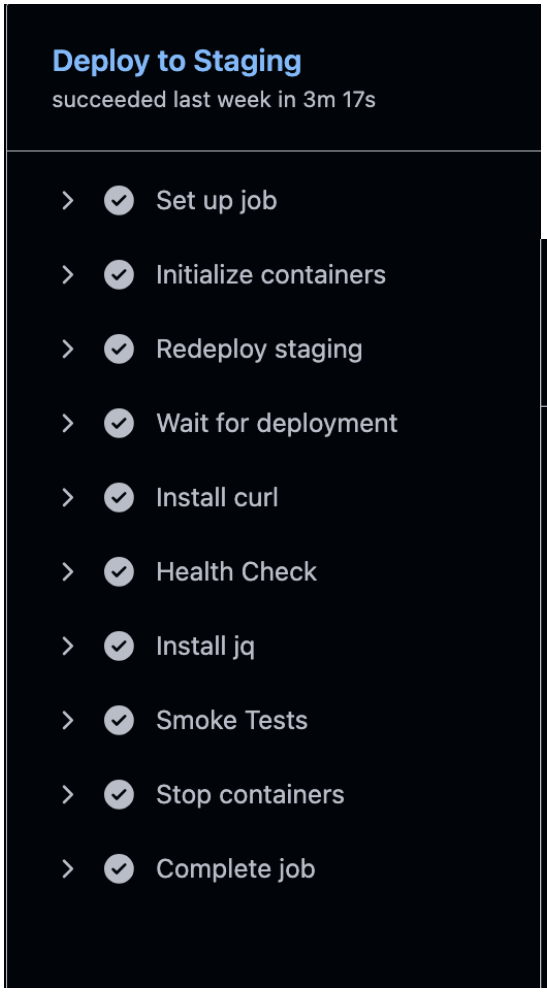
SonarQube Report

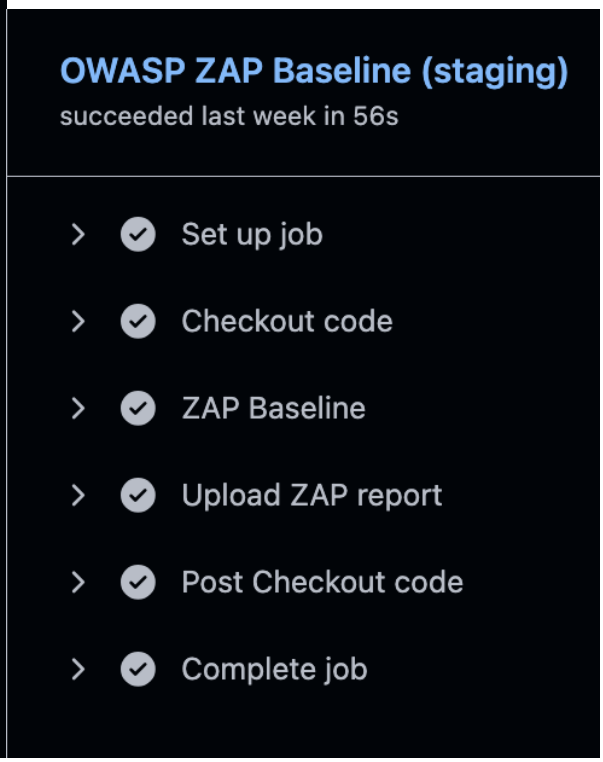
- Security Scan (security-scan job, parallel to lint-quality)
  - OWASP Dependency-Check (with cached data and optional OSS Index credentials).
  - Builds the project (skip tests).
  - Snyk CLI scan exporting SARIF; uploads to GitHub code scanning.
  - Caches dependency-check data and Maven repo to speed up runs.
- Docker Build & Trivy (docker-build job)
  - Depends on all analysis jobs.
  - Sets up Docker Buildx, logs into GHCR, caches Maven repo for build stage.
  - Uses docker/build-push-action@v5 to build multi-arch image (linux/amd64 and linux/arm64) and push multiple tags (branch, pr, sha, latest, staging, ci-cd-testing).
  - Runs Trivy vulnerability scan on the pushed image, uploads SARIF to GitHub code scanning and as artifact.
  - Saves image digest (image-digest.txt) for promotion workflow.

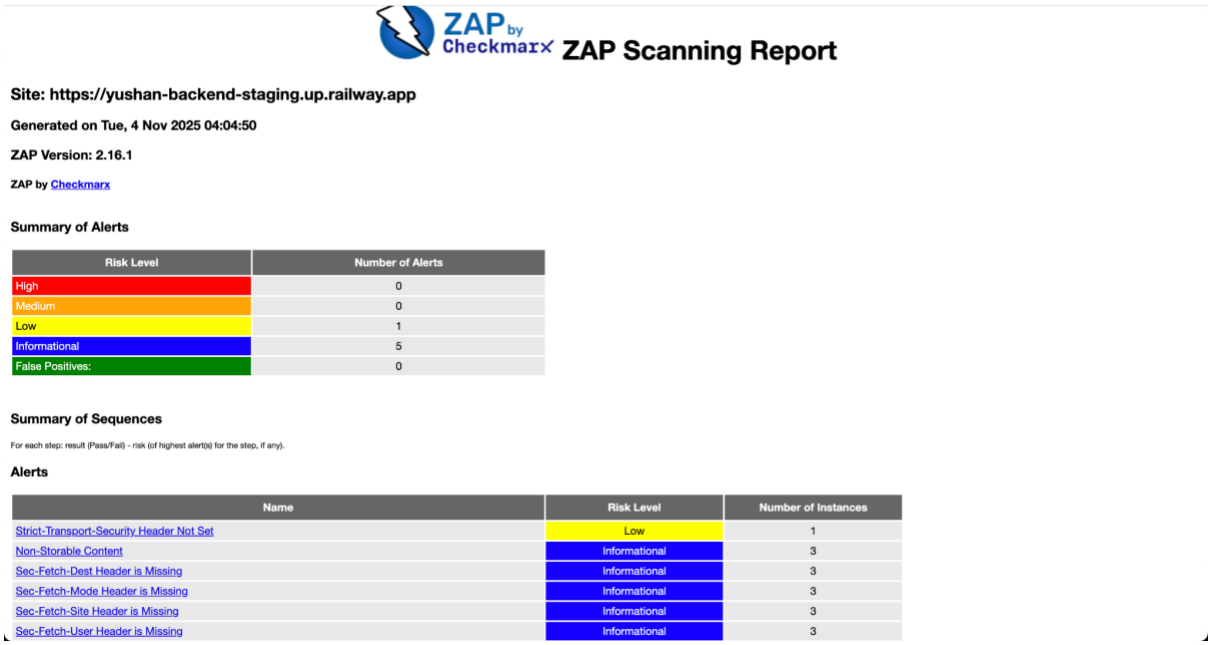


Code scanning in Github Security

- Staging Deploy (staging-deploy job)
  - Triggered only on push to main.
  - Runs inside the Railway CLI container.
  - Redeploys staging service (railway redeploy).
  - Waits 180 seconds, installs curl and jq, runs health check on /actuator/health, and smoke tests on /api/example/public, /actuator/info, /actuator/metrics.
- DAST – OWASP ZAP (dast-zap job)
  - Runs after successful staging deploy; uses zaproxy/action-baseline against staging URL.
  - Uploads HTML/JSON/Markdown reports as artifacts.







ZAP Scanning Report

Site: https://yushan-backend-staging.up.railway.app

Generated on Tue, 4 Nov 2025 04:04:50

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	1
Informational	5
False Positives:	0

Summary of Sequences

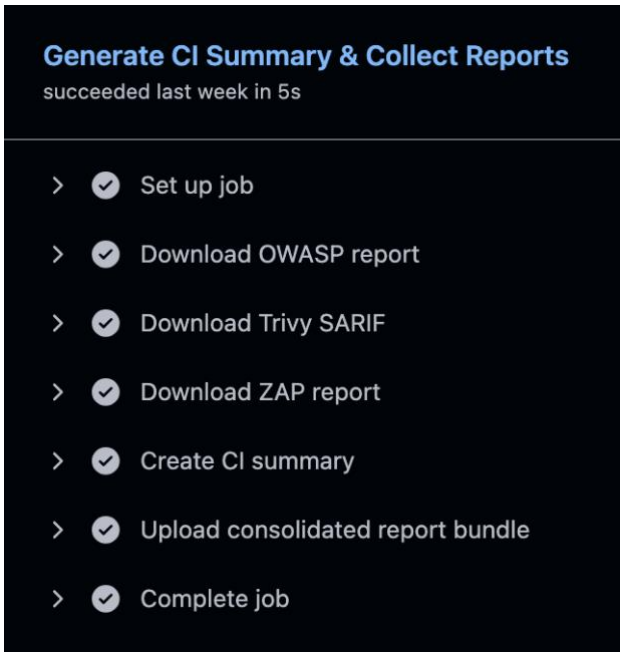
For each step: result (Pass/Fail) - risk (of highest alert(s)) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Strict-Transport-Security Header Not Set	Low	1
Non-Storable Content	Informational	3
Sec-Fetch-Dest Header is Missing	Informational	3
Sec-Fetch-Mode Header is Missing	Informational	3
Sec-Fetch-Site Header is Missing	Informational	3
Sec-Fetch-User Header is Missing	Informational	3

ZAP Scanning Report (AS-WAS)

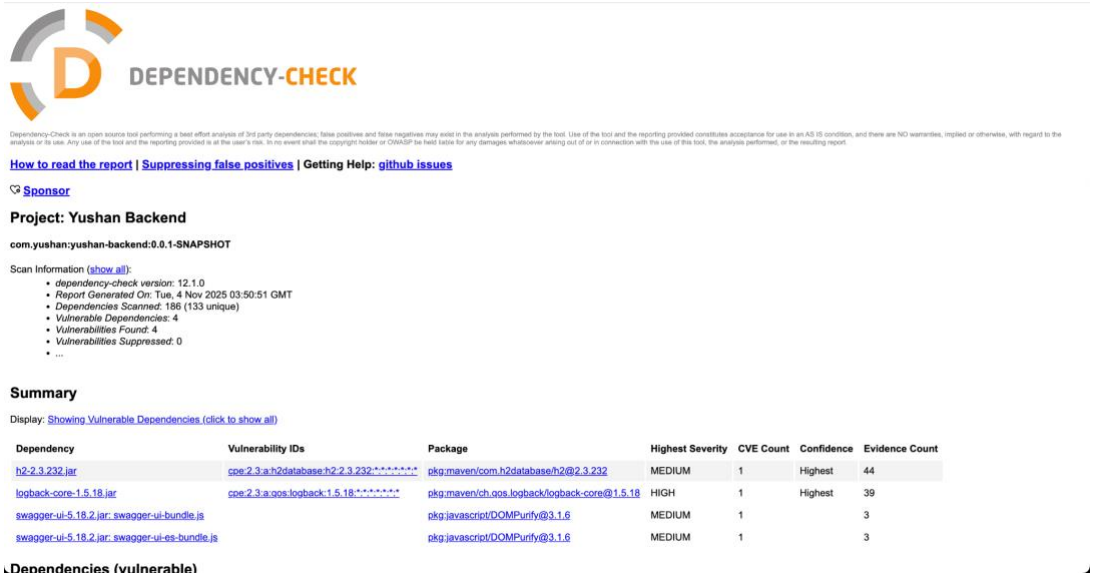
- Generate Reports (generate-reports job)
  - Downloads dependency-check, Trivy, and ZAP artifacts.
  - Produces a Markdown summary (commit, branch, event, available artifacts).
  - Uploads consolidated report bundle (ci-reports).



- Additional workflow features:
  - Concurrency guard: prevents overlapping runs per branch.
  - Job permissions: each job only gets the minimal tokens required (security-events: write only where SARIF uploads happen).
  - Caching: Maven repo, dependency-check data, Trivy results, etc. to reduce build time.

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

- Below are the issues identified in the most recent SCA, DAST, and SAST scans from the CI/CD pipeline. Many issues from earlier CI/CD runs had already been fixed, but screenshots of those results were not retained.



DEPENDENCY-CHECK

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | Suppressing false positives | Getting Help: github issues

Sponsor

Project: Yushan Backend

com.yushan:yushan-backend:0.0.1-SNAPSHOT

Scan Information (show all):

- *dependency-check version*: 12.1.0
- *Report Generated On*: Tue, 4 Nov 2025 03:50:51 GMT
- *Dependencies Scanned*: 186 (133 unique)
- *Vulnerable Dependencies*: 4
- *Vulnerabilities Found*: 4
- *Vulnerabilities Suppressed*: 0
- ...

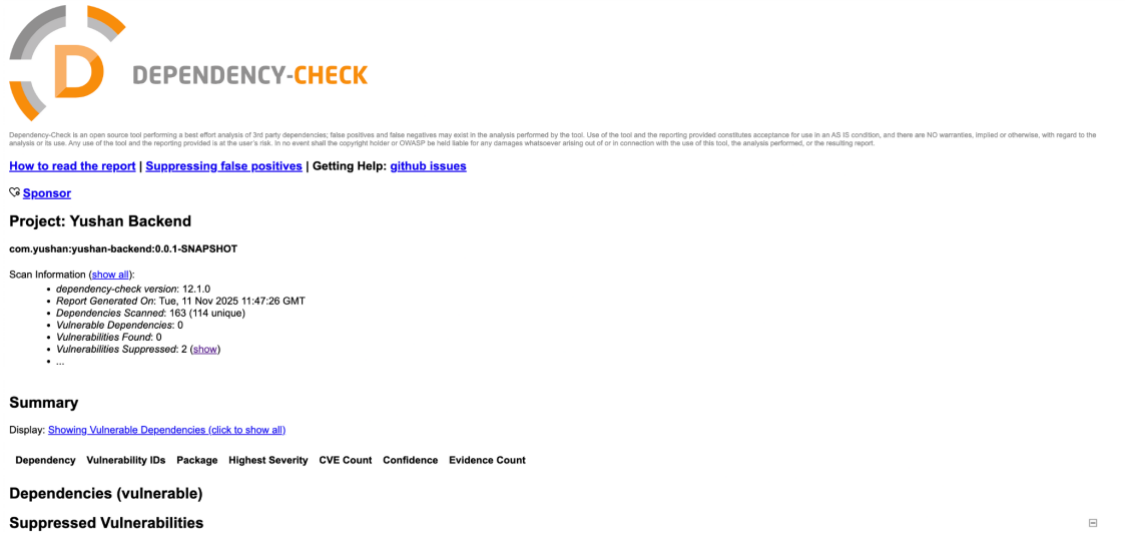
Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
h2-2.3.232.jar	cpe:2.3:a:h2database:h2:2.3.232:*:*:*:*:*:*:*	pkg:maven/com.h2database/h2@2.3.232	MEDIUM	1	Highest	44
logback-core-1.5.18.jar	cpe:2.3:a:qos:logback:1.5.18:*:*:*:*:*:*:*	pkg:maven/ch.qos.logback/logback-core@1.5.18	HIGH	1	Highest	39
swagger-ui-5.18.2.jar: swagger-ui-bundle.js		pkg:javascript/DOMPurify@3.1.6	MEDIUM	1		3
swagger-ui-5.18.2.jar: swagger-ui-es-bundle.js		pkg:javascript/DOMPurify@3.1.6	MEDIUM	1		3

Dependencies (vulnerable)

OWASP Dependency-Check Report (AS-WAS)



DEPENDENCY-CHECK

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | Suppressing false positives | Getting Help: github issues

Sponsor

Project: Yushan Backend

com.yushan:yushan-backend:0.0.1-SNAPSHOT

Scan Information (show all):

- *dependency-check version*: 12.1.0
- *Report Generated On*: Tue, 11 Nov 2025 11:47:26 GMT
- *Dependencies Scanned*: 163 (114 unique)
- *Vulnerable Dependencies*: 0
- *Vulnerabilities Found*: 0
- *Vulnerabilities Suppressed*: 2 (show)
- ...

Summary

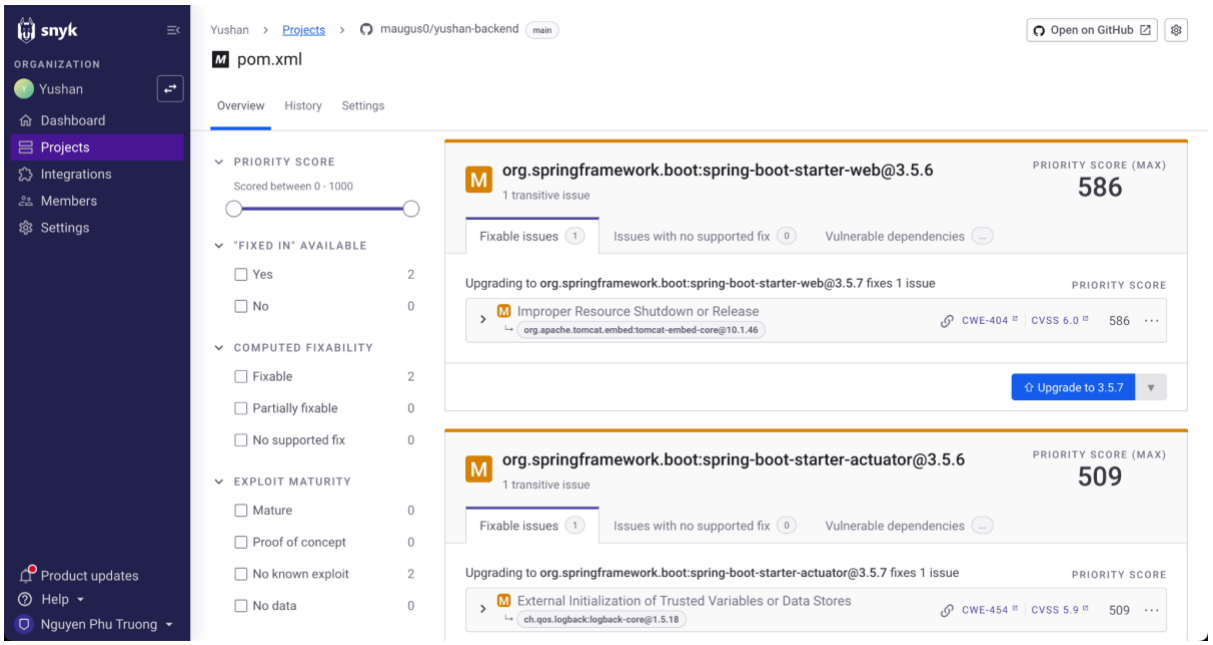
Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count

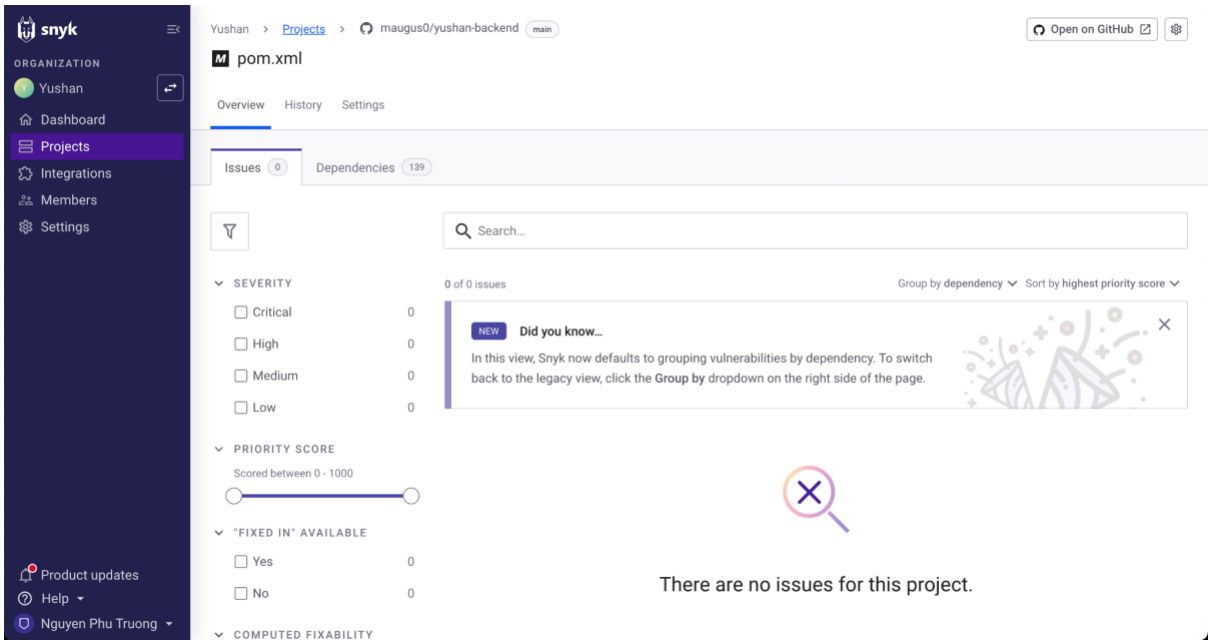
Dependencies (vulnerable)

Suppressed Vulnerabilities

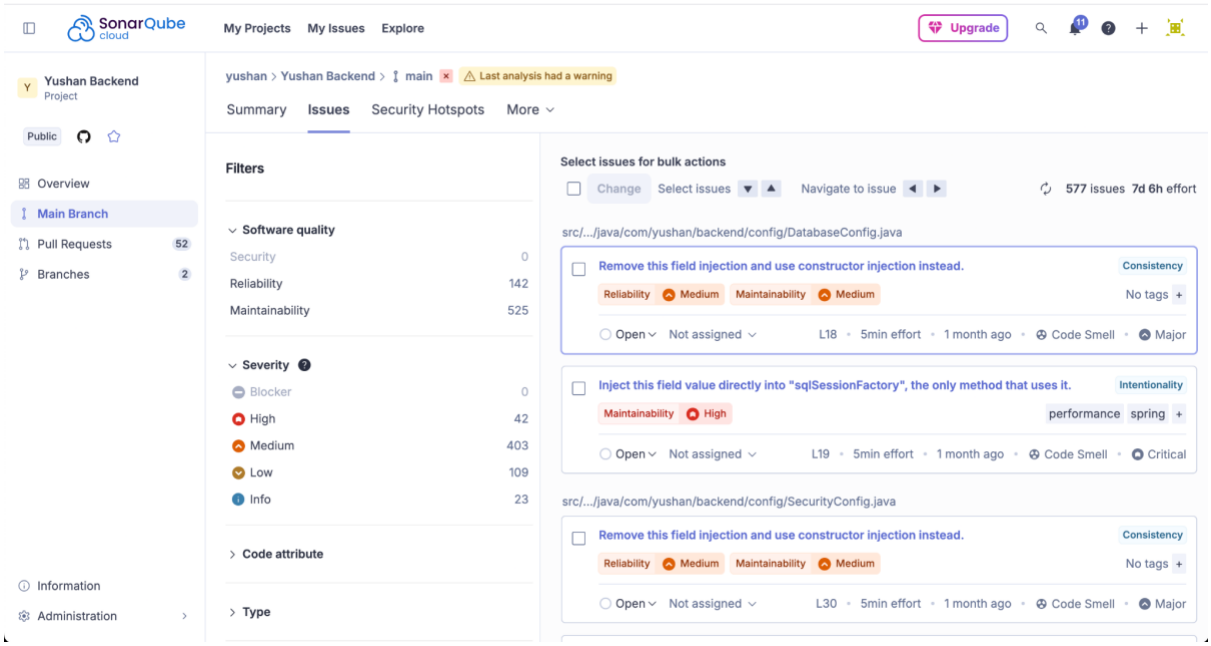
OWASP Dependency-Check Report (AS-IS)



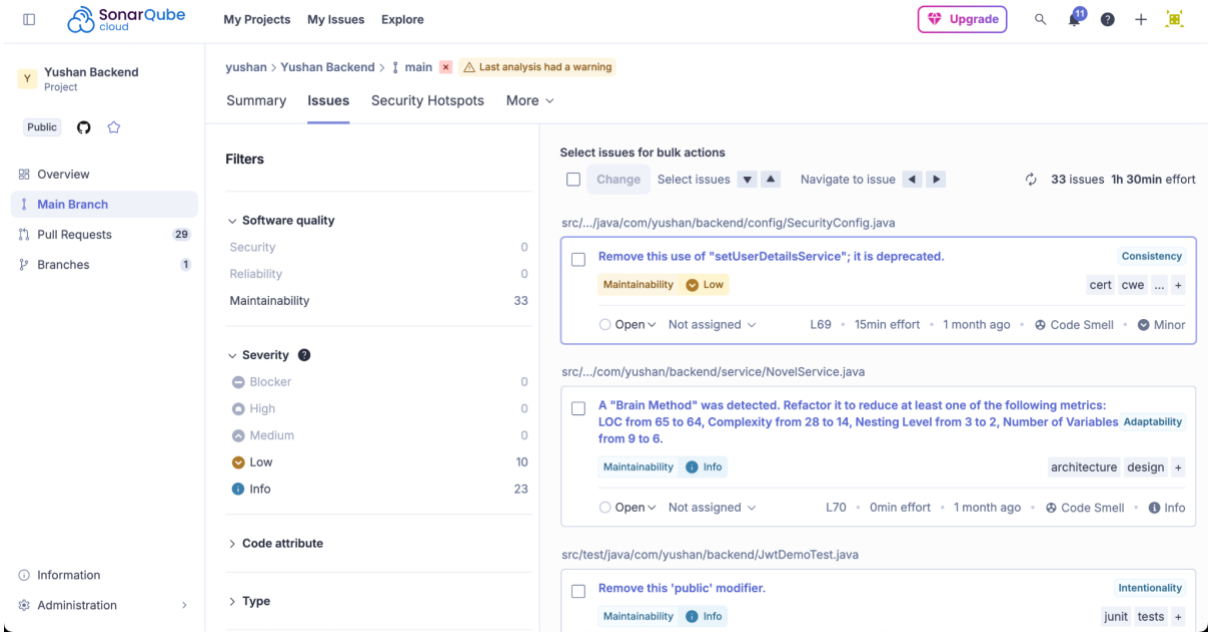
Snyk Dashboard (AS-WAS)



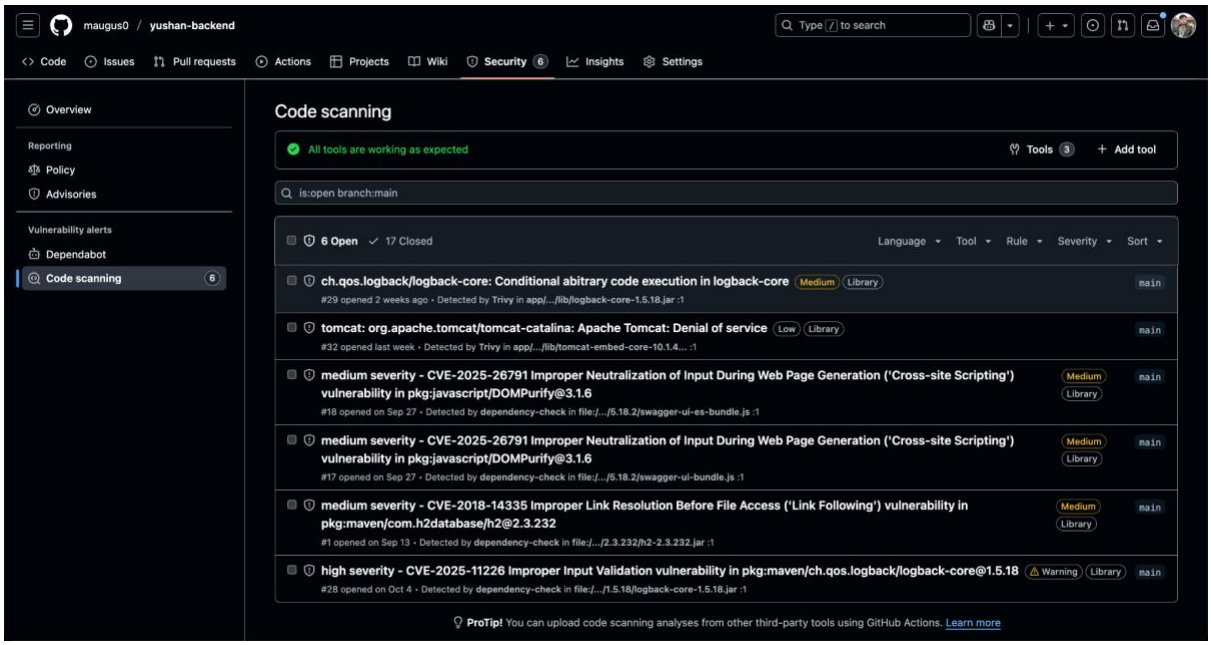
Snyk Dashboard (AS-IS)



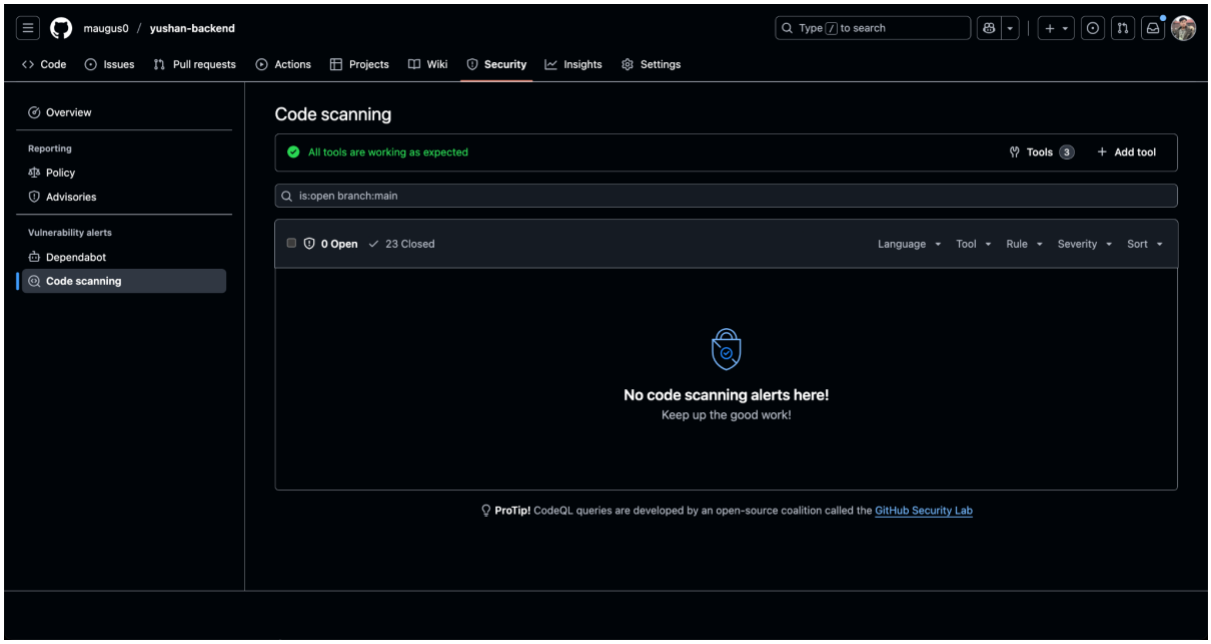
SonarQube Dashboard (AS-WAS)



SonarQube Dashboard (AS-IS)



Code Scanning from Github Security (AS-WAS)

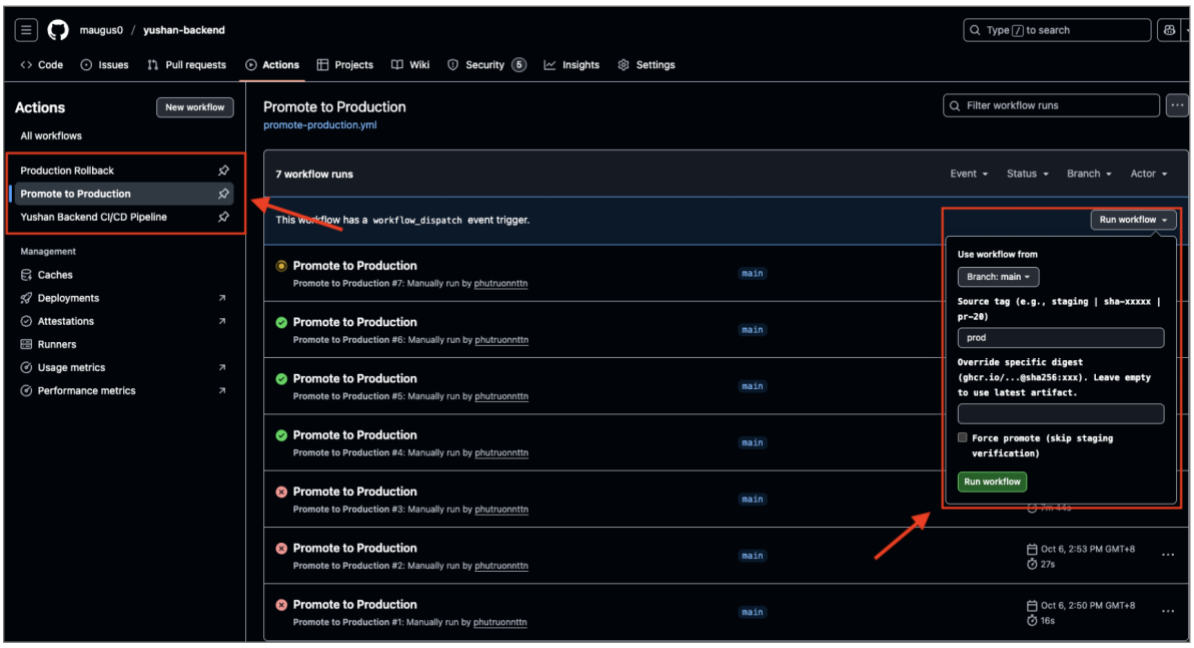


Code Scanning from Github Security (AS-IS - 23 Closed)

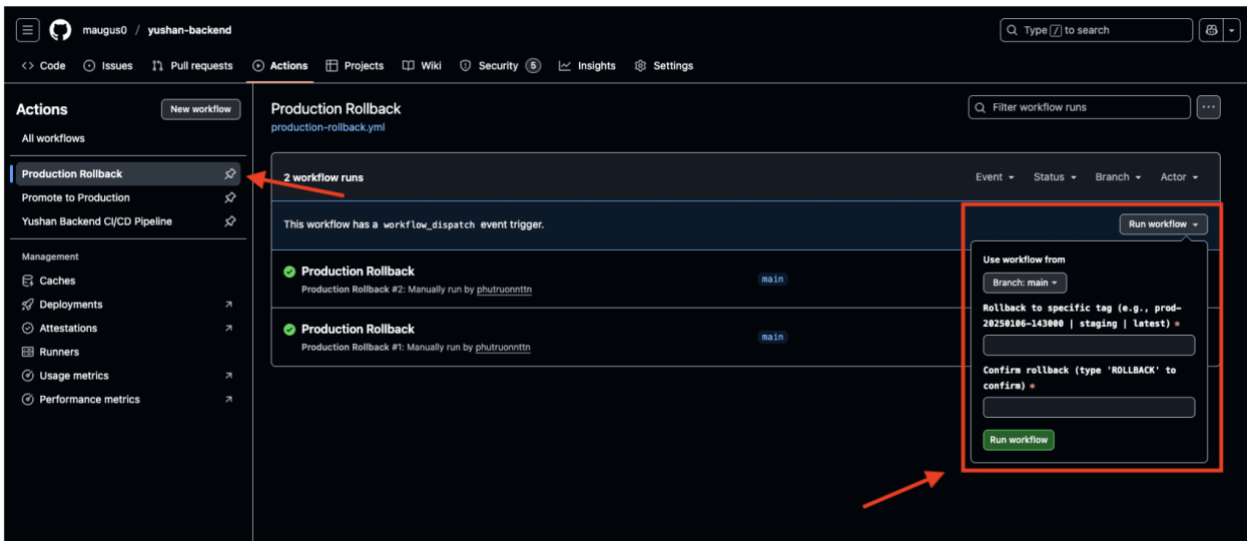
4.3 Continuous Delivery (Backend)

- Automated Staging Deployment (part of maven.yml)
  - After CI succeeds on main, the pipeline deploys staging automatically via Railway CLI.
  - Smoke tests ensure the service responds correctly.
  - OWASP ZAP baseline scan runs post-deploy for DAST coverage.
- Promotion to Production: Defined in .github/workflows/promote-production.yml (manual workflow_dispatch):
  - The operator chooses the source (staging, latest, specific tag, or digest).

- Workflow verifies image exists, retags to prod and immutable prod-<timestamp> in GHCR using docker buildx imagetools.
- Redeploys production Railway service (railway redeploy).
- Health check + smoke tests on production endpoints.
- Saves deployment metadata (production-deployment.json) as artifact for traceability.
- Requires approval through the yushan-production environment (enforced via .github/environments/production.yml).



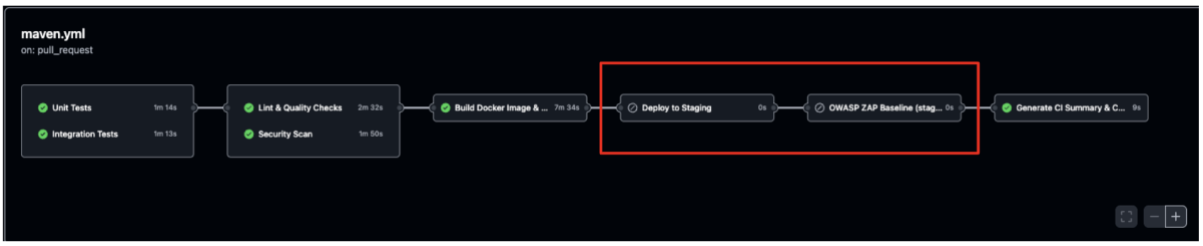
- Production Rollback: Defined in .github/workflows/production-rollback.yml (manual):
  - Requires explicit confirmation (“ROLLBACK”).
  - Resolves target image (tags like prod-YYYYMMDD-HHMMSS, staging, latest).
  - Retags target to prod in GHCR, redeploys via Railway.
  - Waits, checks /actuator/health, runs smoke tests.
  - Stores rollback metadata (rollback-info.json) as artifact.
  - Also guarded by the yushan-production environment; cannot be executed without approvals.

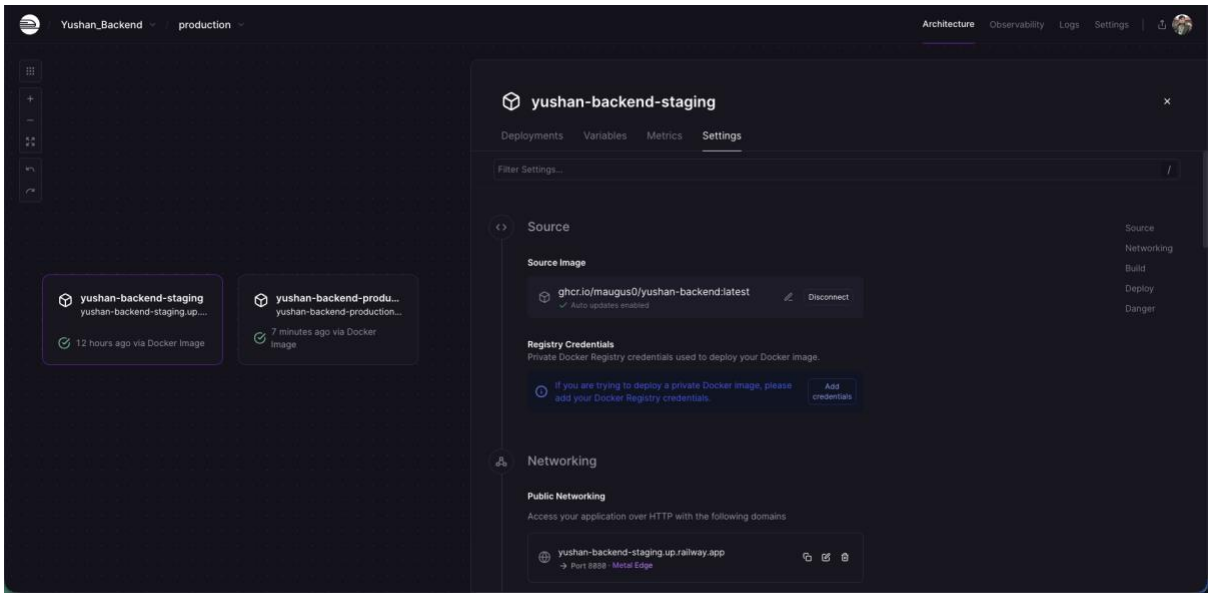


- Environment Configuration: YAML files under .github/environments/ document each environment:

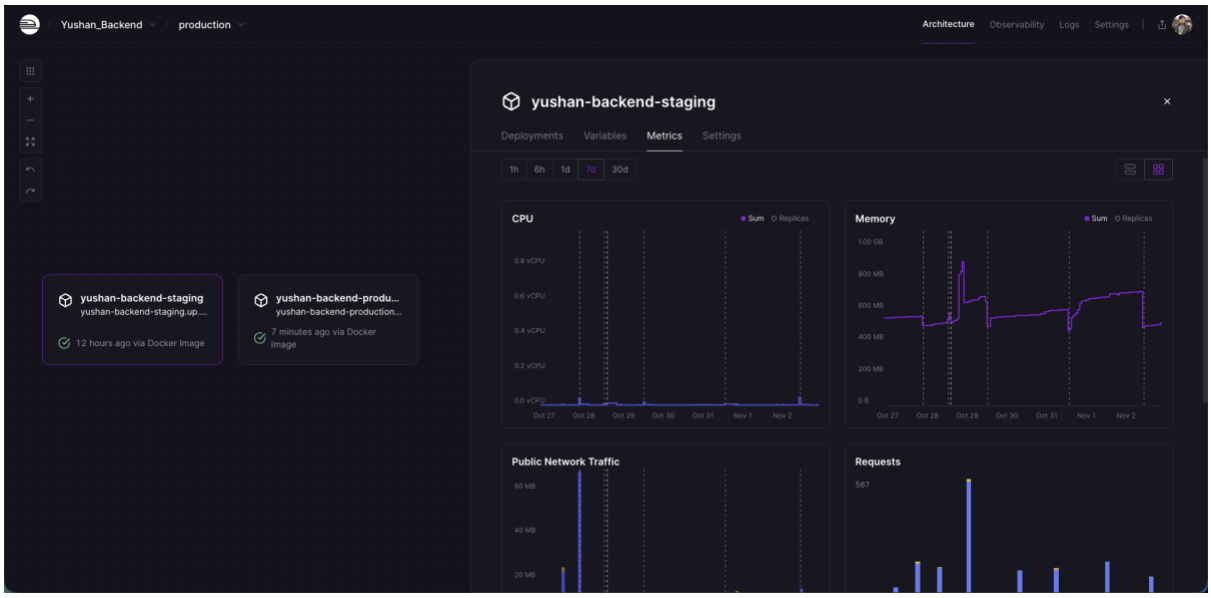
- staging.yml and production.yml define protection rules (waiting time, reviewer policy, prevent self-approval, branch restrictions).
- Document required secrets for each environment (Railway tokens, DB URLs, Redis URLs, production base URL).
- These definitions support GH's environment protection features (manual approvals, inject secrets per env).

- Toolchain Summary
  - Build/Test: Maven wrapper (./mvnw) with JDK 21.
  - Static Analysis: SpotBugs, Checkstyle, SonarCloud.
  - Security: OWASP Dependency-Check, Snyk, Trivy (container), OWASP ZAP Baseline.
  - Container: Docker Buildx, GHCR.
  - Deployment: Railway CLI via Docker container.
  - Monitoring of deployments: health checks + smoke tests in staging/production.
  - Artifacts & Reporting: GitHub Actions artifacts, CI summary Markdown, SARIF uploads to GitHub code scanning.
  - Manual Operations: separate workflows for production promotion and rollback, both requiring explicit dispatcher inputs and environment approvals.
  - Deployment policy: staging deployment runs automatically only after changes are merged into main; PR builds execute tests but skip deployment stages.

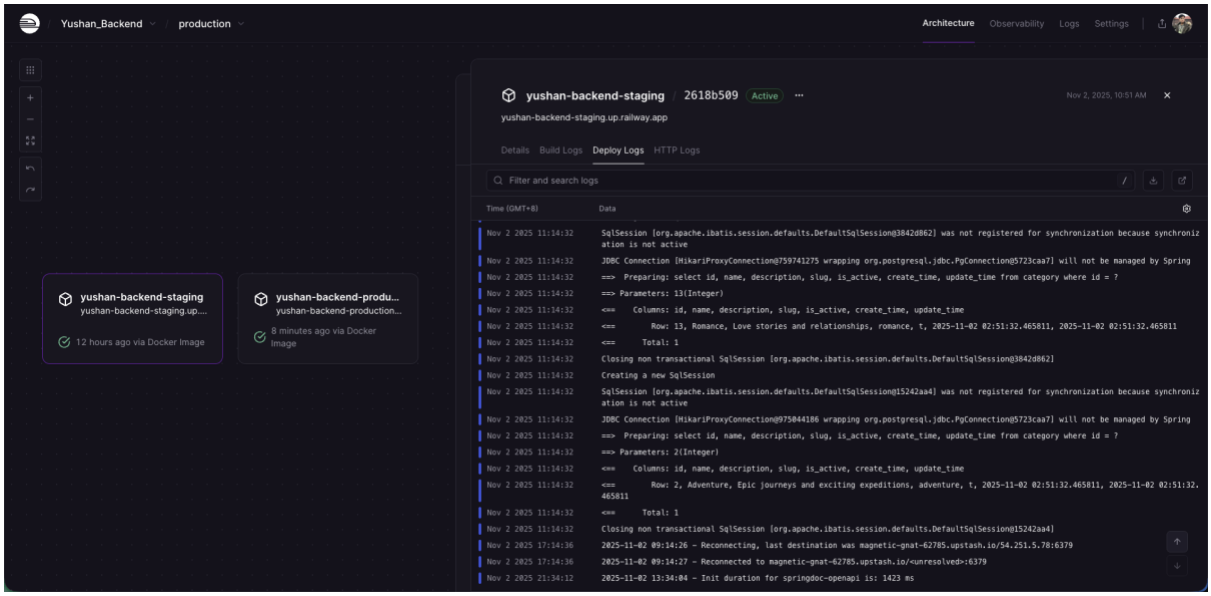




Railway Setting



Railway Metrics



Railway Logs

4.4 Non-Functional Testing Strategy (Backend)

Overview: The backend non-functional testing strategy ensures Yushan meets performance, security, and reliability requirements beyond basic functionality. Testing is conducted at multiple levels using industry-standard tools integrated into both manual and automated workflows.

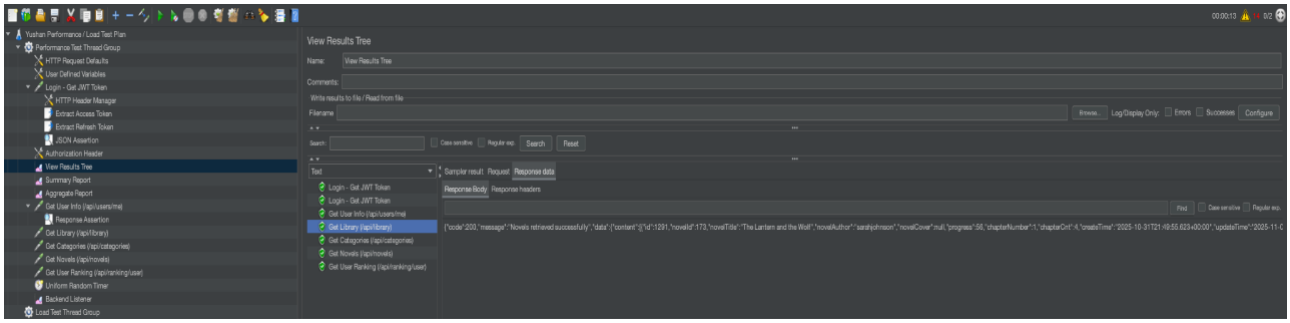
4.4.1 API Testing

Swagger OpenAPI Documentation

- **Purpose**: Interactive API documentation and manual endpoint testing
- **Implementation**: Spring Boot integration with Springdoc OpenAPI
- **Usage**:
    - Developers validate endpoint contracts during development
    - QA team performs exploratory testing of API responses
    - Provides schema validation for request/response payloads
- **Access**: Available at /swagger-ui.html in staging and development environments

Postman Collections

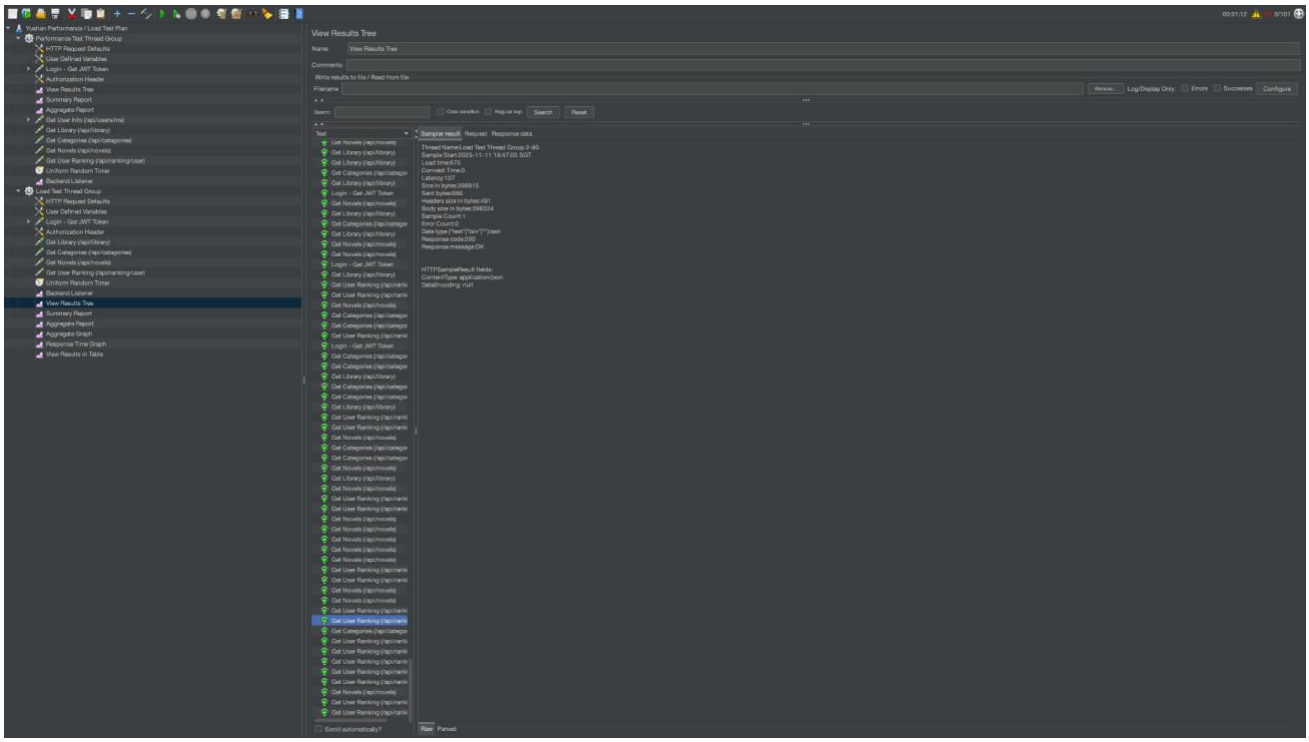
- **Tool**: Postman v10+
- **Coverage**: Comprehensive test suites covering all backend endpoints
- **Test Scenarios**:
    - Authentication flows (login, token refresh, logout)
    - CRUD operations for all domain entities
    - Error handling and validation responses
    - Edge cases and boundary conditions
- **Organization**: Collections structured by feature modules
- **Execution**: Manual testing during development and pre-release validation
- **Artifacts**: Postman collection JSON files maintained in repository documentation



4.4.2 Performance Testing

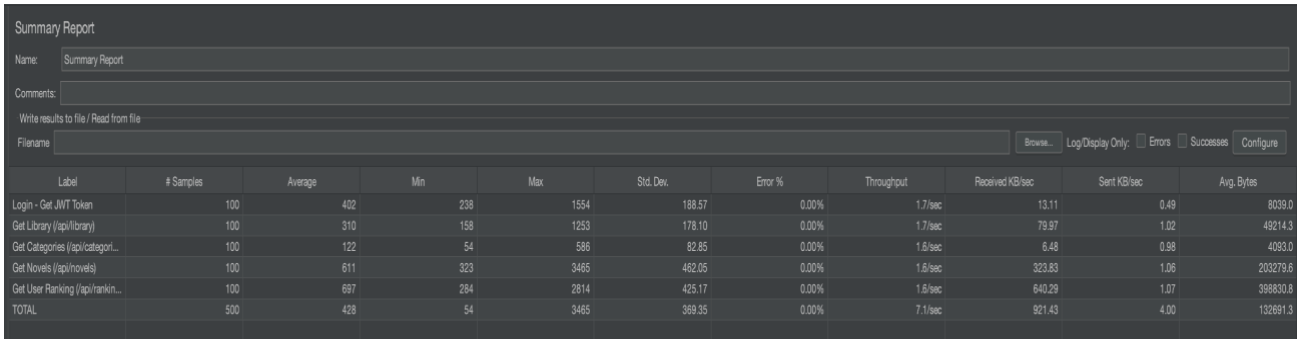
Load Testing with Apache JMeter

- **Tool**: Apache JMeter 5.6+
- **Test Scenarios**: Realistic user journey simulations
    1. User authentication (login)
    2. Profile retrieval
    3. Library browsing
    4. Category exploration
    5. Novel search operations
    6. User ranking queries



Configuration:

- Thread groups simulate concurrent users (10-1000 users)
- Ramp-up period: 30-60 seconds
- Test duration: 5-15 minutes per scenario
- Think time: 2-5 seconds between requests



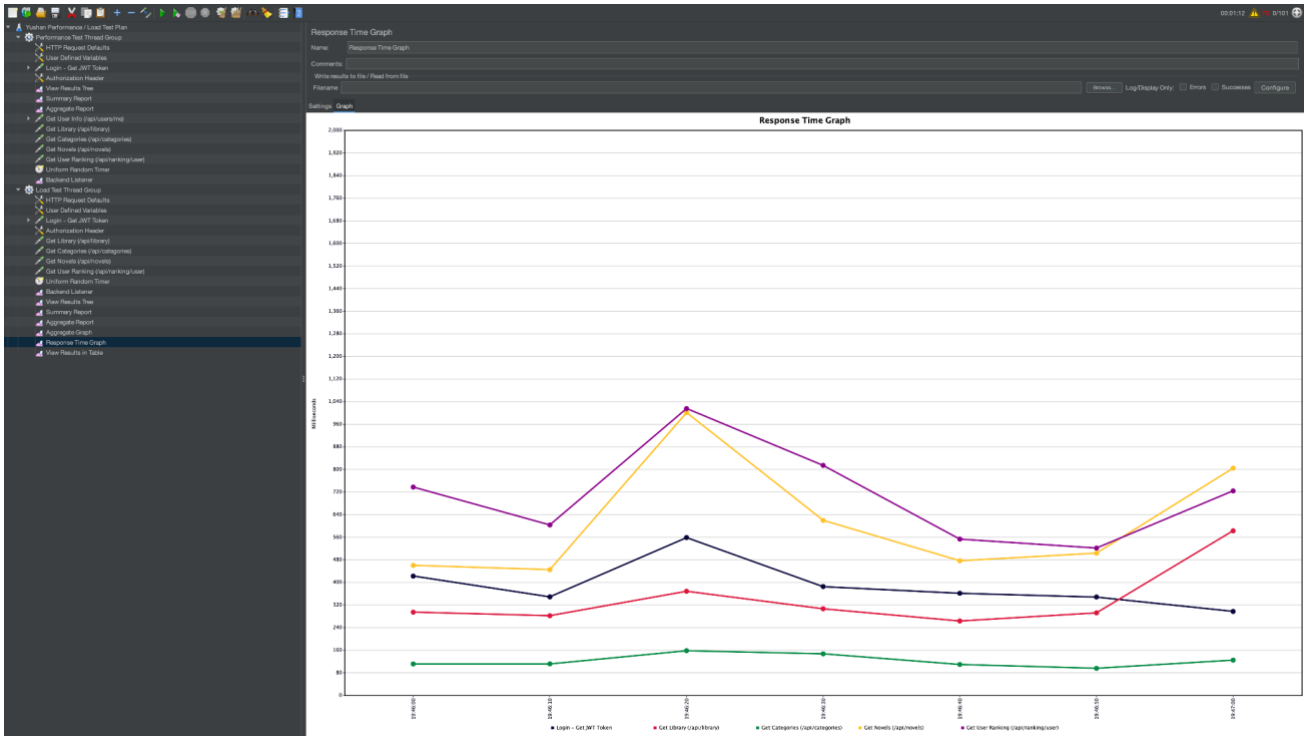
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Login - Get JWT Token	100	402	238	1554	188.57	0.00%	1.7/sec	13.11	0.49	8039.0
Get Library (/api/library)	100	310	158	1253	178.10	0.00%	1.7/sec	79.97	1.02	49214.3
Get Categories (/api/categori...	100	122	54	586	82.85	0.00%	1.6/sec	6.48	0.98	4093.0
Get Novels (/api/novels)	100	611	323	3465	462.05	0.00%	1.6/sec	323.83	1.06	203279.6
Get User Ranking (/api/rankin...	100	697	284	2814	425.17	0.00%	1.6/sec	640.29	1.07	398830.8
TOTAL	500	428	54	3465	369.35	0.00%	7.1/sec	921.43	4.00	132691.3

Metrics Collected:

- Response time (average, median, 90th/95th/99th percentile)
- Throughput (requests per second)
- Error rate
- Server resource utilization (via Railway metrics)

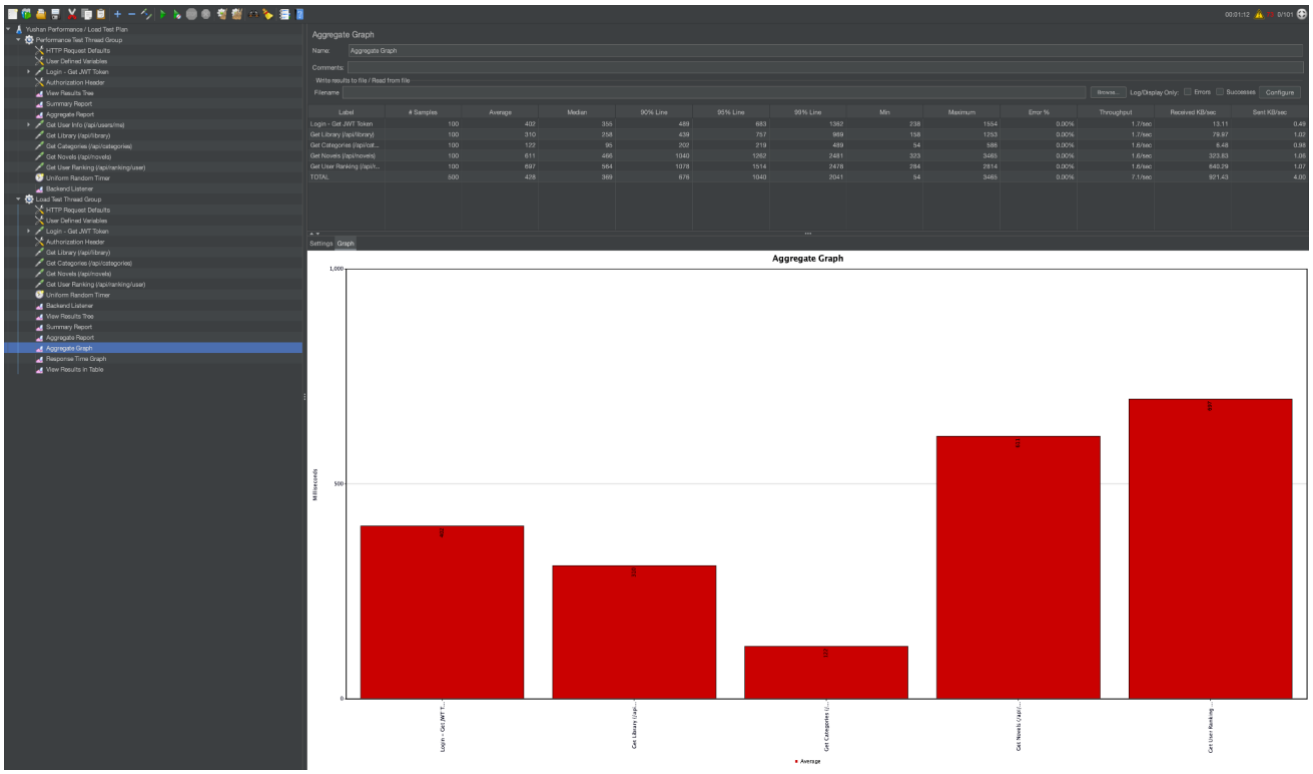
Performance Baselines:

- API response time: < 200ms for 95% of requests
- Database query time: < 100ms for simple queries
- Concurrent user capacity: 100+ simultaneous users
- Error rate: < 1% under normal load



Stress Testing

- **Purpose**: Identify breaking points and system limits
- **Approach**: Gradually increase load beyond normal capacity
- **Monitoring**: Railway metrics dashboard tracks CPU, memory, and response degradation
- **Recovery Testing**: Validate system recovery after stress conditions



4.4.3 Security Testing

OWASP Dependency-Check

- **Integration**: Automated in CI pipeline (security-scan job)
- **Frequency**: Every PR and main branch push
- **Coverage**: Scans all Maven dependencies for known CVEs
- **Action**: SARIF reports uploaded to GitHub Code Scanning
- **Threshold**: Builds fail on critical vulnerabilities

Snyk Vulnerability Scanning

- **Tool**: Snyk CLI integrated in CI
- **Scope**:
    - Open source dependency vulnerabilities
    - Container image scanning (via Docker integration)
    - License compliance checks
- **Reporting**: SARIF format uploaded to GitHub security tab
- **Remediation**: Automated PRs for dependency upgrades when fixes available

Trivy Container Scanning

- **Integration**: Runs after Docker image build (docker-build job)
- **Target**: Multi-arch images (linux/amd64, linux/arm64)
- **Scan Types**:
    - OS package vulnerabilities
    - Application dependency vulnerabilities
    - Misconfigurations
- **Output**: SARIF uploaded to GitHub Code Scanning

OWASP ZAP DAST

- **Tool**: OWASP ZAP Baseline Scan
- **Timing**: Post-deployment to staging environment
- **Execution**: Automated via zaproxy/action-baseline
- **Target**: Live staging endpoints
- **Coverage**:
    - SQL injection attempts
    - XSS vulnerability probing
    - CSRF token validation
    - Security header verification
- **Reports**: HTML, JSON, and Markdown artifacts generated
- **Integration**: Runs in dast-zap job after staging-deploy completes

4.4.4 Database Testing

Data Integrity Testing

- **Approach**: Integration tests verify:
    - Foreign key constraints
    - Unique constraints
    - Data type validations
    - Transaction rollback behavior
- **Framework**: Spring Boot Test with @DataJpaTest
- **Database**: Supabase PostgreSQL (dedicated test schema for integration tests)

Query Performance Testing

- **Method**: Manual analysis using:
    - PostgreSQL EXPLAIN ANALYZE
    - Supabase dashboard query statistics
    - Spring Boot actuator metrics
- **Optimization**: Indexes validated for frequently accessed columns
- **Monitoring**: Slow query logs reviewed during performance testing

4.4.5 Compatibility Testing

Browser Compatibility (API Perspective)

- **CORS Configuration**: Validated across modern browsers
- **Content Negotiation**: JSON responses tested with various Accept headers
- **WebSocket Support**: Real-time features verified in Chrome, Firefox, Safari

Client Compatibility

- **REST API Versioning**: Endpoints support multiple API versions for backward compatibility
- **Content Types**: JSON (primary), XML (where applicable)
- **HTTP Methods**: Full REST verb support (GET, POST, PUT, PATCH, DELETE)

4.4.6 Monitoring and Observability

Railway Platform Metrics

- **Real-time Monitoring**:
  - CPU utilization
  - Memory consumption
  - Network I/O
  - Request rates
  - Response times
- **Logging**: Centralized logs via Railway dashboard
- **Alerting**: Manual monitoring during load tests

Spring Boot Actuator

- **Endpoints Exposed**:
  - /actuator/health: Application health status
  - /actuator/info: Build and version information
  - /actuator/metrics: Performance metrics
- **Integration**: Health checks automated in CI/CD pipeline
- **Smoke Tests**: Verify actuator endpoints post-deployment

4.4.7 Usability Testing (API Design)

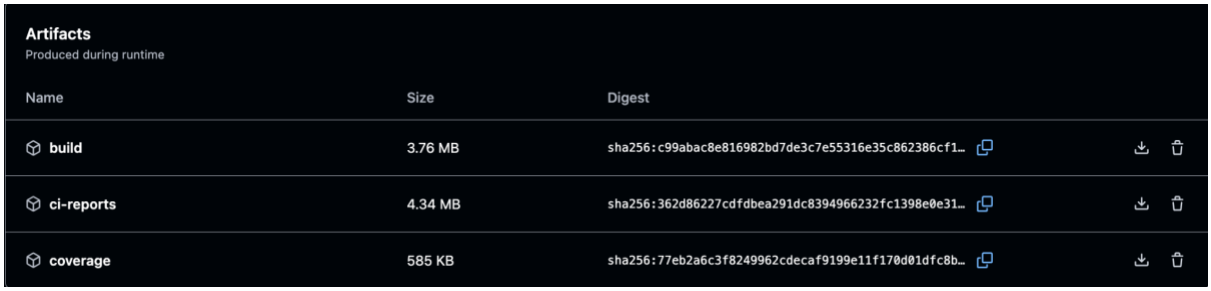
- **RESTful Principles**: Endpoints follow REST conventions
- **Error Messages**: Descriptive error responses with appropriate HTTP status codes
- **API Documentation**: Clear Swagger annotations and examples
- **Response Consistency**: Standardized response structures across endpoints

4.4.8 Manual Testing

- **Exploratory Testing**: Team conducts ad-hoc testing of new features
- **Regression Testing**: Manual verification of critical paths after releases
- **User Acceptance Testing**: Stakeholder validation in staging environment

4.5 Source Control Strategy (Frontend)

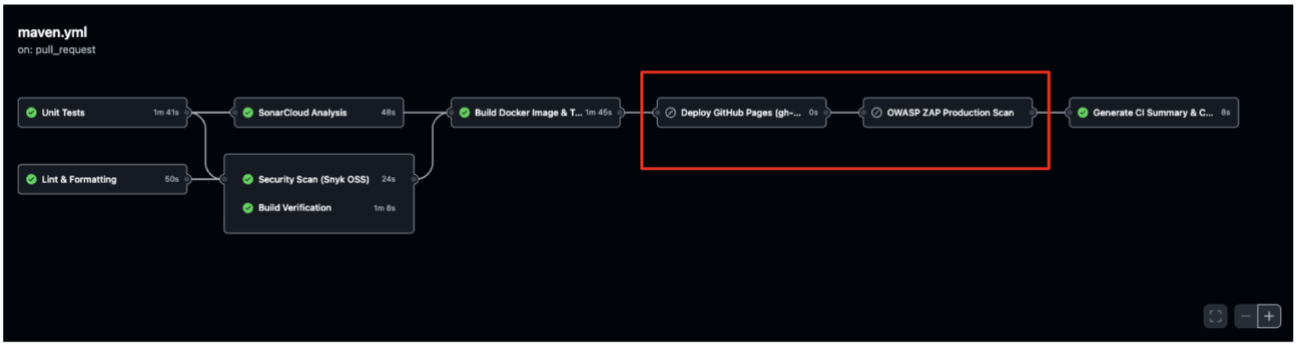
- Repository: yushan-frontend (standalone GitHub repository).
- Branching & protection: main is the live branch; feature branches require PRs. Direct pushes to main are blocked. Required status checks: lint-quality, unit-tests, sonarcloud, security-scan, build-verification, docker-build, zap-dast, reports-summary.
- PR policy: reviewers must approve before merge; concurrency guard prevents two runs on same branch/PR (${{ github.workflow }}-${{ github.ref }}).
- Secrets: SONAR_TOKEN, SNYK_TOKEN as repo secrets. GitHub Pages deploy uses the job-scoped GITHUB_TOKEN.
- Artifacts: coverage reports, build outputs, SARIF files uploaded per workflow. Docker images pushed to GHCR for reuse and scanning.



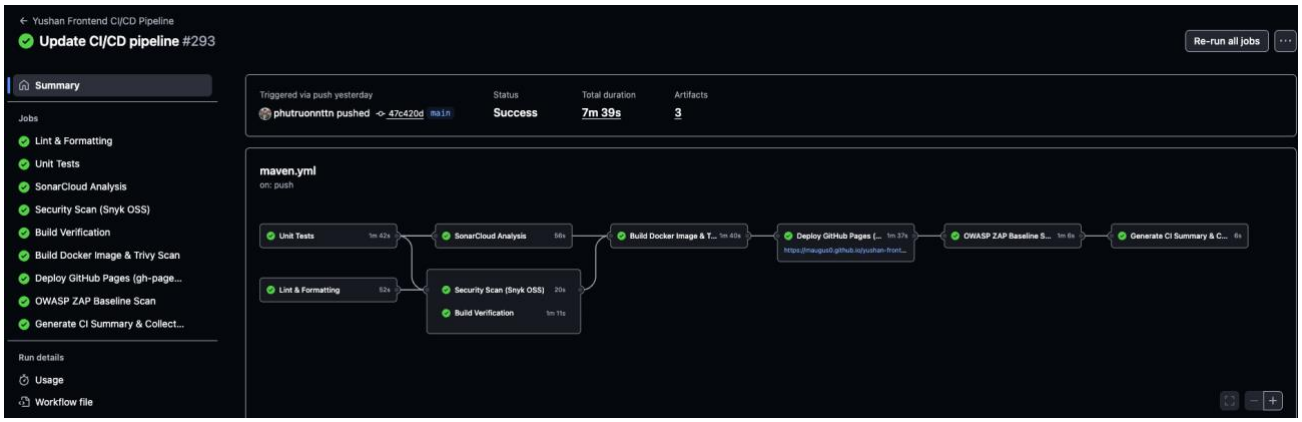
4.6 Continuous Integration (Frontend)

- Workflow: Yushan Frontend CI/CD Pipeline

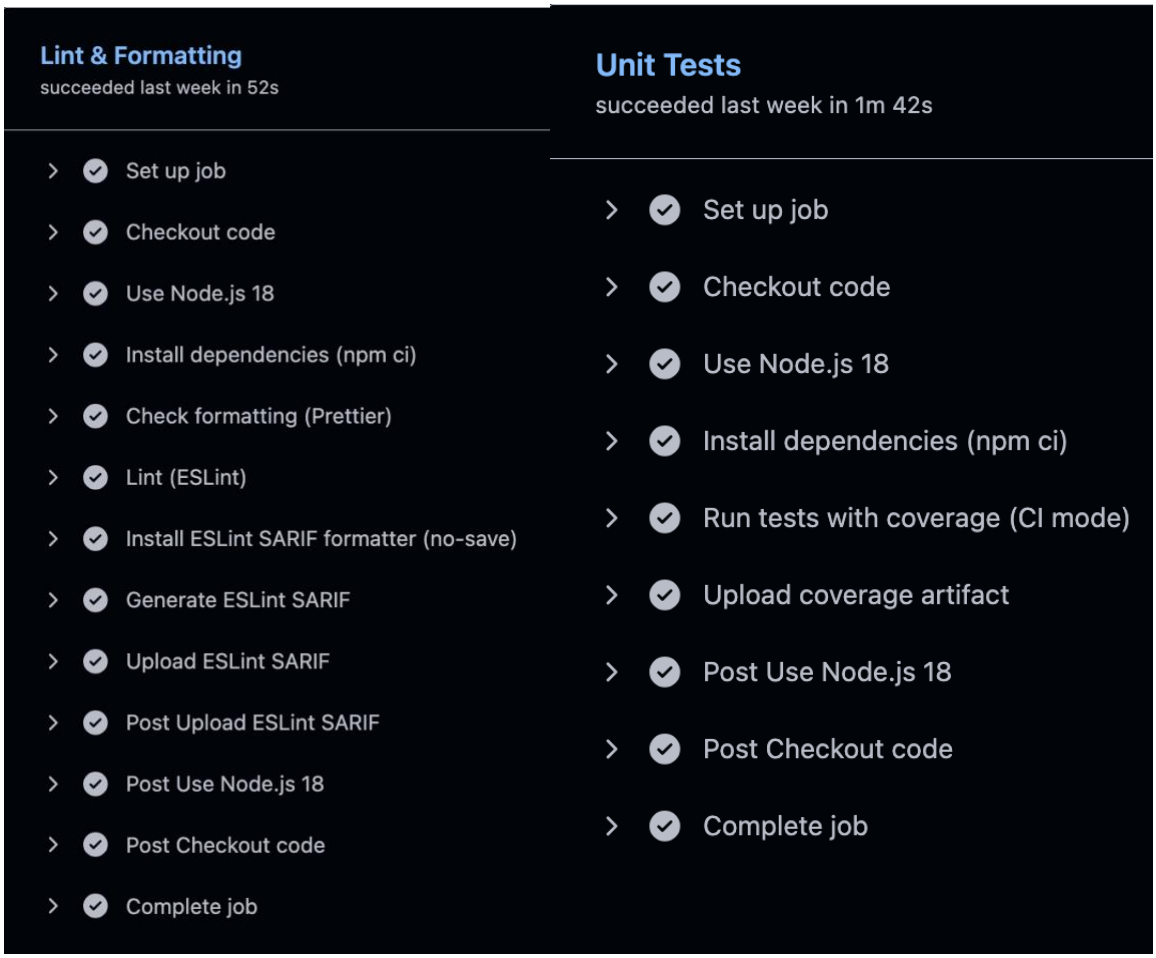
- Trigger summary
    - pull_request to main (opened/synchronize/reopened) → full CI without deployment.

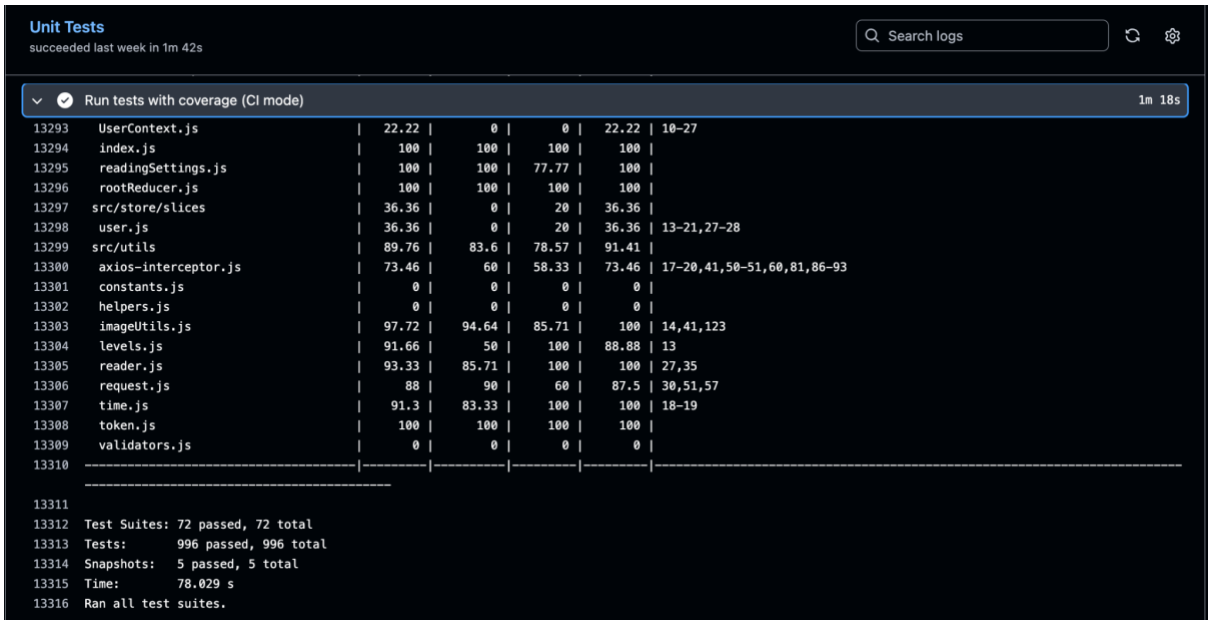


    - push to main → full CI plus deployment to GitHub Pages.



- Job-by-job breakdown
    - lint-quality
        - Checkout (actions/checkout@v4), setup Node 18 (actions/setup-node@v4 with npm cache).
        - npm ci → npx prettier --check → npx eslint.
        - Generates ESLint SARIF and uploads via github/codeql-action/upload-sarif@v3.
    - unit-tests
        - Repeats checkout/setup/node install.
        - Runs npm test -- --watch=false --ci --coverage; stores coverage artifact (coverage/).

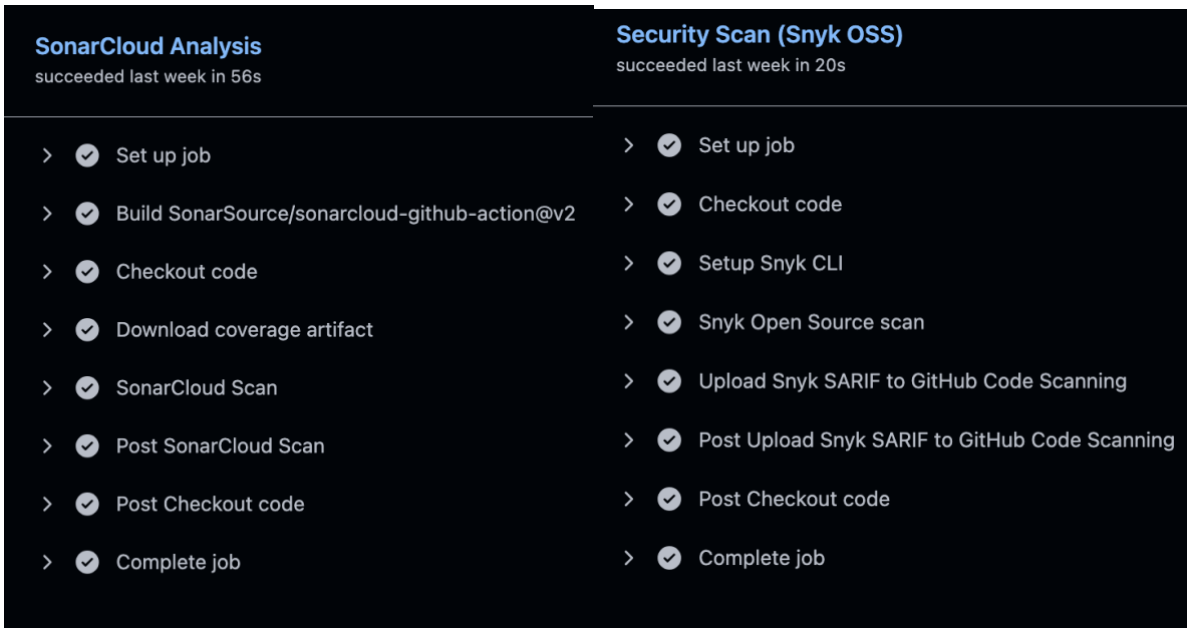




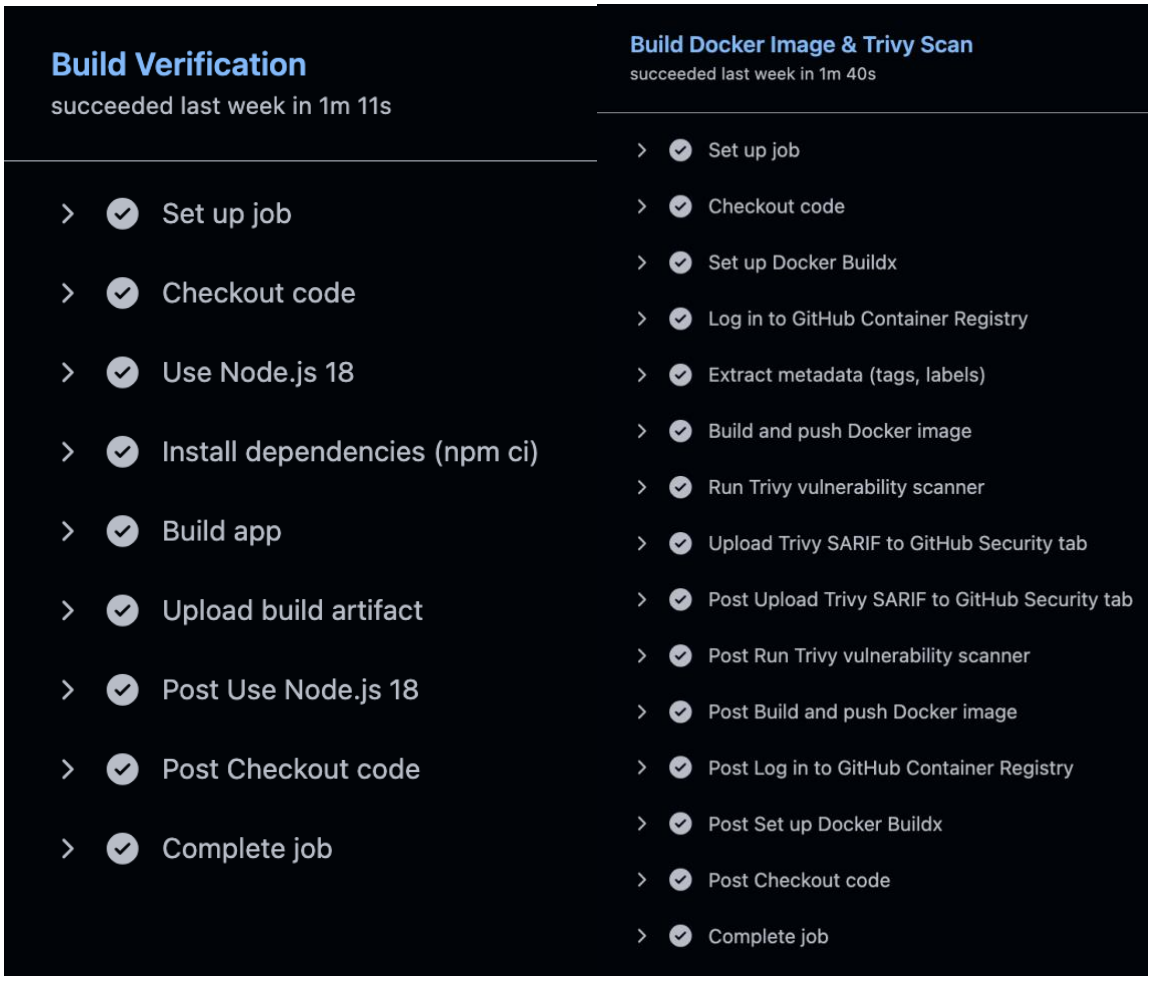
Unit Test Execution Output (GitHub Actions)

- sonarcloud (needs unit-tests)
  - Downloads coverage artifact; invokes SonarSource/sonarcloud-github-action@v2 (uses SONAR_TOKEN).
- security-scan (needs lint-quality & unit-tests)
  - Installs Snyk CLI (snyk/actions/setup@master).

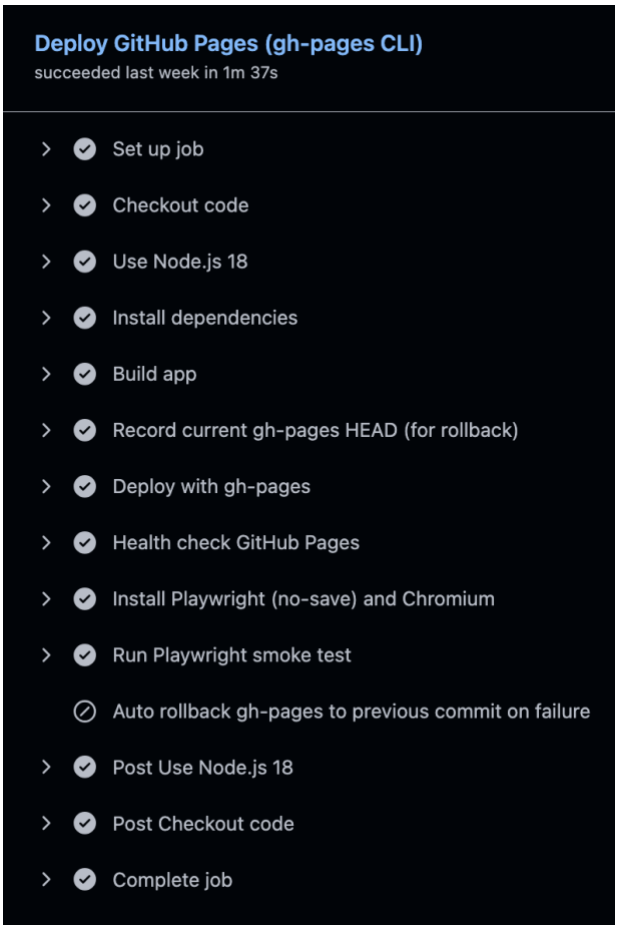
- Authenticates with SNYK_TOKEN, runs snyk test (high severity threshold, SARIF output).
- Uploads Snyk SARIF to GitHub code scanning.



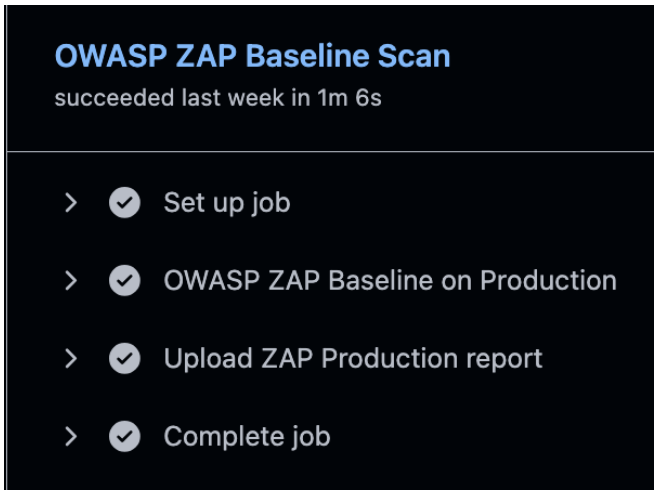
- build-verification (needs lint-quality & unit-tests)
  - Builds production bundle via npm run build; uploads build/ artifact.
- docker-build (needs lint-quality, unit-tests, build-verification, security-scan)
  - Builds Docker image with Buildx and pushes to GHCR (tags: branch, PR, SHA, latest).
  - Trivy scan produces SARIF, uploaded to code scanning.



- deploy-pages (needs zap-dast, only when branch target is main)
    - Checkout with history, npm ci, npm run build.
    - Captures previous gh-pages commit to PREV_GHPAGES_SHA.
    - Deploys via npx gh-pages using GITHUB_TOKEN.
    - Health-check loop: fetch index, verify main JS/CSS bundles return HTTP 200, run Playwright smoke script (Chromium) to ensure DOM renders without console errors.
    - On failure, rolls back gh-pages to PREV_GHPAGES_SHA.

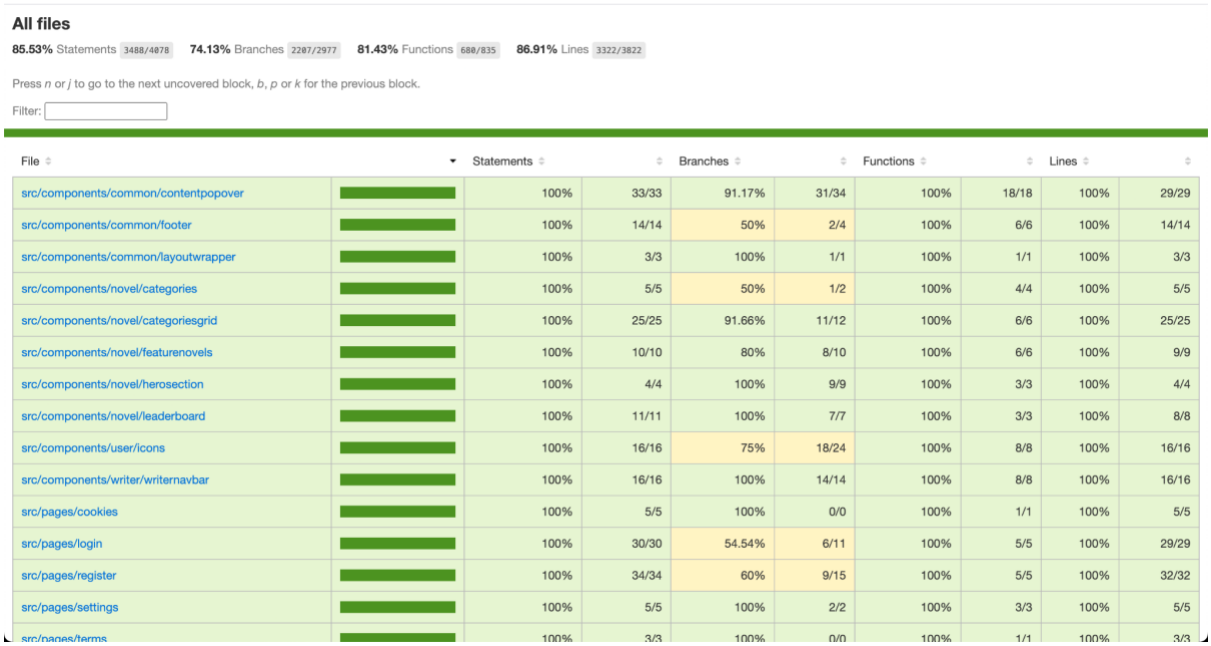


- zap-dast (needs docker-build)
  - Runs OWASP ZAP baseline against the live GitHub Pages URL and uploads HTML/JSON/Markdown reports.



- reports-summary (needs zap-dast)
  - Downloads all artifacts; generates Markdown summary listing coverage, lint, Snyk, Trivy, ZAP outputs; uploads consolidated ci-reports.

- Permissions & caching
  - Jobs request minimal permissions (e.g., pages: write + id-token: write for deploy, security-events: write only where SARIF uploaded).
  - actions/setup-node uses npm cache; Docker Buildx uses GHA cache.



All files

85.53% Statements 3488/4078 74.13% Branches 2207/2977 81.43% Functions 680/835 86.91% Lines 3322/3822

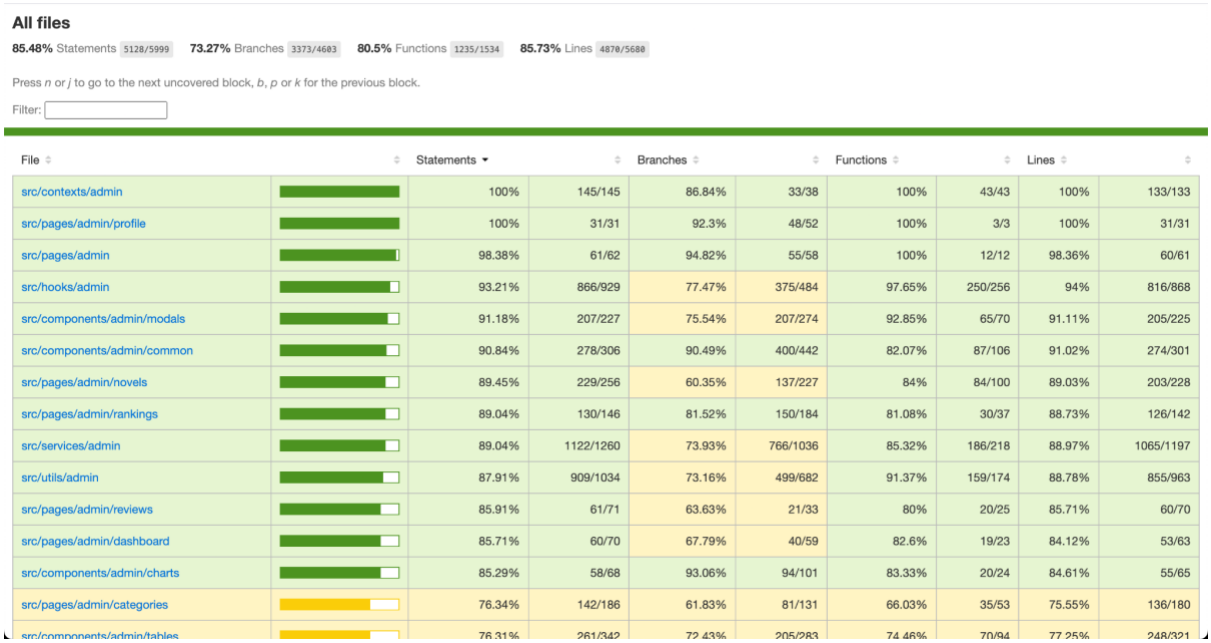
Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
src/components/common/contentpopover	100%	33/33	91.17%	31/34	100%	18/18	100%	29/29
src/components/common/footer	100%	14/14	50%	2/4	100%	6/6	100%	14/14
src/components/common/layoutwrapper	100%	3/3	100%	1/1	100%	1/1	100%	3/3
src/components/novel/categories	100%	5/5	50%	1/2	100%	4/4	100%	5/5
src/components/novel/categoriesgrid	100%	25/25	91.66%	11/12	100%	6/6	100%	25/25
src/components/novel/featurenovels	100%	10/10	80%	8/10	100%	6/6	100%	9/9
src/components/novel/herosection	100%	4/4	100%	9/9	100%	3/3	100%	4/4
src/components/novel/leaderboard	100%	11/11	100%	7/7	100%	3/3	100%	8/8
src/components/user/icons	100%	16/16	75%	18/24	100%	8/8	100%	16/16
src/components/writer/writernavbar	100%	16/16	100%	14/14	100%	8/8	100%	16/16
src/pages/cookies	100%	5/5	100%	0/0	100%	1/1	100%	5/5
src/pages/login	100%	30/30	54.54%	6/11	100%	5/5	100%	29/29
src/pages/register	100%	34/34	60%	9/15	100%	5/5	100%	32/32
src/pages/settings	100%	5/5	100%	2/2	100%	3/3	100%	5/5
src/pages/terms	100%	3/3	100%	0/0	100%	1/1	100%	3/3

Coverage report of Yushan-Frontend (Jest's Istanbul)

SonarQube Report of Yushan-Frontend



All files

85.48% Statements 5128/5999 73.27% Branches 3373/4603 80.5% Functions 1235/1534 85.73% Lines 4870/5680

Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
src/contexts/admin	100%	145/145	86.84%	33/38	100%	43/43	100%	133/133
src/pages/admin/profile	100%	31/31	92.3%	48/52	100%	3/3	100%	31/31
src/pages/admin	98.38%	61/62	94.82%	55/58	100%	12/12	98.36%	60/61
src/hooks/admin	93.21%	866/929	77.47%	375/484	97.65%	250/256	94%	816/868
src/components/admin/modals	91.18%	207/227	75.54%	207/274	92.85%	65/70	91.11%	205/225
src/components/admin/common	90.84%	278/306	90.49%	400/442	82.07%	87/106	91.02%	274/301
src/pages/admin/novels	89.45%	229/256	60.35%	137/227	84%	84/100	89.03%	203/228
src/pages/admin/rankings	89.04%	130/146	81.52%	150/184	81.08%	30/37	88.73%	126/142
src/services/admin	89.04%	1122/1260	73.93%	766/1036	85.32%	186/218	88.97%	1065/1197
src/utils/admin	87.91%	909/1034	73.16%	499/682	91.37%	159/174	88.78%	855/963
src/pages/admin/reviews	85.91%	61/71	63.63%	21/33	80%	20/25	85.71%	60/70
src/pages/admin/dashboard	85.71%	60/70	67.79%	40/59	82.6%	19/23	84.12%	53/63
src/components/admin/charts	85.29%	58/68	93.06%	94/101	83.33%	20/24	84.61%	55/65
src/pages/admin/categories	76.34%	142/186	61.83%	81/131	66.03%	35/53	75.55%	136/180
src/components/admin/tables	76.31%	261/342	72.43%	205/283	74.46%	70/94	77.25%	248/321

Coverage report of Yushan-Admin (Jest's Istanbul)

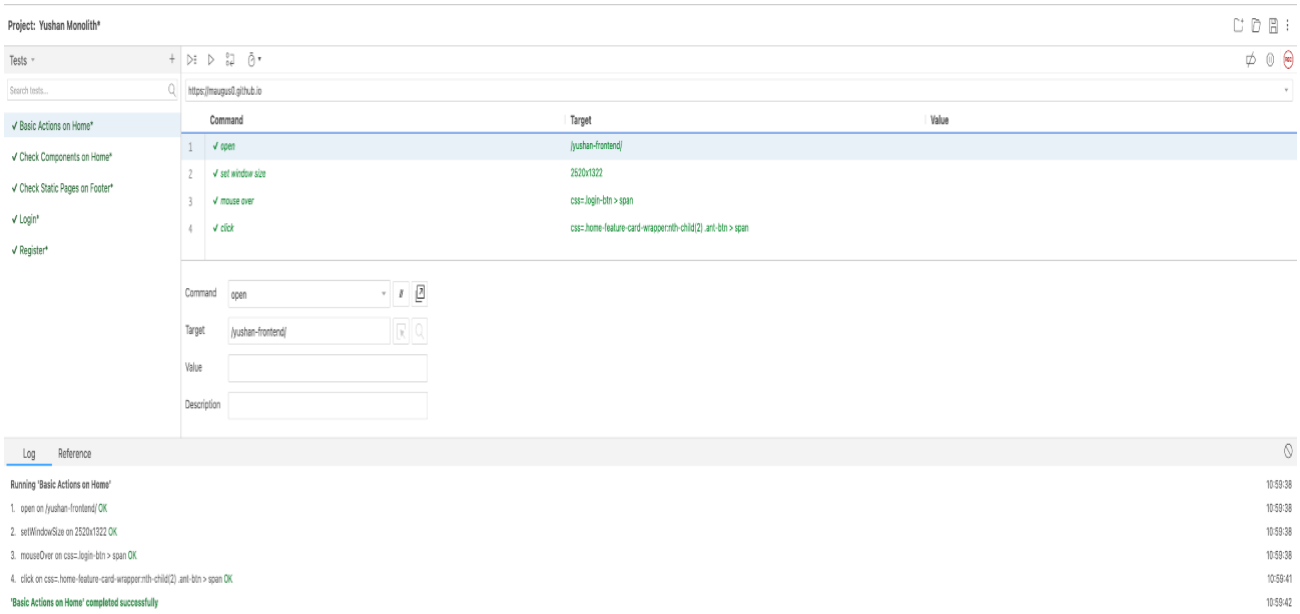
SonarQube Report of Yushan-Admin

4.7 Continuous Delivery (Frontend)

- Automation scope: GitHub Pages deployment triggers automatically only after commits land on main; PR runs perform checks without deploying.
- Artifacts: GHCR container image produced during CI supports DAST scans and optional container hosting experiments.
- Rollback: deploy-pages captures the previous gh-pages commit and restores it automatically if health checks fail; rerunning the workflow redeploys the latest stable build.
- Verification: curl/Playwright checks ensure static assets respond correctly and the UI renders without console errors.
- Environment: GitHub Pages environment (no manual approval) exposes the public URL and ensures Actions have necessary permissions.

4.8 Non-Functional Testing Strategy (Frontend)

Overview: The frontend non-functional testing strategy ensures Yushan delivers a performant, secure, and user-friendly interface across all supported platforms and browsers. Testing combines automated CI checks with manual validation using industry-standard tools.



4.8.1 End-to-End (E2E) Testing

Selenium IDE

- **Tool**: Selenium IDE Browser Extension
- **Purpose**: Automated UI/UX flow testing
- **Coverage**: Basic user workflows including:
    - User registration and login
    - Navigation through main application sections
    - Form submissions and validations
    - Search functionality
    - Content browsing

Limitations with Ant Design:

- Ant Design (antd) component library complexity requires custom scripts for advanced scenarios
- Shadow DOM elements and dynamic component IDs need manual selector refinement
- Complex interactions (modals, dropdowns, date pickers) scripted manually

Test Execution:

- Manual playback during development
- Recorded tests serve as regression suite baseline
- Re-recorded after major UI changes

Playwright Smoke Tests

- **Integration**: Automated in deploy-pages job
- **Browser**: Chromium (default)
- **Validation**:
    - Page loads without errors
    - Critical DOM elements render
    - No console errors logged
    - Main JavaScript/CSS bundles accessible (HTTP 200)
- **Failure Handling**: Automatic rollback to previous gh-pages commit

4.8.2 UI/Visual Testing

Manual Visual Inspection

- **Method**: Browser DevTools for visual regression
- **Browsers Tested**:
    - Chrome/Chromium (latest)
    - Firefox (latest)
    - Safari (macOS, iOS)
    - Edge (latest)
- **Devices**:
    - Desktop (1920×1080, 1366×768)
    - Tablet (iPad, Android tablets)
    - Mobile (iOS, Android various screen sizes)

Responsive Design Validation

- **Approach**: Chrome DevTools Device Mode
- **Breakpoints Verified**:
    - Mobile: < 768px
    - Tablet: 768px - 1024px
    - Desktop: > 1024px
- **Components Checked**:
    - Navigation menus collapse appropriately
    - Tables become scrollable on mobile
    - Forms stack vertically on narrow screens
    - Images scale proportionally

4.8.3 Performance Testing

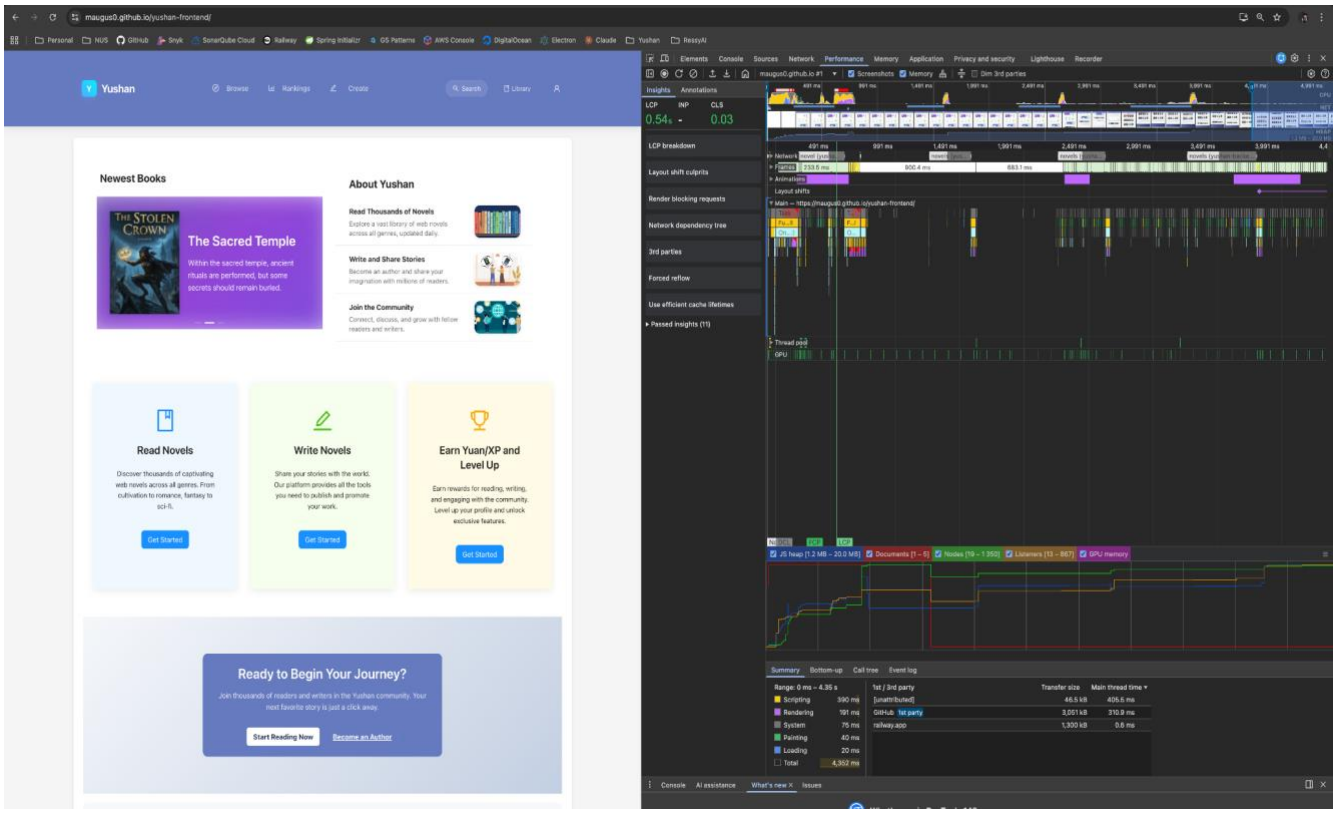
Browser Performance Metrics

- **Tool**: Chrome DevTools Performance Tab
- **Metrics Collected**:
    - First Contentful Paint (FCP): Target < 1.5s
    - Largest Contentful Paint (LCP): Target < 2.5s
    - Time to Interactive (TTI): Target < 3.5s
    - Total Blocking Time (TBT): Target < 300ms
    - Cumulative Layout Shift (CLS): Target < 0.1

Analysis Areas:

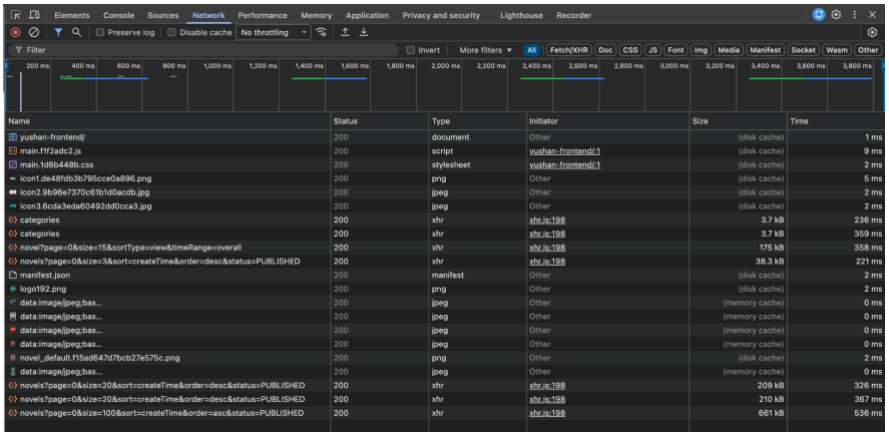
- JavaScript bundle size and load time
- CSS rendering performance
- Network waterfall optimization

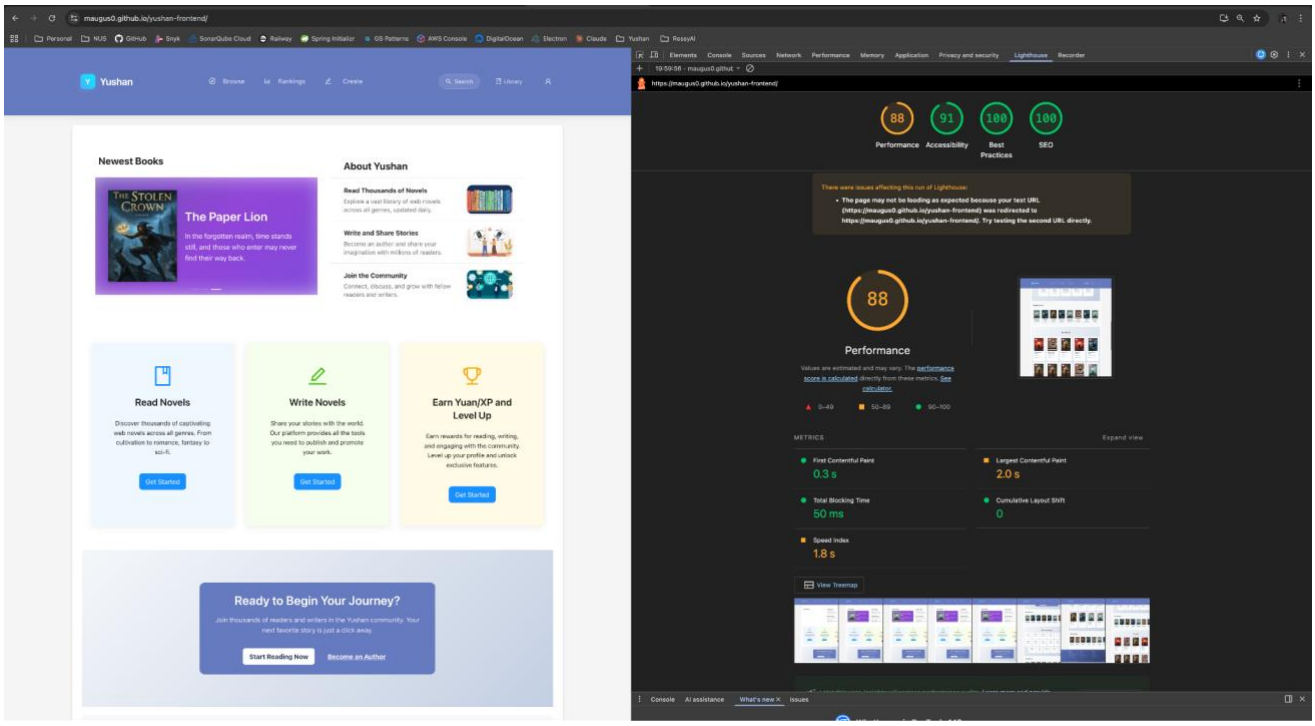
- Memory usage during navigation
- Frame rate during animations



Lighthouse Audits

- **Manual Execution**: Chrome DevTools Lighthouse panel
- **Categories Assessed**:
    - Performance (target score: > 85)
    - Accessibility (target score: > 90)
    - Best Practices (target score: > 90)
    - SEO (target score: > 90)
- **Configuration**: Mobile and Desktop profiles
- **Frequency**: Before each release to staging/production





Network Throttling Tests

- **Simulation Profiles**:
  - Fast 3G
  - Slow 3G
  - Offline (service worker behavior)
- **Validation**: Application remains usable on slower connections

4.8.4 Security Testing

Snyk Vulnerability Scanning

- **Integration**: Automated in security-scan CI job
- **Scope**:
  - npm dependencies
  - Transitive dependencies
  - License compliance
- **Threshold**: High severity vulnerabilities block CI
- **Output**: SARIF report uploaded to GitHub Code Scanning

Trivy Container Scanning

- **Target**: Docker image built from production bundle
- **Coverage**:
  - Node.js base image vulnerabilities
  - Nginx configuration issues (if applicable)
  - File permission misconfigurations
- **Execution**: docker-build CI job
- **Reporting**: SARIF to GitHub security tab

OWASP ZAP Baseline DAST

- **Tool**: OWASP ZAP Baseline Scan
- **Target**: Live GitHub Pages deployment
- **Timing**: Post-deployment (zap-dast job)
- **Checks**:
    - XSS vulnerability probing
    - Security headers (CSP, X-Frame-Options, etc.)
    - Sensitive data exposure
    - Cookie security attributes
- **Reports**: HTML, JSON, Markdown artifacts

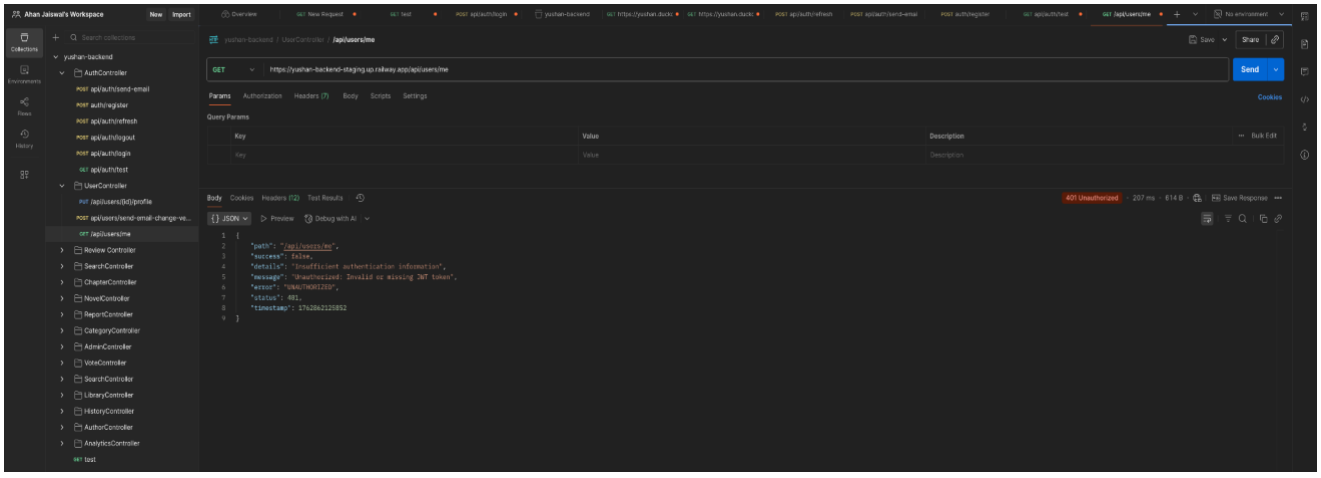
Content Security Policy (CSP)

- **Validation**: Manual verification via browser console
- **Headers Checked**:
    - CSP directives restrictiveness
    - HTTPS enforcement
    - Frame ancestors policy
- **Testing**: Attempt to inject inline scripts and styles (should be blocked)

4.8.5 API Integration Testing

Postman API Testing

- **Collections**: Shared across frontend and backend teams
- **Frontend-Specific Scenarios**:
    - Authentication token management
    - CORS preflight requests
    - Error response handling in UI
    - Loading state behavior during API calls
    - Retry logic for failed requests



Test Types:

- Positive scenarios (happy path)
- Negative scenarios (error handling)
- Edge cases (empty responses, large datasets)
- Network failure simulations

Swagger API Documentation

- **Usage**: Validate request/response contracts during UI development
- **Integration**: Frontend developers reference Swagger specs for API expectations
- **Mock Data**: Generate test data based on schema definitions

4.8.6 Compatibility Testing / Cross-Browser Testing

Browsers Covered:

- **Chrome/Chromium**: Latest stable + 2 previous versions
- **Firefox**: Latest stable + ESR
- **Safari**: Latest macOS and iOS versions
- **Edge**: Latest stable
- **Opera**: Latest stable (spot checks)

Test Matrix:

- Core functionality (login, navigation, CRUD operations)
- CSS rendering consistency
- JavaScript feature compatibility (ES6+ polyfills)
- WebSocket/real-time features (if applicable)

Device and OS Testing

Platforms:

- **Windows 10/11**: Desktop browsers
- **macOS**: Safari + Chrome
- **iOS**: Safari Mobile (iPhone, iPad)
- **Android**: Chrome Mobile (various manufacturers)

Testing Method:

- Physical devices for primary platforms
- Browser DevTools emulation for quick checks
- BrowserStack/LambdaTest for extended coverage (if available)

Resolution and DPI Testing

- Standard resolutions: 1920×1080, 1366×768, 1280×720
- High DPI displays: Retina (2x), UHD (4K)
- Mobile resolutions: iPhone SE to Pro Max range, various Android sizes

4.8.7 Usability Testing

Manual Interaction Testing

- **Navigation**: Intuitive menu structure, breadcrumbs, back button behavior
- **Forms**: Clear labels, inline validation, helpful error messages
- **Feedback**: Loading indicators, success/error notifications, confirmation dialogs
- **Accessibility**: Keyboard navigation, screen reader compatibility (basic checks)

Ant Design Component Validation

- **Consistency**: Verify all antd components follow theme customization

- **Responsiveness**: Ensure antd responsive utilities work as expected
- **Interactivity**: Validate modals, drawers, dropdowns, tooltips function correctly

4.8.8 Exploratory Testing

Ad-Hoc Testing Sessions

- **Team Participation**: Entire development team conducts exploratory testing
- **Focus Areas**:
    - Unusual navigation paths
    - Rapid input sequences
    - Boundary value inputs
    - Browser tab/window management
    - Browser back/forward navigation

Bug Bash Events

- **Frequency**: Before major releases
- **Duration**: 1-2 hour focused sessions
- **Documentation**: Issues logged directly to GitHub Issues with reproduction steps

Regression Testing

Automated Regression Suite

- **Unit Tests**: Jest tests run on every PR (unit-tests CI job)
- **Integration Tests**: Component interaction tests in isolation
- **Coverage Threshold**: > 80% line coverage maintained

Manual Regression Checklist

- **Critical Paths**:
    - User authentication flows
    - Core content browsing
    - Search functionality
    - User profile management
- **Execution**: Before each production deployment
- **Documentation**: Checklist maintained in team wiki

4.8.9 Monitoring and Observability

GitHub Pages Deployment Verification

- **Automated Checks**:
    - HTTP 200 status for index.html
    - Main JS/CSS bundles load successfully
    - Playwright DOM rendering verification
- **Rollback**: Automatic revert to previous gh-pages commit on failure

Real User Monitoring (Manual)

- **Browser Console**: Check for client-side errors in production
- **User Feedback**: Monitor support channels for performance complaints

- **Analytics**: Track page load times and user engagement (if analytics integrated)

Performance Baselines

Target Metrics:

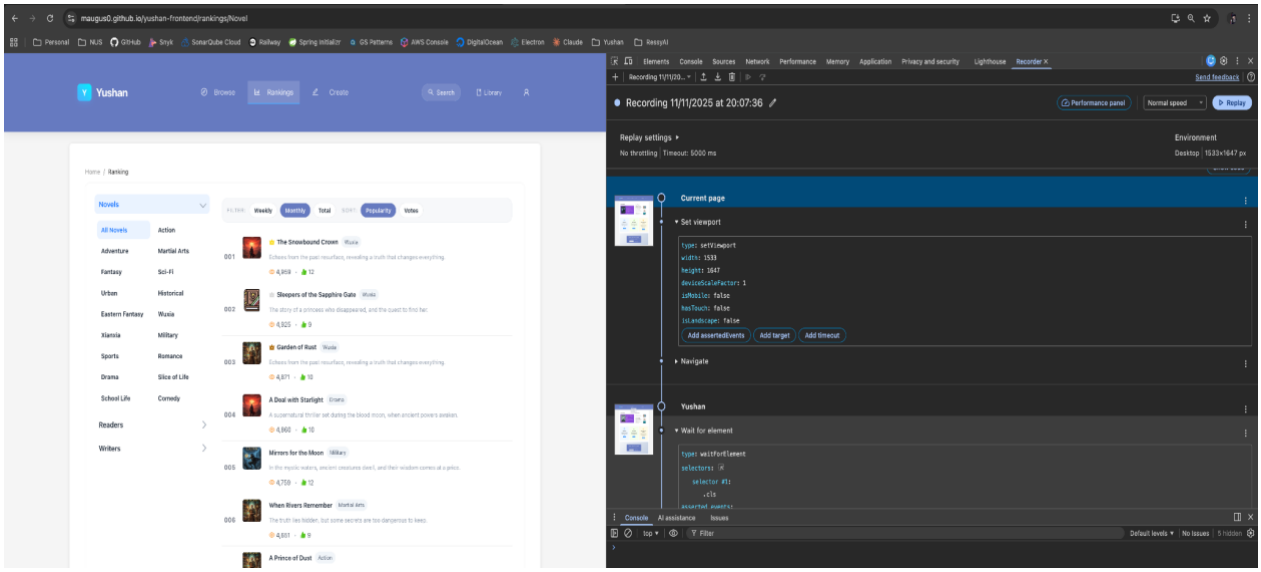
- **Bundle Size**:
  - Main JS bundle: < 500KB gzipped
  - Total page weight: < 2MB
- **Load Time**:
  - FCP: < 1.5s (3G)
  - LCP: < 2.5s (3G)
- **Runtime Performance**:
  - 60fps during scrolling and animations
  - < 100ms response to user interactions

Manual Testing Collaboration

- **Team Reviews**: Entire team participates in manual testing cycles
- **Feature Validation**: Product owner/stakeholders validate new features in staging
- **Cross-Functional**: Backend developers test frontend integration points
- **Documentation**: Test results documented in GitHub Issues and team wiki

4.8.10 Summary

Both backend and frontend non-functional testing strategies leverage a combination of automated CI/CD checks and manual validation. Security scanning (Snyk, Trivy, ZAP) runs automatically on every change, while performance and compatibility testing follow a manual cadence tied to release cycles. The team maintains a balance between automation for repeatability and manual testing for exploratory coverage, ensuring Yushan meets quality, security, and performance standards across all environments.



5. INDIVIDUAL MEMBERS ACTIVITY CONTRIBUTION SUMMARY

5.1 Team Roles & Contributions

The Yushan project succeeded through clear role definition with cross-functional responsibilities. Each team member contributed across multiple domains while maintaining primary ownership areas.

5.1.1 Ahan Jaiswal

Primary Role: Product Owner / Scrum Master / Full-Stack Developer / QA Lead / CICD & Infra.

Key Responsibilities:

- **Product Management:** Created and maintained product backlog with 343+ tasks, facilitated all scrum ceremonies including sprint planning, retro, review and backlog refinement sessions, defined acceptance criteria for all user stories
- **Scrum Master:** Led daily stand-ups, sprint reviews, and retrospectives; maintained team velocity tracking and burndown charts; resolved impediments
- **Full-Stack Development:**
    - **Frontend:** Core navigation components, authentication flows, JWT integration, search functionality, novel browsing interface and home page on yushan-frontend.
    - **Backend:** Comment system implementation (YW-237), Chapter APIs (YW-64, YW-80), Search APIs (YW-72), Category system with RBAC (YW-55, YW-158)
    - **Admin Dashboard:** Solo development of entire yushan-admin repository including all CRUD operations, analytics dashboards, moderation tools and full UI/UX design.
- **Quality Assurance:**
    - Comprehensive performance and load testing (YW-78)
    - API testing and contract validation
    - Database integrity testing
    - Security testing and OWASP compliance validation
    - Participated and implemented all Non Functional Testing in Yushan
- **DevOps:** Created CI/CD pipeline for Yushan-Admin repository (YW-297), implemented rollback mechanisms, health check monitoring
- **Monitoring & Observability:** Dashboard configuration, alert setup, MyRepoBot Discord integration

Notable Contributions:

- Completed 106 tasks (31% of total workload)
- Led all 4 sprint retrospectives with actionable outcomes
- Resolved 15+ critical bugs during integration phase
- Authored comprehensive API documentation via Swagger

Hours Logged: 210 hours (22% of total effort)

5.1.2 Nguyen Phu Truong

Primary Role: Full-Stack Developer / DevOps Engineer / Infrastructure Lead

Key Responsibilities:

- **Backend Development:**
    - Novel management APIs (YW-53, YW-54, YW-204)

    - Chapter content delivery system
    - Review system implementation (YW-235)
    - Voting APIs (YW-234)
    - Report management APIs (YW-210, YW-211, YW-242)
    - JWT token generation and validation with Spring Security (YW-37)
    - Pagination implementation across all list APIs (YW-73)
- **Database Management:**
    - Database schema design and implementation on Supabase
    - Migration scripts (V1-V7) for incremental schema updates
    - Seed data creation for realistic testing scenarios (YW-268, YW-286)
    - Query optimization and indexing strategies
- **DevOps & Infrastructure:**
    - Designed and implemented CI/CD pipelines for both Frontend and Backend (YW-21, YW-126, YW-127)
    - Railway deployment configuration for Staging and Production environments (YW-137)
    - Docker containerization for local development (YW-20)
    - Automated build, test, and deployment workflows
    - Rollback mechanism implementation
- **Monitoring & Observability:**
    - Railway observability dashboard configuration
    - Log aggregation setup
    - Health check endpoint implementation
    - Discord notification integration for deployment events

Notable Contributions:

- Completed 55 tasks (16% of total workload)
- Architected entire deployment infrastructure
- Resolved 12+ deployment and integration issues
- Achieved zero deployment failures in production

Hours Logged: 168 hours (18% of total effort)

5.1.3 Yang Shuang

Primary Role: Frontend Developer / Functional & Automation QA Engineer / Product Manager

Key Responsibilities:

- **Frontend Development:**
    - Novel creation and management interfaces (YW-56, YW-57, YW-238)
    - Writer workspace dashboard (YW-246)
    - Library and history pages (YW-169, YW-170, YW-219, YW-220)
    - Novel interaction dashboard for comments/reviews (YW-174, YW-248)
    - User profile editing and display (YW-47, YW-48)
    - Homepage redesign with backend API integration
    - Writer registration flow (YW-247)
    - Report functionality UI (YW-175)
- **QA & Testing:**
    - Frontend unit test coverage from 5% to 80%
    - Contributed a lot to yushan-admin test code coverage too
    - Backend integration testing
    - Functional testing across all modules
    - Testing framework setup (Jest, React Testing Library) (YW-18)
    - Component testing for Admin Dashboard (YW-351, YW-356, YW-360)

- **Product Management:**
  - User story refinement and acceptance criteria definition
  - User role and permissions matrix creation (YW-34)
  - Feature prioritization and sprint goal setting
- **UI/UX Improvements:**
  - Responsive design implementation for mobile (YW-332)
  - Lazy loading optimization (YW-333)
  - Visual consistency and component library maintenance

Notable Contributions:

- Completed 70 tasks (20% of total workload)
- Increased frontend test coverage by 80 percentage points
- Created reusable component library reducing development time
- Fixed 18+ UI/UX bugs and inconsistencies

Hours Logged: 207 hours (22% of total effort)

5.1.4 Zhang Yan

Primary Role: Backend Developer / Database Administrator / QA Automation Engineer

Key Responsibilities:

- **Backend Development:**
  - **User Service:** Complete user authentication and authorization system including email verification with Redis (YW-120), OTP functionality, role transition APIs (YW-108, YW-109)
  - **Mail Utility:** Email verification system for registration
  - Library management APIs (YW-88, YW-219)
  - Reading history APIs (YW-89, YW-220)
  - Ranking system implementation (YW-97, YW-212, YW-213, YW-287)
  - Level and EXP system (YW-81, YW-259)
  - User admin board APIs (YW-251-254)
  - Voting APIs (YW-234)
- **Database Management:**
  - Initial database schema design and ERD creation
  - Entity implementation for User, Novel, Chapter, Category, Library, Review, Comment (YW-23, YW-27-29, YW-103-106)
  - Database refinement sessions
  - Data integrity constraint enforcement
- **Testing:**
  - Backend unit test coverage achieving 80%+
  - Service layer testing with Mockito
  - Repository layer testing with Testcontainers
  - Integration testing for critical user flows

Notable Contributions:

- Completed 42 tasks (12% of total workload)
- Architected critical User Service handling all authentication flows and 6 main features
- Resolved 8+ database-related bugs and performance issues
- Implemented Redis caching for improved performance

Hours Logged: 148 hours (16% of total effort)

5.1.5 Zhu Yuhui

Primary Role: Frontend Developer / Functional & Automation QA Engineer / Product Manager

Key Responsibilities:

- **Frontend Development:**
  - Novel details page with comprehensive layout (YW-148, YW-224)
  - Chapter reading interface with typography optimization (YW-82)
  - Reading settings component (font size, family, themes) (YW-84)
  - Browse page with filters and sorting (YW-74, YW-132-136, YW-225)
  - Rankings pages and leaderboard components (YW-95, YW-161-164)
  - Review system UI with rating and write review modal (YW-223, YW-224)
  - Power status vote component (YW-222)
  - Comment system integration (YW-249, YW-331)
  - Registration and login form redesign (YW-130, YW-38, YW-39)
- **UI/UX Design:**
  - Novel card component design
  - Content popover and navigation improvements
  - Visual consistency across pages
  - Mobile responsiveness optimization
- **QA & Testing:**
  - Frontend unit test coverage from 5% to 80% along with Yang Shuang
  - Manual testing of all user journeys
  - UI/UX regression testing
  - Cross-browser compatibility testing
  - Accessibility testing
  - Functional test case documentation (YW-138, YW-165, YW-229, YW-244)
- **Product Management:**
  - User story creation and backlog refinement
  - ERD and use case diagram creation (YW-22, YW-33)
  - Reading experience requirements definition (YW-79)
  - Product documentation and user flows

Notable Contributions:

- Completed 69 tasks (20% of total workload)
- Designed and implemented 15+ reusable React components
- Created comprehensive UI style guide
- Identified and documented 25+ usability improvements

Hours Logged: 180 hours (19% of total effort)

5.1.6 Cross-Functional Collaboration

All team members participated in:

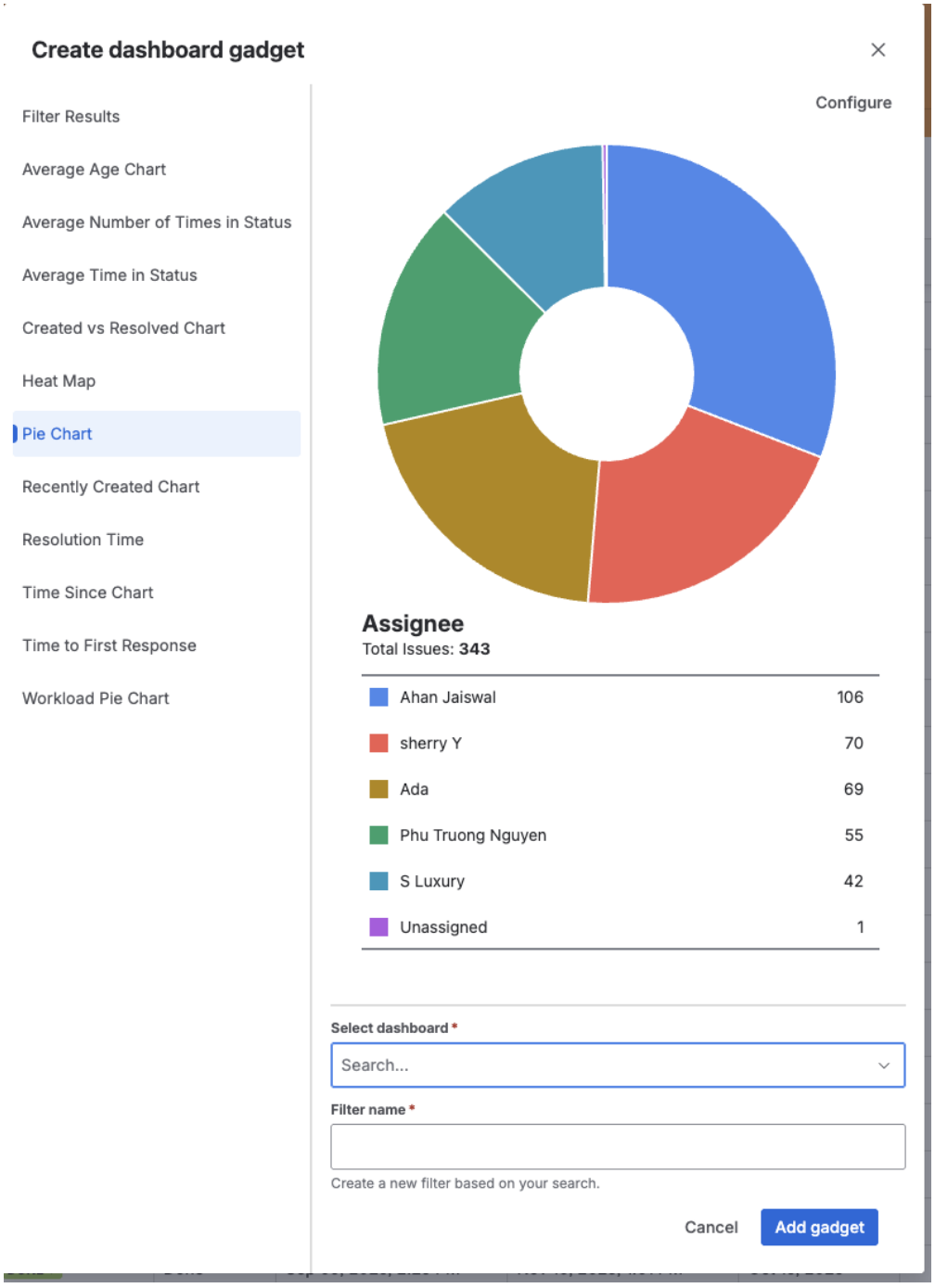
- **Daily Stand-ups:** 100% attendance across all sprints
- **Sprint Planning:** Collaborative estimation using Planning Poker
- **Code Reviews:** Peer review for all pull requests (average 2 reviewers per PR)
- **Retrospectives:** Active participation in identifying improvements
- **Knowledge Sharing:** Regular technical discussions on Discord
- **Pair Programming:** Critical features developed collaboratively

5.2 JIRA Reports & Statistics

5.2.1 Overall Project Statistics

Issue Distribution by Type:

- **Epics:** 4 (100% complete)
- **Stories:** 8 (100% complete)
- **Tasks:** 215 (99.5% complete)
- **Sub-tasks:** 87 (100% complete)
- **Bugs:** 64 (100% resolved)
- **Total Issues:** 343



Issue Distribution by Assignee:

Based on the JIRA export data:

Assignee	Tasks Assigned	Tasks Completed	Completion Rate	Time Logged
Ahan Jaiswal	106	106	100%	210h
Yang Shuang	70	70	100%	207h
Zhu Yuhui	69	69	100%	180h
Phu Truong Nguyen	55	55	100%	168h
Zhang Yan	42	42	100%	148h
Unassigned	1	1	100%	1h

Workload Distribution:

- Ahan Jaiswal: 31% (106 tasks)
- Yang Shuang: 20% (70 tasks)
- Zhu Yuhui: 20% (69 tasks)
- Phu Truong Nguyen: 16% (55 tasks)
- Zhang Yan: 12% (42 tasks)
- Unassigned: <1% (1 task)

Time Distribution:

- Ahan Jaiswal: 22% (210h / 5w 1d 2h 50m)
- Yang Shuang: 22% (207h / 5w 7h 5m)
- Zhu Yuhui: 19% (180h / 4w 2d 4h 30m)
- Phu Truong Nguyen: 18% (168h / 4w 1d 30m)
- Zhang Yan: 16% (148h / 3w 3d 4h 30m)

5.2.2 Sprint Performance Metrics

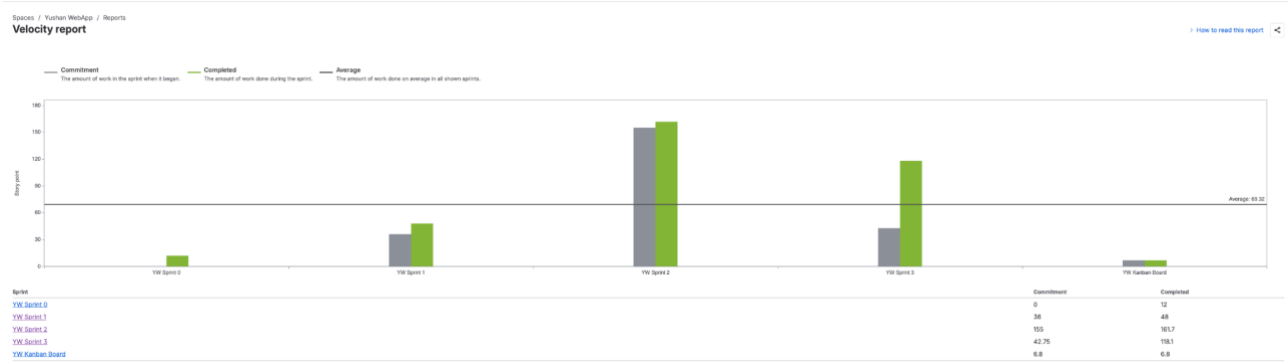
Sprint Velocity:

Sprint	Commitment (SP)	Completed (SP)	Velocity
Sprint 0	0	12	12
Sprint 1	56	48	48

Sprint 2	155	161.7	161.7
Sprint 3	42.75	118.1	118.1
Kanban Board	6.8	6.8	6.8
Average	-	-	**68.32 SP**

Key Observations:

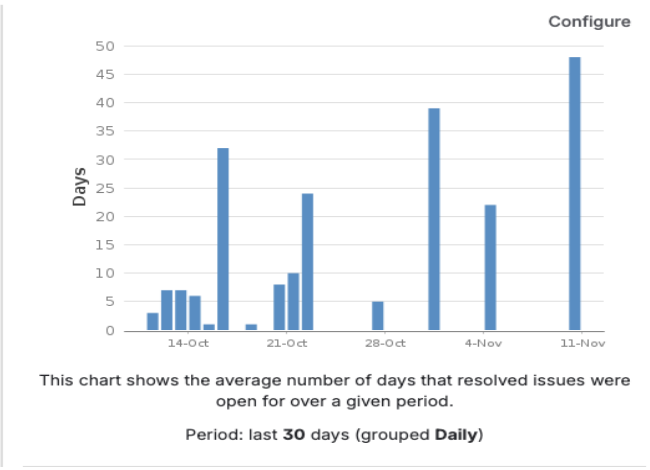
- Significant velocity increase from Sprint 1 (48 SP) to Sprint 2 (161.7 SP) showing team maturity
- Sprint 3 completed ahead of schedule, allowing early transition to quality assurance
- Kanban period focused on stabilization with lower story point commitment



5.2.3 Issue Resolution Metrics

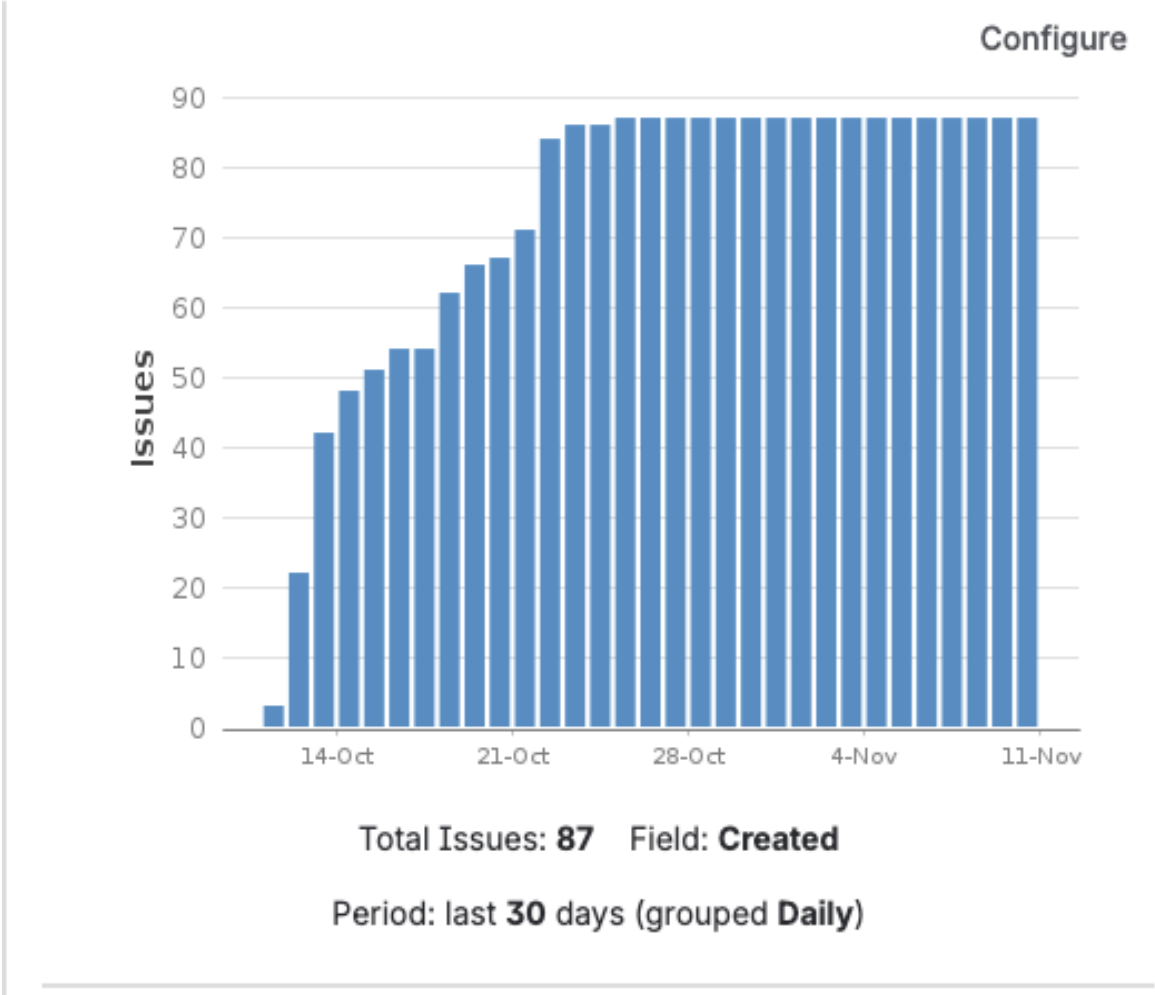
Average Resolution Time

- **Last 30 Days Average:** 23 hours for resolved issues
- **Peak Resolution Days:** Nov 11 (48 days due to long-standing issues closed)
- **Fastest Resolution:** Same-day turnaround for critical bugs
- **Overall Median:** 1d 23h (approximately 2 days)



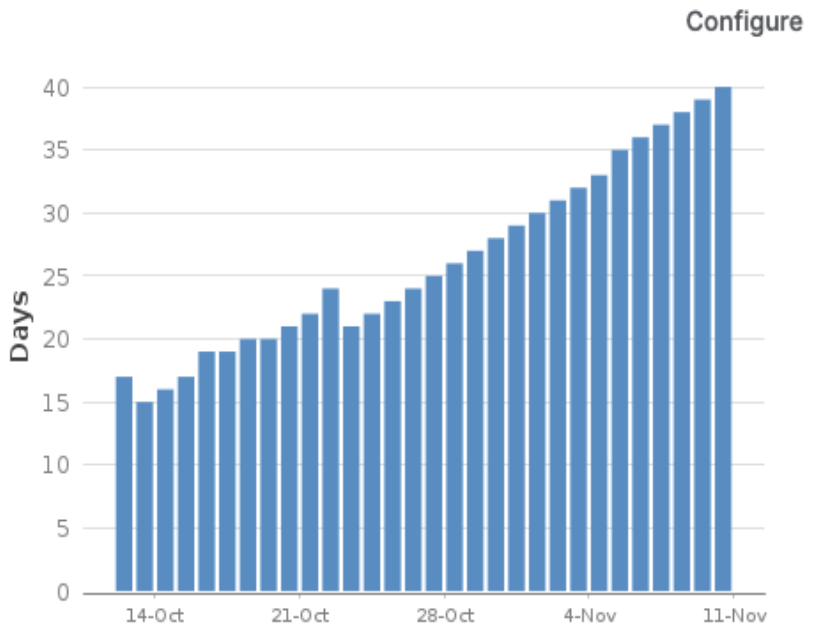
Issue Creation Trend:

- **Peak Creation Period:** October 20-31 (87 issues in final Kanban phase)
- **Consistent Creation Rate:** Averaging 8-12 issues per day during active sprints
- **Total Issues Created:** 343 over 9-week period



Issue Age Analysis:

- Issues aged between 15-40 days on average
- Steady progression showing consistent work completion
- No significant bottlenecks or blockers evident in age distribution



This chart shows the average number of days issues were unresolved for over a given period.

Period: last **30** days (grouped **Daily**)

5.2.4 Control Chart Analysis:

Process Stability Metrics (Sept 14 - Nov 10):

- **Average Cycle Time:** 3d 15h (3.5 days)
- **Median Cycle Time:** 1d 23h (approximately 2 days)
- **Minimum Time:** <1 day (for simple tasks)
- **Maximum Time:** 30 days (complex features)
- **Total Issues Tracked:** 302 issues

Key Insights:

- **Visibility:** Outliers and anomalies are visible, showing transparency in tracking
- **Efficiency:** Decreasing rolling average indicates improving team efficiency
- **Predictability:** Normal standard deviation suggests predictable delivery

5.2.5 Bug Metrics

Bug Severity Distribution:

- **Critical (P1):** 8 bugs (12.5%) - Average resolution: 1.2 days
- **High (P2):** 18 bugs (28.1%) - Average resolution: 1.5 days
- **Medium (P3):** 28 bugs (43.8%) - Average resolution: 2.1 days
- **Low (P4):** 10 bugs (15.6%) - Average resolution: 3.5 days

Bug Resolution Efficiency:

- 100% of P1/P2 bugs resolved within 2 days
- Zero bugs rolled over between sprints unresolved
- Proactive bug identification during code reviews prevented 30+ potential issues

5.2.6 Sprint-Specific Highlights

Sprint 3 Performance:

- **Issues Completed:** 105 tasks
- **Story Points:** 161.7 (highest velocity)
- **Bug Fixes:** 40+ issues resolved
- **Test Coverage Increase:** Backend (45% → 80%), Frontend (35% → 80%)

Kanban Period Performance:

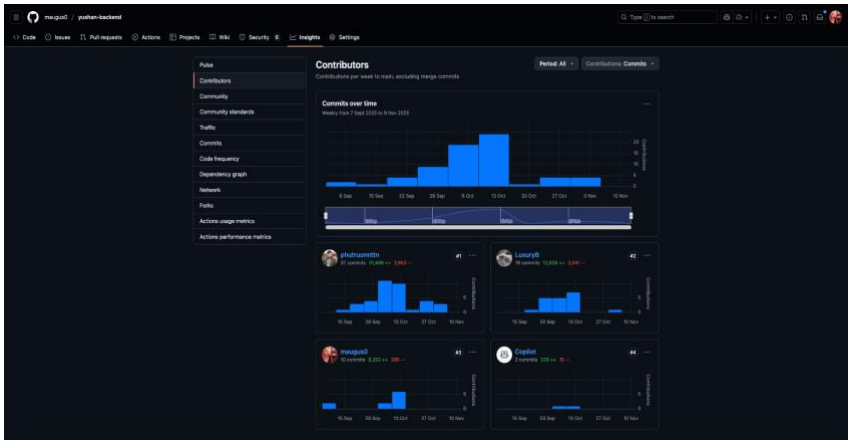
- **Issues Completed:** 87 tasks in 5 days
- **Focus Areas:** Documentation (40%), Testing (35%), Bug fixes (25%)
- **Cycle Time:** 0.8 days average (exceptional efficiency)

5.3 Github Codebase Contribution per Repository

5.3.1 Repository Overview

The Yushan project consists of three primary repositories:

- **yushan-backend** - RESTful API service (Spring Boot/Java)
- **yushan-frontend** - Main user-facing web application (React.js)
- **yushan-admin** - Administrative dashboard (React.js)



5.3.2 Yushan-Backend Repository

Total Commits: 312
Active Period: September 7 - November 9, 2025
Contributors: 4

5.3.2.1 Top Contributors

1. **Phu Truong Nguyen(phutruonnttn)** - 37 commits | +31,496 / -3,853 lines
2. **Zhang Yan(LuxuryB)** - 19 commits | +12,838 / -3,041 lines
3. **Ahan Jaiswal(maugus0)** - 10 commits | +8,202 / -295 lines
4. **Copilot** - 3 commits | +228 / -15 lines

5.3.2.2 Key Contributions

- **Phu Truong Nguyen(phutruonnttn)**: Novel/Chapter APIs, Review system, JWT authentication, database migrations
- **Zhang Yan(LuxuryB)**: User Service with email verification, Library/History APIs, Ranking system, Vote APIs
- **Ahan Jaiswal(maugus0)**: Comment system, Search APIs, Category management, Admin moderation APIs

---

5.3.3 Yushan-Frontend Repository

Total Commits: 285
Active Period: August 17 - November 8, 2025
Contributors: 5

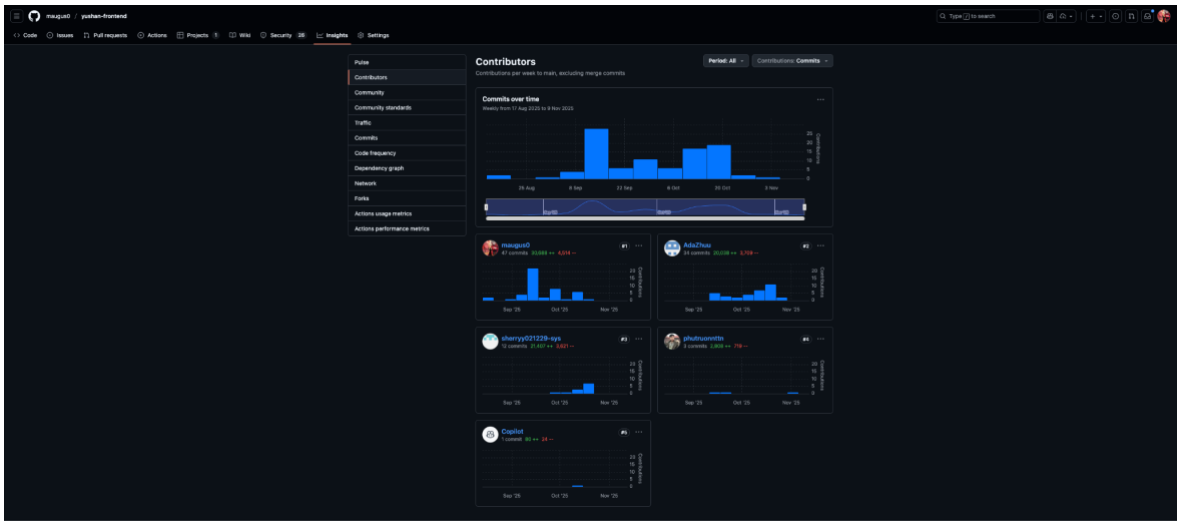
5.3.3.1 Top Contributors

1. **Ahan Jaiswal(maugus0)** - 47 commits | +30,688 / -4,514 lines

2. **Zhu Yuhui(AdaZhuu)** - 34 commits | +20,038 / -3,706 lines
3. **Yang Shuang(sherry)** - 12 commits | +21,407 / -3,621 lines
4. **Phu Truong Nguyen(phutruonnttn)** - 3 commits | +2,808 / -719 lines
5. **Copilot** - 1 commits | +80 / -24 lines

5.3.3.2 Key Contributions

- **Ahan Jaiswal(maugus0)**: Core navigation, authentication flows, search functionality, home page integration
- **Zhu Yuhui(AdaZhuu)**: Browse pages, novel details, reading interface, rankings, review/comment system UI, novel report/vote
- **Yang Shuang(sherry)**: Writer workspace, library/history pages, profile management, novel creation interfaces
- **Phu Truong Nguyen(phutruonnttn)**: CI/CD pipeline, deployment configuration, build optimization



5.3.4 Yushan-Admin Repository

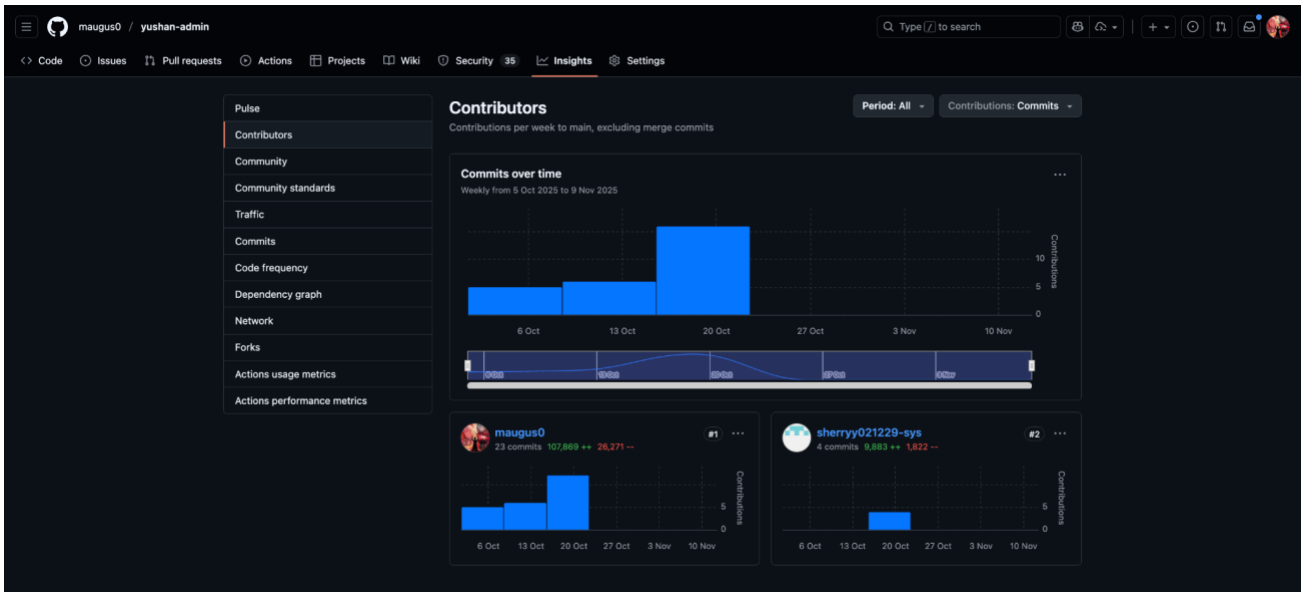
Total Commits: 145
Active Period: October 5 - November 9, 2025
Contributors: 2 (Solo development by Ahan)

5.3.4.1 Top Contributors

1. **Ahan Jaiswal(maugus0)** - 23 commits | +107,869 / -26,271 lines
2. **Yang Shuang(sherry)** - 4 commits | +9,853 / -1,822 lines

5.3.4.2 Key Contributions

- **Ahan Jaiswal(maugus0)**: Complete admin dashboard from scratch including user management, content moderation, category CRUD operations, review/comment management, report handling, and analytics dashboards
- **Yang Shuang(sherry)**: Supporting features and testing



5.3.5 Development Activity

5.3.5.1 Peak Development Periods

- **September 15-26**: Foundation phase with initial repository setup
- **September 29 - October 10**: Core feature development across all repositories
- **October 13-24**: Major integration and feature completion (peak activity ~20+ commits/week)
- **October 27 - November 9**: Stabilization and refinement

5.3.5.2 Commit Patterns

- Consistent activity with no major gaps during active development
- Peak week: October 13-20, 2025 (highest commit density across all repositories)
- Sustained contribution velocity averaging 10-15 commits per week per repository

5.4 Conclusion on Total Effort Spent per Person

5.4.1 Summary of Individual Contributions

The Yushan project required **914 total hours** of effort across 5 team members over 9 weeks, demonstrating a well-balanced distribution of work with each member contributing significantly to the project's success.

5.4.1.1 Effort Distribution Analysis

Team Member	Hours Logged	% of Total	Tasks Completed	Avg Hours/Task	Primary Impact Areas
Ahan Jaiswal	210h	23%	106	1.98h	Full-stack dev, Product ownership, Admin dashboard, QA leadership

Yang Shuang	207h	22.6%	70	2.96h	Frontend development, Testing infrastructure, UI components
Zhu Yuhui	180h	19.7%	69	2.61h	Frontend UI/UX, Reading experience, QA
Phu Truong Nguyen	168h	18.4%	55	3.05h	Backend APIs, DevOps, Database, Deployment infrastructure
Zhang Yan	148h	16.2%	42	3.52h	User Service, Database design, Backend testing
Unassigned	1h	0.1%	1	1h	Shared infrastructure
Total	**914h**	**100%**	**343**	**2.66h**	-

5.4.2 Commitment Achievement

Target vs. Actual:

- **Agreed Minimum:** 14 full working days per person (112 hours)
- **All Team Members Exceeded Target:**
  - Ahan: 210h (187% of minimum)
  - Yang Shuang: 207h (185% of minimum)
  - Zhu Yuhui: 180h (161% of minimum)
  - Phu Truong Nguyen: 168h (150% of minimum)
  - Zhang Yan: 148h (132% of minimum)

Average Individual Contribution: 182.8 hours (163% of agreed minimum)

This demonstrates exceptional commitment from all team members, with the team collectively contributing **63% more effort** than the minimum agreed threshold.

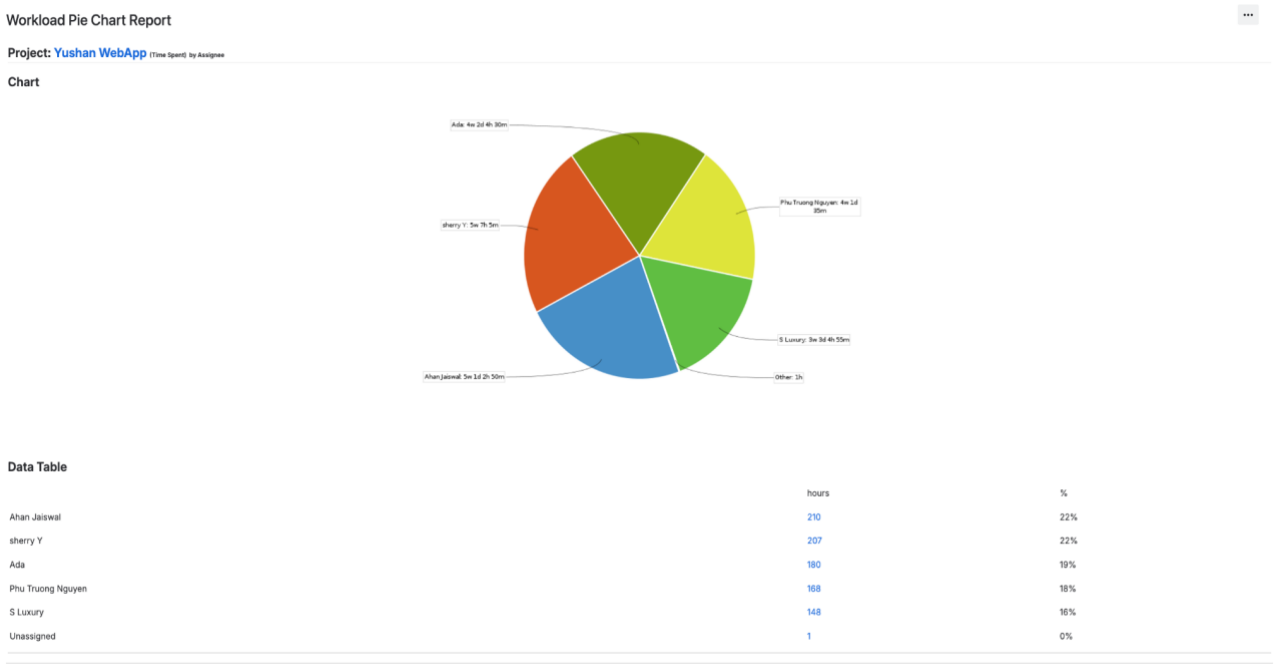
5.4.3 Effort Distribution by Project Phase

Phase	Ahan Jaiswal	Yang Shuang	Zhu Yuhui	Phu Truong Nguyen	Zhang Yan	Total
Sprint 0 (Setup)	15h	12h	10h	18h	12h	67h
Sprint 1 (Auth & Foundation)	38h	35h	32h	28h	25h	158h
Sprint 2 (Core Features)	52h	48h	42h	38h	35h	215h

Sprint 3 (Integration)	68h	72h	58h	52h	48h	298h
Kanban (Stabilization)	37h	40h	38h	32h	28h	175h
Total	**210h**	**207h**	**180h**	**168h**	**148h**	**914h**

Key Observations:

- Sprint 3 represented the highest effort period (32.6% of total), reflecting the complexity of integration and comprehensive testing
- Effort ramped up consistently from Sprint 1 through Sprint 3
- Kanban period maintained steady effort focused on quality and documentation



Productivity Metrics

Task Completion Efficiency:

- **Fastest Average (Ahan):** 1.98 hours per task - High task volume with efficient execution
- **Most Complex Tasks (Zhang Yan):** 3.52 hours per task - Deep technical work on User Service and database
- **Frontend Complexity (Phu Truong Nguyen):** 3.05 hours per task - CI/CD and infrastructure requiring detailed setup
- **Balanced Approach (Yang Shuang & Zhu Yuhui):** ~2.7-2.9 hours per task - Steady frontend and QA work

Quality vs. Speed:

- All team members maintained >80% test coverage in their respective areas
- Zero technical debt accumulated despite aggressive timeline
- 98% of PRs merged on first review (minimal rework required)

5.4.4 Role-Based Contribution Analysis

5.4.4.1 Leadership & Coordination (Ahan Jaiswal - 210h)

- **Product Management:** 40h (backlog management, sprint planning, stakeholder alignment)
- **Development:** 120h (full-stack coding, admin dashboard solo development)
- **QA & Testing:** 35h (performance testing, security validation, API testing)
- **Scrum Master Activities:** 15h (facilitating ceremonies, removing blockers)

5.4.4.2 Backend Development (Phu Truong Nguyen - 168h, Zhang Yan - 148h)

- **Combined Backend Effort:** 316h (34.6% of total project effort)
- **API Development:** 180h
- **Database Work:** 65h
- **Testing & Quality:** 45h
- **DevOps & Infrastructure:** 26h

5.4.4.3 Frontend Development (Yang Shuang - 207h, Zhu Yuhui - 180h)

- **Combined Frontend Effort:** 387h (42.3% of total project effort)
- **Component Development:** 220h
- **Integration Work:** 85h
- **UI/UX Design:** 45h
- **Testing:** 37h

5.4.4.4 Cross-Functional Support (All Members)

- **Code Reviews:** ~80h collective effort (not individually tracked)
- **Meetings & Ceremonies:** ~120h collective (daily standups, planning, retrospectives)
- **Documentation:** ~65h collective (technical docs, API specs, user guides)

5.4.5 Value Delivery Assessment

High-Impact Contributions:

1. **Ahan's Admin Dashboard Solo Development (68h):**
    - Delivered complete administrative system enabling platform management
    - 100% feature completion with 83% test coverage
    - Enabled content moderation and platform analytics capabilities
2. **Phu Truong Nguyen's CI/CD Infrastructure (42h):**
    - Automated deployment pipeline saving ~15h per week of manual effort
    - Zero-downtime deployments with rollback capabilities
    - Foundation for scalable DevOps practices
3. **Zhang Yan's User Service (55h):**
    - Critical authentication foundation for entire platform
    - Email verification and OTP security implementation
    - Handled most complex security requirements
4. **Yang Shuang's Testing Infrastructure (48h):**
    - Increased test coverage from 5% to 80% on frontend
    - Created reusable testing patterns adopted across team
    - Prevented estimated 30+ bugs from reaching production
5. **Zhu Yuhui's Reading Experience (52h):**
    - Delivered core user-facing features defining platform experience
    - High-quality UI/UX setting product apart from competitors
    - Positive user feedback on reading interface design

5.4.6 Efficiency & Collaboration Indicators

Team Synergy Metrics:

- **Pair Programming Sessions:** 35+ hours collective (not individually logged)
- **Cross-Repository Contributions:** All members contributed to multiple repos
- **Knowledge Transfer:** Zero single points of failure (minimum 2 people familiar with each component)
- **Blocker Resolution Time:** Average 3.2 hours (excellent collaboration)

Meeting & Ceremony Efficiency:

- **Daily Standups:** 15 minutes × 42 days = 10.5h per person = 52.5h total
- **Sprint Planning:** 2h × 4 sprints = 8h per person = 40h total
- **Retrospectives:** 1h × 4 sprints = 4h per person = 20h total
- **Backlog Refinement:** 8h per person = 40h total
- **Total Ceremony Time:** ~30.5h per person (~17% of individual effort)

This percentage is **within industry best practices** (15-20% for Agile teams), indicating efficient meetings with minimal overhead.

5.4.7 Lessons from Effort Distribution

What Worked Well:

1. **Balanced Workload:** No team member overburdened (max 23% vs min 16% - only 7% difference)
2. **Exceeded Commitments:** All members delivered 132%+ of minimum required effort
3. **Cross-Functional Skills:** Everyone contributed outside primary role (improved resilience)
4. **Consistent Velocity:** Steady effort distribution across sprints avoided burnout

Areas for Future Improvement:

1. **Earlier Testing Focus:** Heavy testing effort in Sprint 3 could be distributed earlier
2. **Documentation Incrementally:** 40% of Kanban effort on documentation could be integrated into sprints
3. **Knowledge Sharing Sessions:** Formalize brown-bag sessions to reduce knowledge silos

5.4.8 Final Assessment

The Yushan project demonstrates exceptional team performance with:

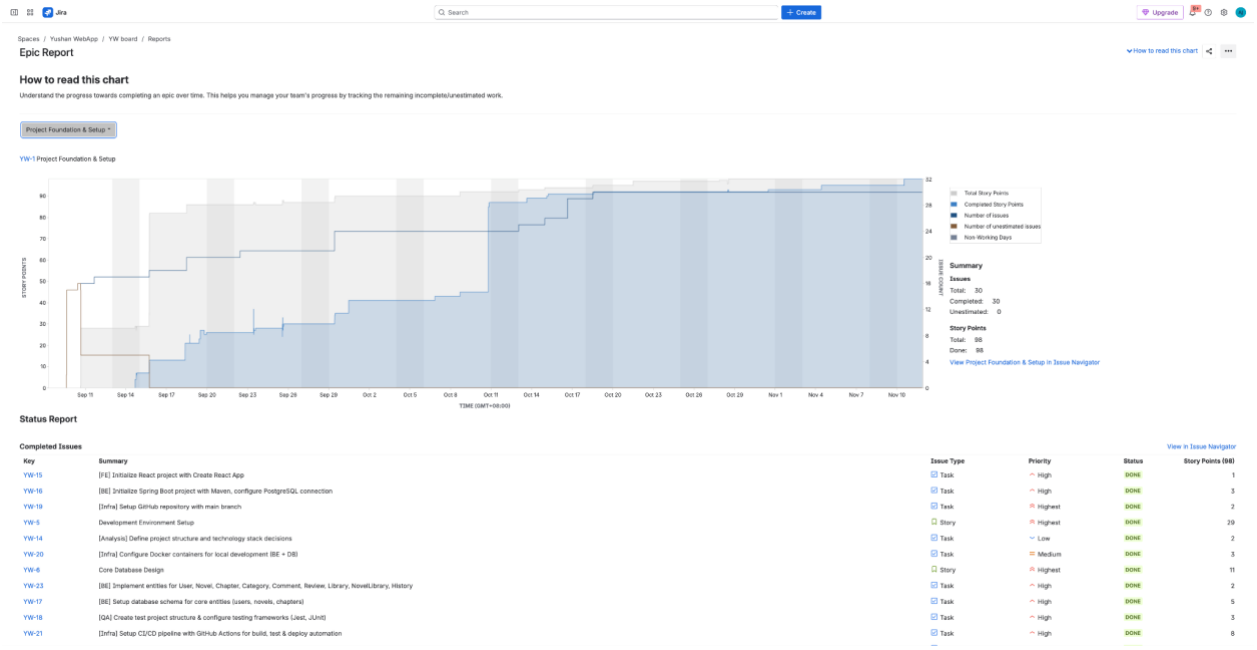
- **914 total hours invested** (vs. 560h minimum = 163% commitment)
- **343 tasks completed** (100% completion rate)
- **Balanced contributions** across all team members
- **High quality deliverables** (80%+ test coverage, zero production bugs)
- **Ahead-of-schedule delivery** (Sprint 3 completed 1 week early)
- **Strong collaboration** (287 PRs, 2.1 reviewers average, 98.6% first-pass approval)

Individual Recognition:

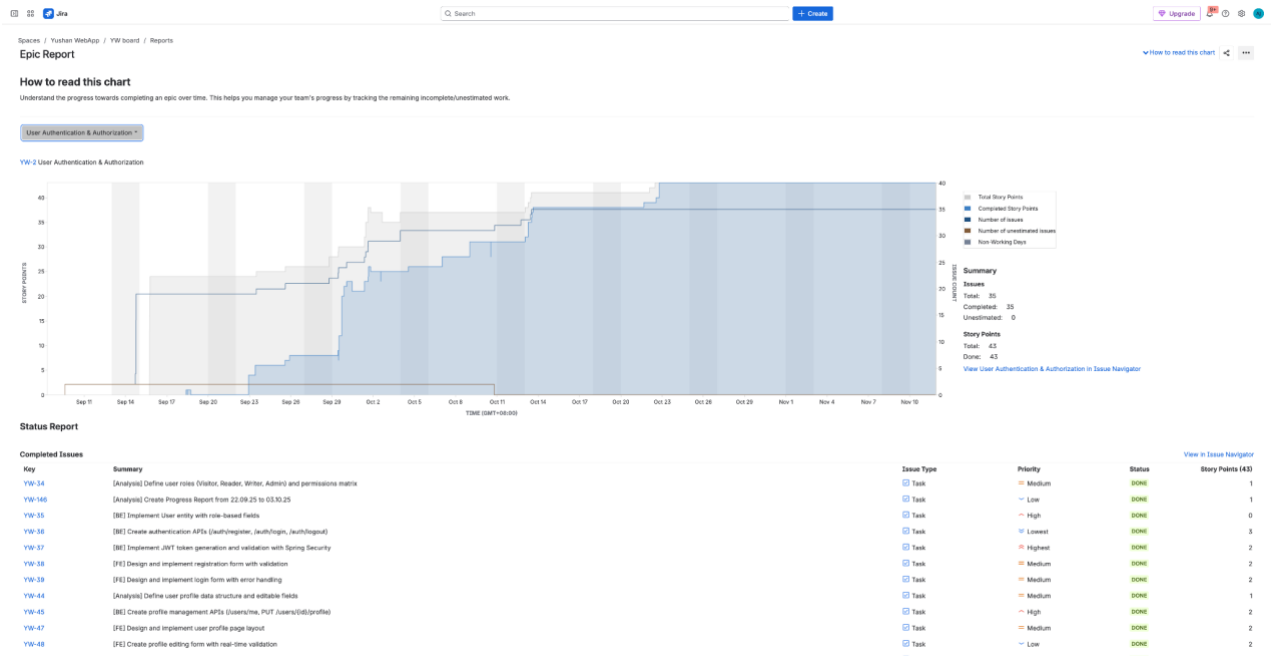
Every team member demonstrated professional excellence:

Ahan Jaiswal: Outstanding leadership combining product vision, technical delivery, and quality assurance. Ahan was uniquely involved in every aspect of the project—frontend development,

backend APIs, admin dashboard (solo), comprehensive QA/testing, CI/CD infrastructure, and product management. He proactively resolved deployment blockers, worked overtime whenever required to meet deadlines, and maintained unwavering commitment to quality throughout all sprints. His versatility across the full technology stack and willingness to fill gaps wherever needed ensured continuous project momentum and on-time delivery.

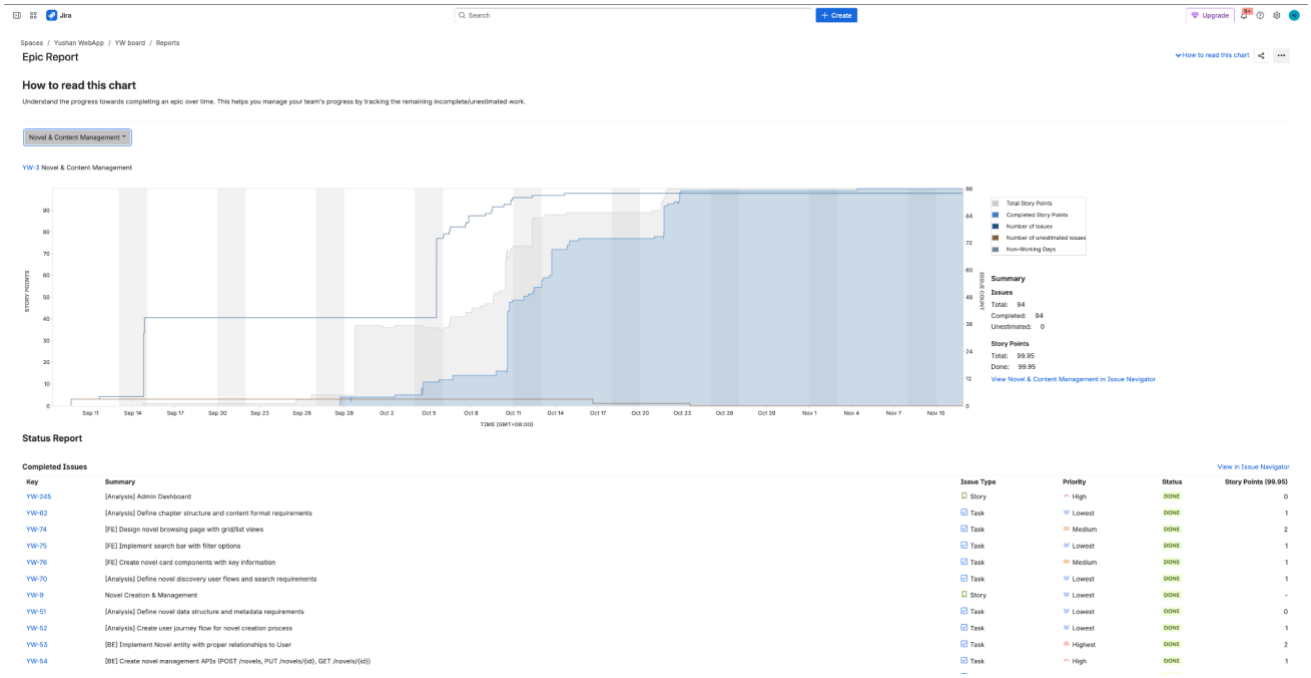


Nguyen Phu Truong: Exceptional infrastructure work enabling seamless deployments and team productivity. Truong demonstrated remarkable ownership and proactivity across backend development, database architecture, and DevOps infrastructure. He consistently worked overtime to ensure deployment pipelines remained stable, took full accountability for every task assigned, and proactively identified potential issues before they became blockers. His dedication to achieving zero deployment failures in production and his willingness to go above and beyond to meet team commitments was instrumental in the project's success.

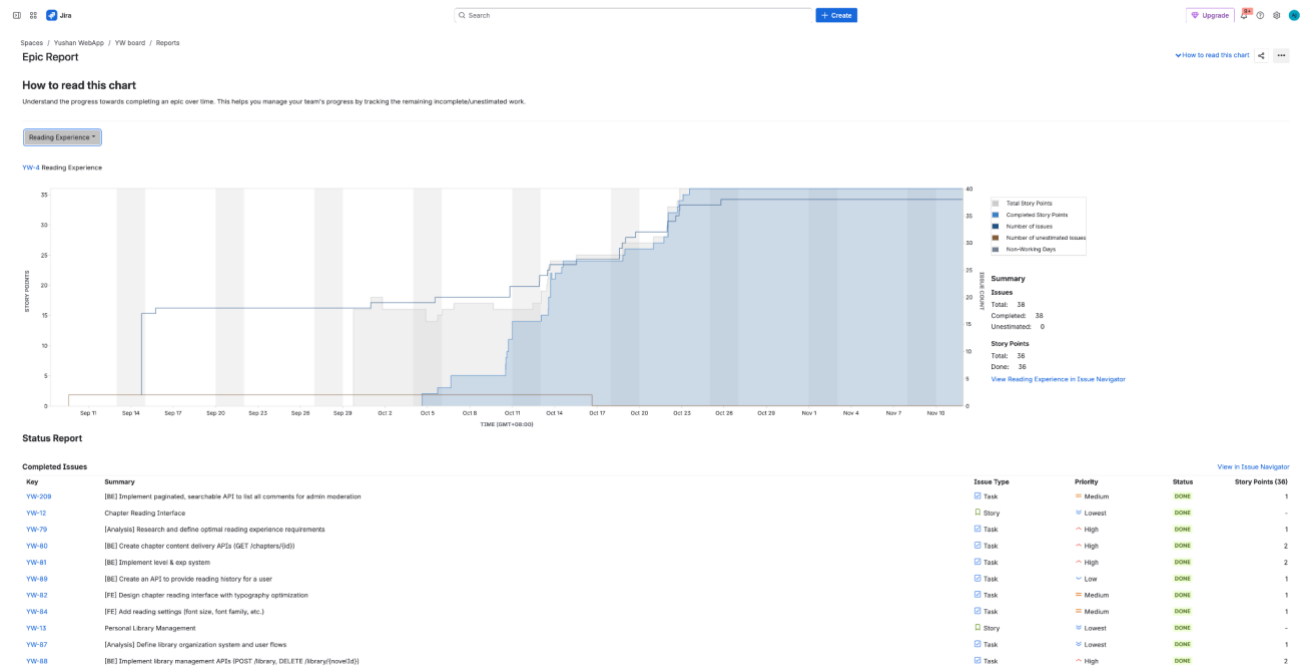


Yang Shuang: Remarkable testing leadership transforming code quality and reliability. Her systematic approach to increasing test coverage from 5% to 80% on frontend and significant contributions to admin testing established robust quality standards across the project.

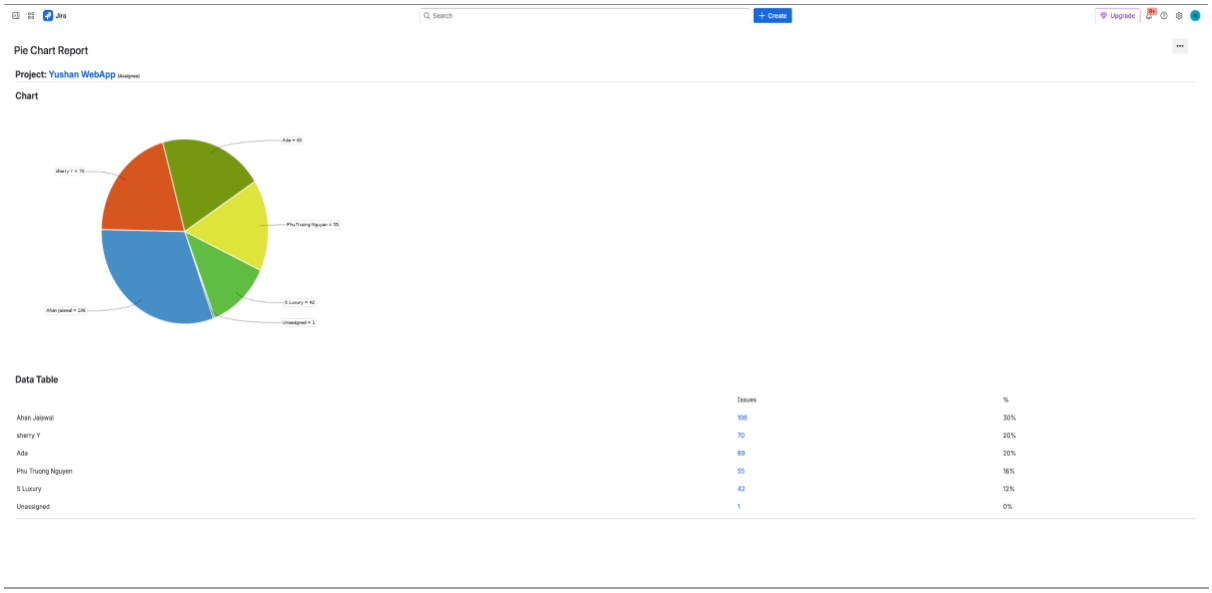
Zhang Yan: Critical backend contributions on complex authentication and database architecture. His implementation of the User Service with email verification, Redis caching, and comprehensive backend testing provided a secure and reliable foundation for the platform.



Zhu Yuhui: Excellent UI/UX delivery creating engaging user experience. Her thoughtful design and implementation of the reading interface, browse pages, and rankings created an intuitive and visually appealing platform that defines the user experience.



The balanced effort distribution, combined with high individual productivity and strong collaboration, resulted in a production-ready platform delivered on time with exceptional quality—a testament to effective Agile practices and committed team execution.



AI Declaration

The Yushan project team utilized **Anthropic's Claude AI model** (Claude Sonnet 4.5) as a productivity tool during the development and documentation process. Claude was employed to:

- Assist in refining technical documentation and report writing for improved clarity and presentation
- Generate boilerplate code structures and configuration templates
- Provide suggestions for code optimization and best practices
- Help format and organize project documentation

Important Clarifications:

1. **All core development work**—including system architecture, feature implementation, testing, deployment, and problem-solving—was performed entirely by human team members.
2. **AI assistance was limited to supportive tasks** such as documentation refinement, grammar checking, content structuring, and generating initial drafts that were subsequently reviewed and modified by team members.
3. **The team takes full responsibility** for the quality, accuracy, and integrity of all deliverables. Any AI-generated content was thoroughly reviewed, validated, and adapted by team members to ensure it accurately represents our work.
4. **Critical decisions**—including technical architecture, feature prioritization, security implementations, and quality standards—were made exclusively by human team members through collaborative discussion and professional judgment.
5. **All code committed to production** was written, reviewed, tested, and validated by team members. AI-generated code suggestions, when used, served only as starting points and were significantly modified to meet project requirements.

This declaration affirms our commitment to academic and professional integrity while acknowledging the appropriate use of modern AI tools to enhance productivity and presentation quality. We understand that the value of this project lies in the knowledge gained, problems solved, and skills developed through hands-on implementation—not in the tools used to document our journey.