

CineQuest: A Distributed Movie Discovery Platform

COMP41720 Distributed Systems Group Project

Fan Ke (24208425) Ze Li (24203409) Shuangning Wei (24206063)

December 19, 2025

Abstract

CineQuest is a microservice-based movie discovery platform demonstrating distributed systems architectural principles through explicit trade-off analysis and decision justification. The system features an API Gateway with centralized authentication (Keycloak), heterogeneous communication patterns (REST, gRPC, Kafka), polyglot persistence (PostgreSQL, MongoDB, Redis, MySQL), and comprehensive fault tolerance mechanisms. This report prioritizes architectural reasoning over implementation details, documenting the “least-worst combination of trade-offs” for each major design decision through Architectural Decision Records (ADRs).

Source Code Repository: <https://github.com/Shelly892/CineQuest>

1 Project Context and Scope

Modern web applications increasingly require scalability, resilience, and rapid evolution. Movie platforms are a particularly useful context: they are read-heavy (browse/search), rely on third-party data providers, and often extend into user engagement features such as ratings, achievements, and notifications. These characteristics create natural fault boundaries and differing scalability needs, making the domain well-suited for exploring distributed-system architecture.

CineQuest enables users to browse movies, submit ratings, perform daily sign-ins, unlock achievements, and receive email notifications. The domain naturally exposes distributed systems challenges:

- **External dependency management:** Movie Service depends on TMDB API with a rate limit of 40 requests per 10 seconds as specified in the official TMDB API documentation and exhibits 200–500 ms latency based on empirical measurements.
- **Varying scalability profiles:** Read-heavy Movie Service vs write-intensive Rating Service; bursty Achievement Notification event processing.
- **Fault isolation:** External API failures (TMDB, SMTP) must not cascade to core functionality.
- **Strict latency constraints:** Minimizing network overhead for real-time internal triggers (Achievement unlocks) while ensuring reliable, decoupled delivery for asynchronous notifications.

2 System Architecture Diagram

The system follows a microservice-oriented architecture with explicit service boundaries. An API Gateway is the single external entry point, while backend services are responsible for distinct business capabilities. Communication patterns include synchronous REST, synchronous gRPC, and asynchronous event-driven messaging.

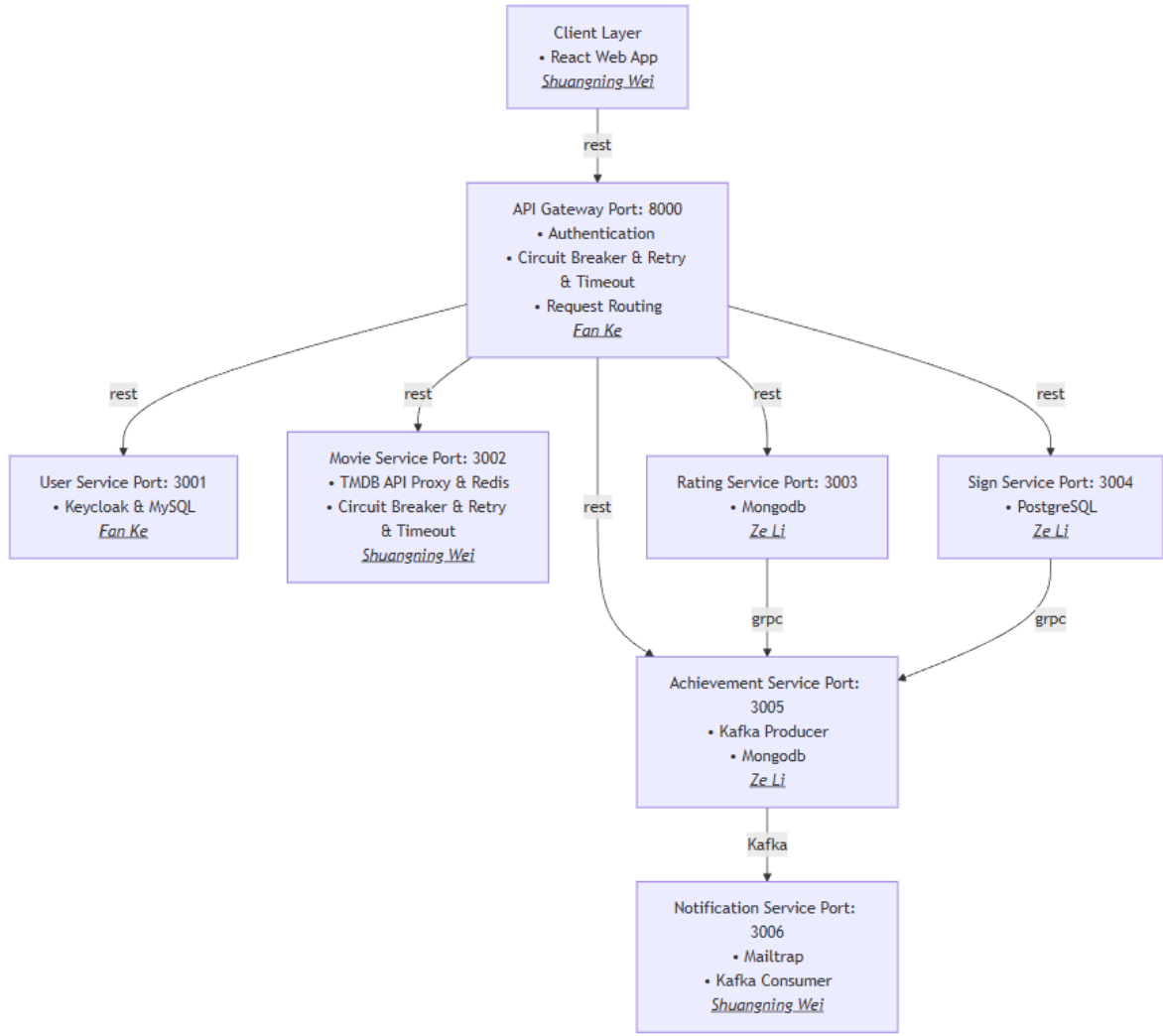


Figure 1: CineQuest System Architecture (service boundaries, communication, and data stores).

Table 1: Service inventory

Component	Port(s)	Responsibility	Data Store	Communication
Frontend	5173(dev)	Web UI	—	REST to Gateway
API Gateway	8000	Routing, JWT validation, Circuit Breaker, resilience policies, fallback-responses, authentication injection	—	REST to services;
Keycloak	3001	OIDC authentication	MySQL	OIDC (auth/token/JWK)
Movie Service	3002	TMDB proxy, caching	Redis	REST, TMDB API
Rating Service	3003	Rating CRUD, stats	MongoDB	REST, gRPC
Sign Service	3004	Daily check-in	PostgreSQL	REST, gRPC
Achievement Service	3005, 50051	Badge query & awarding logic, events publishing	MongoDB	REST, gRPC, Kafka producer
Notification Service	3006	Email delivery	—	Kafka consumer, SMTP

3 Architectural Design and Justification

This section details the architectural choices, focusing on the analysis of trade-offs, service boundaries, communication patterns, and data management as per the assessment criteria.

3.1 Service Decomposition and Responsibilities

Services are decomposed by business capability rather than by technical layer. This design supports independent scaling and evolution, while keeping strong consistency within service boundaries.

3.1.1 Decomposition Rationale: Service Boundaries

The system decomposition follows Domain-Driven Design principles to define Bounded Contexts, ensuring high cohesion within services and loose coupling between them. The rationale for the specific service boundaries is driven by three architectural imperatives:

- **Isolation of External Volatility:** Specific services act as anti-corruption layers for unstable external dependencies.
 - The *Movie Service* encapsulates the TMDB API proxy logic. By isolating this dependency, strict rate-limiting (40 req/10s) and potential upstream latency spikes are contained, preventing them from cascading to critical user authentication or data recording flows.
 - Similarly, the *Notification Service* isolates the SMTP interaction (Mailtrap). By decoupling this via Kafka, slow email transport does not block the synchronous user interaction path.
- **Segregation of Data Access Patterns:** The distinct storage requirements necessitated physical separation to optimize performance.
 - *User and Sign Services* require ACID compliance for identity and daily logs, justifying the use of relational databases (MySQL/PostgreSQL).
 - *Rating and Achievement Services* manage rapidly evolving, unstructured data, necessitating a document-oriented approach (MongoDB).
 - The *Movie Service* is read-heavy and tolerates eventual consistency, allowing it to leverage Redis for aggressive caching without managing complex write transactions.
- **Decoupling Core Actions from Side Effects:** We separated the "Core Domain" (Rating/Signing) from the "Gamification Sub-domain" (Achievements).
 - While *Rating* and *Sign* services handle the primary user intent, the *Achievement Service* acts as a downstream aggregator.
 - This separation prevents achievement rule complexity (e.g., "rate 5 movies to unlock badge") from polluting the clean logic of basic interaction services. The use of gRPC bridges these boundaries to maintain low-latency updates across these physically separated contexts.

3.1.2 Component Details

- **API Gateway:** Centralizes cross-cutting concerns: Security (validates JWTs), Routing, Resilience (circuit breakers), and Context propagation (injects user headers).
- **Movie Service:** Proxy to TMDB with caching and resilience. Uses Redis cache for popular/search/details with different TTLs (e.g., popular: 1h). Implements Circuit breaker + retry + timeout around TMDB calls.
- **Rating Service:** Handles rating CRUD and stats. Prevents duplicate rating per user/movie. Computes per-movie statistics and performs best-effort gRPC notification to Achievement.

- **Sign Service:** Enforces daily sign-in uniqueness using a relational uniqueness constraint (`user_id`, `sign_date`). Notifies Achievement via gRPC.
- **Achievement Service:** The “gamification engine”. Receives progress updates via gRPC, evaluates badge tiers, and publishes `achievement_unlocked` events to Kafka (Avro).
- **Notification Service:** Consumes `achievement_unlocked` (Kafka consumer group `notification-group`) and sends an email via SMTP.

3.2 Communication Models and Data Flow

3.2.1 Client-facing REST via Gateway

The frontend communicates exclusively through the Gateway. Public endpoints are unauthenticated; write operations require valid JWTs. The Gateway injects user headers derived from JWT claims to simplify downstream authorization and user-context handling.

3.2.2 Internal gRPC: progress updates

Rating and Sign notify Achievement using gRPC for strongly typed internal contracts. The interface includes `NotifyRatingSubmitted` and `UpdateSignCount`. This avoids JSON schema drift and reduces overhead for high-frequency internal calls.

3.2.3 Asynchronous Kafka events: `achievement_unlocked`

Achievement publishes domain events to Kafka when a new badge is awarded. Notification subscribes and triggers email delivery. Event schema is governed using Avro + Schema Registry to reduce integration risk when the event evolves.

3.3 Data Management and Consistency

3.3.1 Polyglot persistence

We intentionally adopted different data stores to match different domain requirements:

- **PostgreSQL (Sign):** relational constraints express daily uniqueness naturally and safely under concurrency.
- **MongoDB (Rating/Achievement):** document model supports flexible rating/comment fields and badge records.
- **Redis (Movie):** caches read-heavy results to reduce external TMDB dependency and latency.
- **MySQL (Keycloak):** stable backing store for identity metadata.

3.3.2 Consistency model

- **Within service boundaries:** strong consistency is maintained using each service’s database semantics.
- **Across services:** eventual consistency is accepted. `Rating/Sign` → `Achievement` uses best-effort gRPC (failures do not roll back core writes). `Achievement` → `Notification` uses async events.

3.4 Fault Tolerance and Resilience

3.4.1 Gateway resilience

The Gateway applies circuit breakers and provides standardized fallback endpoints returning HTTP 503 with a consistent JSON payload. This prevents cascading failures and gives the frontend predictable error handling.

3.4.2 External dependency isolation (TMDB)

Movie Service wraps TMDB calls with Resilience4j (circuit breaker/retry/timeout) and returns controlled fallback responses. Redis caching reduces call frequency and mitigates rate limiting.

3.4.3 Failure experiments (demonstration)

We verified graceful degradation through controlled failures:

- **Downstream service failure:** stop a service container (e.g., Rating) and confirm Gateway returns a 503 fallback response.
- **External API instability:** simulate TMDB failures and confirm Movie Service returns fallback responses (empty results/placeholder details).
- **Notification outage:** stop Notification service and confirm core workflows (rating/sign/achievement) continue.

3.5 Security Model and Trust Boundaries

3.5.1 Centralized authentication

Keycloak provides OIDC login/registration and issues JWT access tokens. The Gateway validates JWTs and enforces authentication requirements by route.

3.5.2 Identity propagation and trust boundary

After validating JWTs, the Gateway injects identity context into downstream calls via **X-User-*** headers. This improves simplicity in downstream services but expands the trust boundary:

- Downstream services must **not** be directly exposed to the public network; otherwise headers could be forged.
- For a stronger zero-trust posture, services could additionally validate JWTs or service-to-service mTLS could be enforced (not implemented in this demo for simplicity).

4 Architectural Decision Records and Trade-off Analysis

4.1 ADR-001: API Gateway and Centralised Authentication

Status: Accepted

Context: The system consists of multiple business services. If the frontend were to communicate directly with each service, each service would be required to independently implement authentication, authorization, CORS handling, and resilience logic. This would result in duplicated cross-cutting concerns and unclear security boundaries.

Decision: The system adopts Spring Cloud Gateway (MVC) as a unified API Gateway. The Gateway is responsible for request routing, validating JWT tokens as an OAuth2 Resource Server, enforcing Resilience4j policies, and extracting user information from validated JWTs to be propagated to downstream services via **X-User-*** headers. Keycloak is adopted as the centralized authentication provider.

Consequences:

- **Positive:** Centralized security enforcement, reduced duplication of cross-cutting logic, and a smaller external attack surface (only Gateway is exposed).
- **Negative:** The Gateway becomes a critical path and potential bottleneck. Propagating user identity via HTTP headers expands the trust boundary, making it essential to prevent downstream services from being accessed directly.

4.2 ADR-002: gRPC for Internal Progress Updates

Status: Accepted

Context: After successful write operations, Rating and Sign services must notify the Achievement service. These interactions occur frequently, require clearly defined field semantics, and are strictly internal.

Decision: The system adopts gRPC as the synchronous communication mechanism between the Rating/Sign services and the Achievement service. Protocol Buffers are used to define strongly typed service interfaces.

Consequences:

- **Positive:** Clear and strongly typed contracts, reduced risk of schema drift compared to JSON, and improved performance for high-frequency internal calls.
- **Negative:** Increased build complexity due to proto generation. As a synchronous mechanism, gRPC may amplify latency under extreme load.

4.3 ADR-003: Event-driven Achievement Notification with Kafka

Status: Accepted

Context: When an achievement is unlocked, a notification email should be sent. Implementing this as a synchronous call would increase coupling and expand the failure impact surface.

Decision: The Achievement service publishes a domain event, `achievement_unlocked`, to Kafka. The Notification service subscribes to this topic and sends email notifications.

Consequences:

- **Positive:** Loose coupling between services; notification failures do not block achievement processing.
- **Negative:** Increased deployment complexity (Kafka, Zookeeper). Message delivery semantics require consideration of at-least-once delivery.

4.4 ADR-004: Polyglot Persistence and Redis Caching

Status: Accepted

Context: Different services have heterogeneous data requirements. Movie Service depends on the external TMDB API which has rate limits. Sign Service requires transactional consistency.

Decision: Adopt a polyglot persistence strategy: PostgreSQL for Sign, MongoDB for Rating/Achievement, Redis for Movie metadata caching, and MySQL for Keycloak. For Movie Service, implement a Cache-Aside pattern.

Consequences:

- **Positive:** Improved performance for movie data access; cached metadata reduces TMDB API calls. Each service scales independently.
- **Negative:** Increased operational complexity due to maintaining four different database technologies.

5 Implementation, Correctness, and Reproducibility

5.1 Deployment and Reproducibility

Containerization: All services are packaged as Docker images using multi-stage builds to ensure small image size and clean runtime environments (Frontend: Nginx-based image; Backend: Java base images). All images are pre-built and published to Docker Hub.

Orchestration: Docker Compose is used for local development and integration testing, while Kubernetes manifests are provided for production-style deployment. The Kubernetes configuration includes Deployments, Services, ConfigMaps, Secrets, and PersistentVolumeClaims.

Deployment Environment: Reproducible execution requires Docker (v20+), Docker Compose v2, and Kubernetes tooling (`kubect1` v1.25+). For local Kubernetes-based deployment, Minikube (v1.30+) is used to create a single-node cluster, from which all services are deployed by applying the provided Kubernetes manifests. Since all images are pulled directly from Docker Hub, no local image build is required.

Kubernetes Features:

- Service discovery via ClusterIP Services for internal communication.
- Auto-scaling using Horizontal Pod Autoscaler.
- Health monitoring through liveness and readiness probes.
- Rolling updates with controlled availability.

Verification: Successful deployment is verified by ensuring all pods reach a ready state, the frontend is accessible via port forwarding, and core service endpoints respond correctly. Functional verification includes browsing movies, authenticated user actions, and end-to-end event flows across services.

Repository: The project repository (<https://github.com/Shelly892/CineQuest>) contains comprehensive documentation, including setup instructions, Dockerfiles, Docker Compose configuration, Kubernetes manifests, API endpoint and port mappings, and a verification guide.

5.2 Correctness and Functionality

The system implements and demonstrates:

- Public browsing via Movie Service through the Gateway.
- Authenticated write paths for rating and sign-in.
- gRPC progress updates to the Achievement service with idempotent badge awarding.
- Kafka event publication and consumption leading to email notifications.
- Resilience behavior (fallbacks, graceful degradation) under controlled failures.

6 Limitations, Future Work, and Conclusion

6.1 Limitations and Future Work

- **Security:** Add mTLS for service-to-service communication; reduce trust surface of header-based identity.
- **Observability:** Integrate distributed tracing (OpenTelemetry) for latency analysis and failure root-cause identification.
- **Reliability:** Implement dead-letter queues and exponential backoff for Kafka consumers.
- **Monitoring:** Add Prometheus metrics and Grafana dashboards for real-time system health monitoring.

6.2 Conclusion

CineQuest demonstrates practical application of distributed-systems architectural principles in a realistic scenario. By explicitly documenting decisions and trade-offs (rather than only code), the project shows how scalability, resilience, security, and evolvability can be balanced through context-aware design choices.

7 Team Contributions

This section documents individual contributions to ensure transparency and to acknowledge each team member's efforts.

7.1 Ze Li

- Led the overall system design of the project
- **Rating Service:** Implemented a MongoDB-based rating system
- **Sign Service:** Developed a PostgreSQL-based daily check-in system
- **Achievement Service:** Built the gamification engine
- Debugged and ensured smooth communication across backend services through the API Gateway
- Dockerised both frontend and backend services
- Implemented final Kubernetes deployment for the system
- Contributed to the final report

7.2 Shuangning Wei

- Contributed to the overall system design of the project
- **Movie Service:** Developed a Redis-based movie service integrated with an external API
- **Notification Service:** Developed the notification service and email delivery mechanism
- **Frontend:** Developed the React-based frontend web application
- Integrated frontend and backend services via the API Gateway
- Initialized Kubernetes deployment manifests
- Contributed to the final report

7.3 Fan Ke

- Contributed to the overall system design of the project
- **API Gateway:** Implemented Spring Cloud Gateway with JWT validation, request routing, Resilience4j policies, user context injection, and fallback response handling
- **Keycloak Configuration:** Set up the realm, clients, and user management
- Contributed to the final report

Note: While each team member had primary responsibility for specific services, all architectural decisions were made collaboratively through discussion and consensus.