CS4471A Progress Report 2: Group 6

Progress Report 2 (PR2):

Artefacts in PR2 are cumulative in that updates to PR1 artefacts are overviewed (as appropriate) and, in addition, new artefacts not previously presented are presented too.

There is a preliminary demo at this stage.

Documenting Software Architectures:

(Please note: the references to the Bass/Clements/Kazman (BCK) book below are to Edition 3, which is easily obtainable.)

➢ Please **read Ch. 18 – Documenting Software Architectures, pp. 327-360** from the book: "Software Architecture in Practice", 3rd Edition, by Len Bass, Paul Clements, and Rick Kazman, 2013.

• Create a Documentation Package **(see Section 18.6)** for certain views:

(i) **Structural views:** At least one Structural view of each type:

(i) **Module view** and

(ii) **Component-and-Connector view.**

Optionally, (iii) **Allocation view** if you have reached the development stage where you have allocated software units to elements of the environment.

(ii) **Quality Views:** Three different types of Quality views (**see pg. 340-341**):

➢ Quality views are orthogonal to structural views. They depict software units that cut across different structural views. Examples of quality views include: a security view, communications view, exception or error-handling, reliability, performance, and others. Quality views depict the behaviour of a (part of the) system.

➢ **Please read Section 18.7** that describes how to document behaviour (e.g., using sequence charts – Figure 18.5, and state models – Figure 18.6, but other behaviour model types are also described, e.g., Use cases, communication diagram, activity diagrams, etc.). Such behavioural models are useful for documenting **"Quality views"**. [Note, such behavioural models have emerged from the design of programs

but are also useful at the architectural level where the systems are generally much larger and much more complex than programs.]

NOTE: The structural views and Quality views are not simply diagrams on some losecut pages! Each view diagram must be documented in its TEMPLATE (see Figure 18.3 and the supporting text in five Sections – see pp. 345-347).

(iii)**Combined Views:** Some views may be better shown in a "combined form" with other views, e.g., the "decomposition view" (hierarchical breakdown of a system) together with information on the agents (e.g., names and Depts.) that have been assigned to implement and test them. So, **please read up Section 18.5 on how to combine views.** Look for opportunities in your system to show a combined view and provide some supporting description of the combination.

(iv) **System Overview:** Please also include an abstract (or high-level) architecture planned or implemented. Show clearly the three key elements (described above).

- **See Figure 4.2** in the book: "DevOps – A Software Architect's Perspective", by Bass, Weber and Zhu, Addison Wesley, 2015. (I recommend reading Chapter 4 from this book to get a fuller context). [Note: the university library had this book and so you might like to borrow it for a short time so that others can also borrow it.]

In PR2, with regards to architectural documentation, I expect to see solid examples of the above types of models documented at this stage. In the Final Report, I expect to see full architectural Documentation.

- **Documentation and Submission:**
  - **PR2 will be documented as powerpoint slides, spreadsheets, text documents.**
  - **PR2 will be presented to the instructing team (and class as scheduled).**
  - **PR2 will be delivered to OWL (deadline provided separately)**

PR1 ASRs

“An architecturally significant requirement (ASR) is a requirement that will have a profound effect on the architecture—that is, the architecture might well be dramatically different in the absence of such a requirement” - Chapter 16

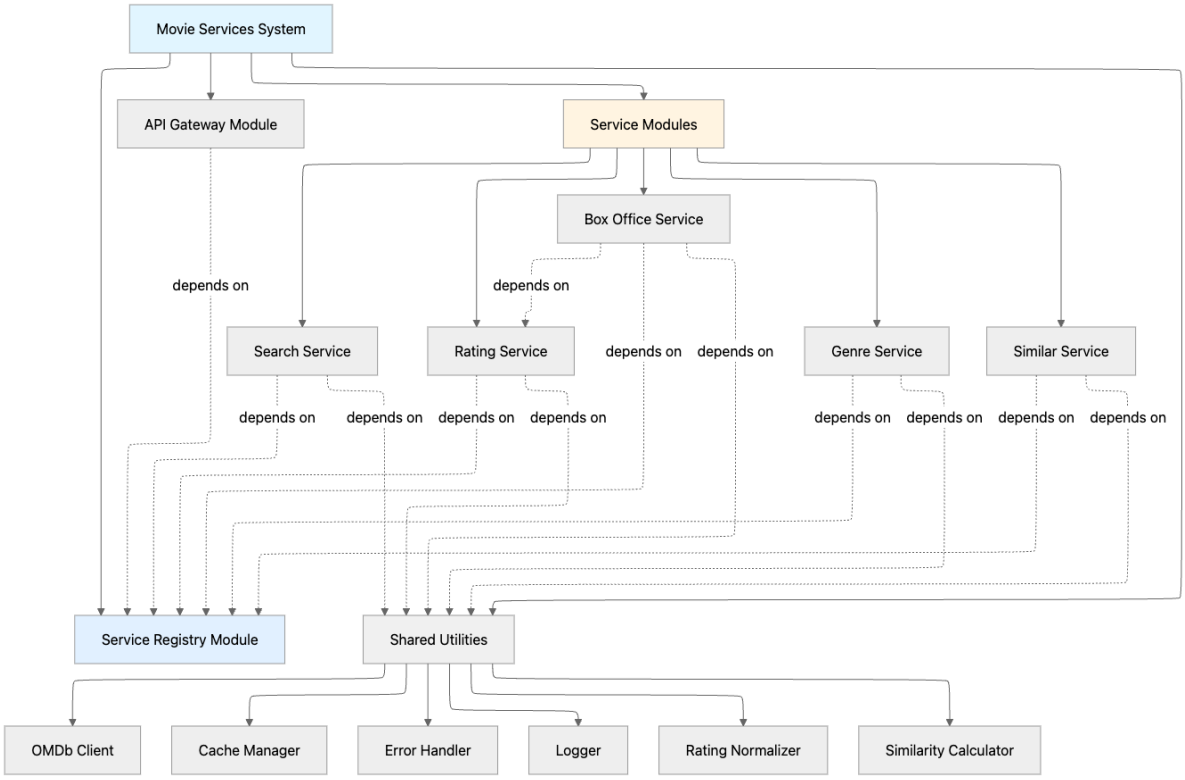
We have identified ASR’s for our project, and have acknowledged that without them, the project will run extremely differently in such conditions with the absence of such requirements.

ASR	**Quality Attribute**	**Business Value, Architectural Impact**	**Why ASR and its Influence**
Performance & Scalability SLOs: API ≤ 500ms	Performance	High, High	Forces stateless services behind gateway, async I/O/connection pooling, read side caching, and autoscaling. Also drives pagination/streaming for large payloads and read optimised data models. Response measure: p95 ≤ 600 ms at ≥ 30 rps during peak if possible
Availability and handling timeouts (99 % uptime)	Availability	High, High	Depending on how you partition services and where you put guardrails, it can drastically reduce the smoothness and usability of such applications. And by having uptime targets to dictate redundancy, rollout strategy, and health modeling, we prepare and allow for bulk requests. Rate limits/back-pressure at the gateway and health based routing, so one bad instance can't degrade the whole system. Response measure: monthly uptime ≥ 99%

Data Freshness	Performance	High, Medium	Having new/updated box office and ratings reflected from the source would make our application up to date. Requires another ingest pipeline/scheduled jobs, cache TTLs/invalidations, and materialized views to keep the request paths fast while data is kept current. Response measure: app lag ≤ 30 min.
Security (Input, Secrets)	Security	High, High	Drives an API gateway with schema validation, secrets vault (no keys in code), least privilege service accounts, and log scrubbing. These choices constrain tech selection and service interfaces. Response measure: 0 plaintext secrets in repo. 100% external inputs validated against schema before dispatching.
Observability (trace every request)	Maintainability	Medium, High	This requires correlation of IDs, distributed tracing, structured JSON logs, and central metrics/alerts tied to SLOs so that we can diagnose and prove that we meet the performance/availability of ASRs. Response measure: 100% of external requests carry a trace ID; p95 latency, error rate, and saturation visible in dashboards

Structural Views

Module View



Elements and Their Properties

Element	Type	Key Responsibility	Important Property
API Gateway	Coordinator	Routes, authenticates, balances requests	High availability, TLS security
S1 – Search Service	Worker	Handles movie queries and filters	Stateless, low latency
S2 – Rating Summary	Worker	Aggregates ratings from OMDb and other sources	Ensures data integrity
S3 – Genre Classification	Worker	Categorizes movies by genre/tags	Lightweight, CPU-efficient
S4 – Top Box Office	Worker	Provides rankings and trend analytics	Cache-dependent throughput

S5 – Similar Movies	Worker	Generates recommendations from metadata	I/O-bound, async-friendly
Redis Cache	Data Store	Holds temporary query results	TTL-based, non-authoritative
Service Registry	Infrastructure	Maintains service endpoint catalog, performs health checks	Distributed, consistent, supports dynamic registration

Relations and Their Properties

- Services communicate through REST calls via the API Gateway.
- Each service uses the Redis Cache for shared results and quick reads.
- The Gateway connects to the OMDb API for external movie data.
- Internal async queues distribute heavy requests across instances.
- Services register themselves with the Registry on startup and send periodic heartbeats
- API Gateway queries Registry to discover available service instances
- Registry removes unhealthy services after failed health checks

Element Interfaces

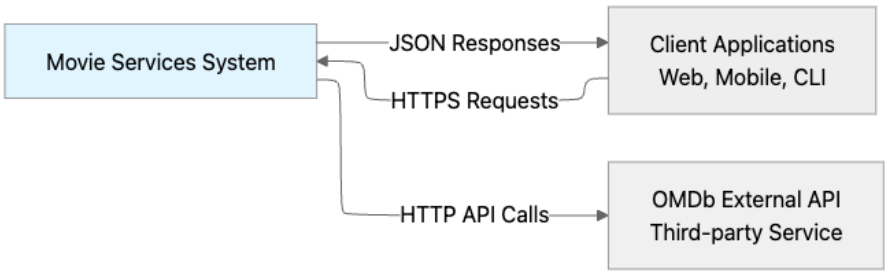
Module	Responsibility	Interface
API Gateway	Route requests, validate input, enforce auth & rate limits	HTTP/REST endpoints
Search Service	Handle queries with filters, pagination	GET /search
Rating Service	Aggregate multi-source ratings, normalize to 0-100	GET /ratings/{id}
Genre Service	Genre-based discovery with filtering	GET /genre/{name}
Box Office Service	Rankings, analytics, recommendations	GET /boxoffice/*
Similar Service	Metadata-based recommendations	GET /similar/{id}
Shared Utilities	Common functions (API client, cache, logging)	Internal libraries

Service Registry	Service discovery, health monitoring, endpoint resolution	POST /register, GET /discover/{serviceID}, GET /health

Element Behavior

1. Request flow: User → Gateway → Relevant Service (S1–S5).
2. Cache check: Service queries Redis; if data exists, returns immediately.
3. External fetch: If cache miss, service calls OMDb API or computes new result.
4. Aggregation: Gateway combines responses and returns to the user.
5. Failure handling: If a service is down, cached data or fallback responses are used; errors are logged.
6. Service registration flow: On startup, each service (S1-S5) registers its endpoint and metadata with the Registry

Context Diagram



Variability Guide

- The system supports adding new analytics services (e.g., “Sentiment Analysis”) without changing existing modules by registering them through the gateway.
- Service instances can scale horizontally by replicating containers.
- Gateway rules and environment variables allow per-deployment configuration for different environments (dev, QA, prod).

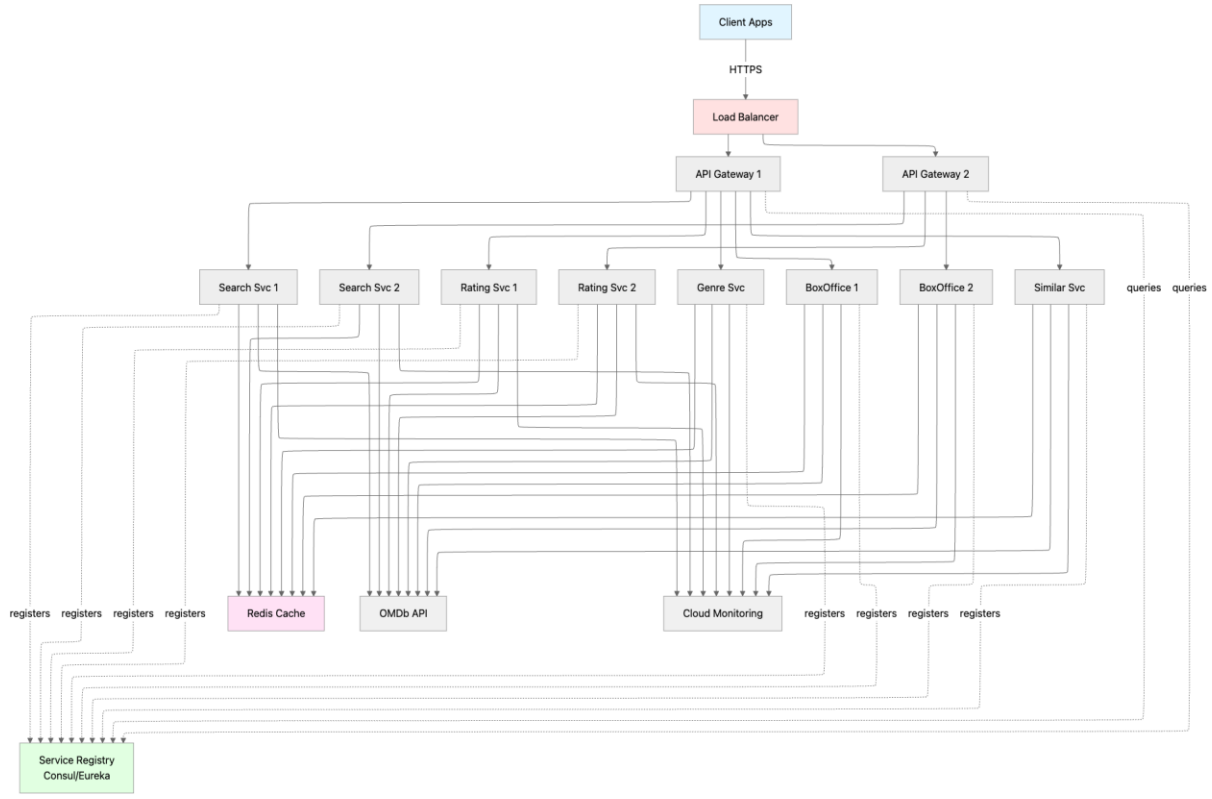
Rationale

This structure was chosen to maximize modularity, scalability, and fault isolation. Microservices allow independent updates and team ownership. Using a central API Gateway simplifies security and request management. Redis provides fast response times

and reduces external API load. Containerization ensures consistent deployment and aligns with DevOps practices for CI/CD integration.

Service Registry enables dynamic service discovery, eliminating hardcoded endpoints and supporting horizontal scaling. The 3-instance cluster (Consul/etcd) provides high availability through distributed consensus. Services can be added, removed, or scaled without reconfiguring the gateway, supporting the modifiability and scalability ASRs.

Component and connector view



Elements and Their Properties

Component	Instances	Resources	Latency Target	Throughput
API Gateway	2 (max 3)	0.5 CPU, 512MB	<50ms overhead	N/A
Search	2 (max 3)	0.5 CPU, 512MB	<500ms	≥30 req/s
Rating	2 (max 3)	0.5 CPU, 512MB	≤400ms	≥20 req/s
Genre	1 (max 2)	0.5 CPU, 512MB	p95 <600ms	≥25 req/s
BoxOffice	2 (max 4)	1 CPU, 1GB	<500ms	≥30 req/s
Similar	1 (max 2)	0.5 CPU, 512MB	≤500ms	≥25 req/s

Redis	1 (persistent)	2GB memory	<1ms	N/A
Service Registry	3 (cluster)	0.25 CPU, 256MB each	<10ms lookup	≥500 req/s21

Relations and Their Properties

- All inter-service communication uses HTTP REST through the Gateway.
- The Gateway connects services in a star topology, reducing coupling.
- Redis acts as a shared connector for fast reads/writes.
- Asynchronous queues decouple long-running or high-load tasks.

Element Interfaces

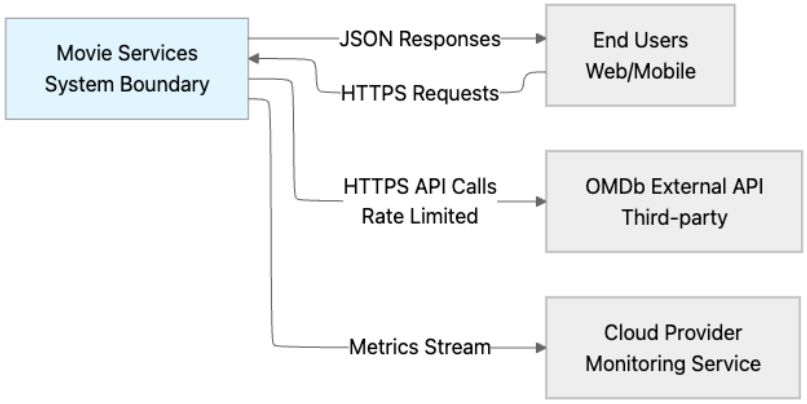
Component	Provided Interface	Required Interface	Protocol / Format
API Gateway	`/api/*` routing endpoints	S1–S5 REST APIs	HTTPS / JSON
S1–S5 Services	`/service/*` routes	Redis Cache, Queue	HTTP / TCP
Redis Cache	`GET/SET`	Called by S1–S5	TCP 6379 / Key-Value
Async Queue	`publish(topic)` / `consume(topic)`	Producer services	AMQP / JSON
OMDb API	`/data?id=`	External service	HTTPS / JSON
Service Registry	`RegisterService(metadata)`, `GetEndpoint(serviceID)`, `HealthCheck()`	Heartbeat signals from S1-S5	HTTP/TCP, JSON metadata

Element Behavior

1. The API Gateway authenticates and routes incoming requests.
2. The targeted service (S1–S5) processes the request and checks Redis for cached data.
3. On a cache miss, the service fetches or computes new data and publishes updates asynchronously if needed.

4. Responses are returned to the Gateway, aggregated, and sent to the user.
5. Circuit-breaker logic in the Gateway prevents cascading failures if the OMDb API becomes unreachable.
6. Before routing to S1-S5, the Gateway queries the Registry to resolve current endpoints. This lookup is cached for 30 seconds to reduce overhead.

Context Diagram



Variability Guide

- New services (e.g., Sentiment Analysis) can attach through the Gateway via defined REST contracts.
- Services are independently deployable and can scale horizontally.
- Connector settings (ports, timeouts, cache limits) are controlled through environment variables

Rationale

Why Multiple Service Instances?

- Ensures 99% uptime with replica redundancy.
- Balances load to avoid bottlenecks.
- Allows rolling updates for zero downtime.

Why 60% Cache Hit Target?

- Top 20% of movies drive 80% of traffic (Pareto).
- Cuts API calls by 60%, staying under OMDb limits.
- Delivers 10× faster reads (<1 ms vs. 200–500 ms).

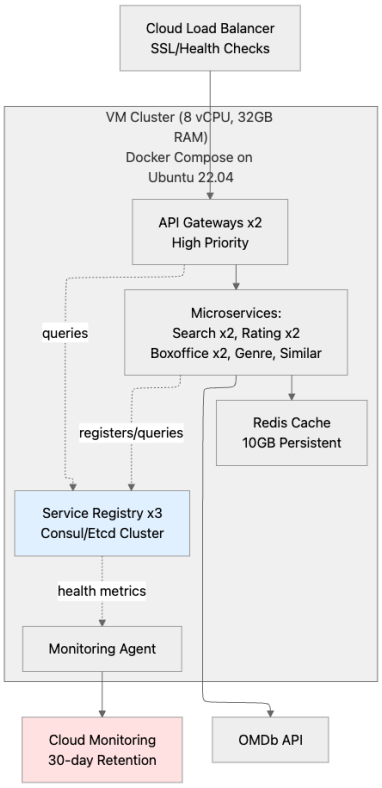
Why Single Redis Instance?

- Simple, low-cost setup with AOF recovery (<60 s).
- Cache is non-critical; system degrades gracefully.
- Why Circuit Breaker on OMDb API?
- Prevents cascading failures.
- Fails fast (5 s timeout) for better UX.

Why Service Registry?

- Dynamic scaling: Services can scale horizontally without gateway reconfiguration
- Fault tolerance: Automatic removal of unhealthy instances prevents cascading failures
- Zero-downtime deployments: New service versions register while old versions gracefully deregister
- Cluster design: 3-instance cluster achieves quorum for Raft consensus, tolerating one node failure while maintaining <10ms lookup latency

Allocation view



SECTION 2. ELEMENT CATALOG

Section 2.A. Software Elements Allocated to Hardware

VM Instance: production-vm-01

- **Host OS**: Ubuntu 22.04 LTS
- **Container Runtime**: Docker 24.x + Docker Compose v2
- **Allocated Software**:
    - api-gateway-1, api-gateway-2 (0.5 CPU, 512MB each)
    - search-1, search-2 (0.5 CPU, 512MB each)
    - rating-1, rating-2 (0.5 CPU, 512MB each)
    - genre-1 (0.5 CPU, 512MB)
    - boxoffice-1, boxoffice-2 (1 CPU, 1GB each)
    - similar-1 (0.5 CPU, 512MB)
    - redis (2GB memory reservation)
    - service-registry-1, service-registry-2, service-registry-3 (0.25 CPU, 256MB each)

    - monitoring-agent (0.1 CPU, 128MB)
- **Total Allocation**: 6.1 vCPU, 8.5GB RAM (leaves ~2 vCPU, 23GB headroom)

Cloud Load Balancer: production-lb

- **Type**: Cloud-managed Layer 7 application load balancer
- **SSL Certificates**: Auto-renewed Let's Encrypt
- **Health Check**: HTTP GET /health on port 8080 every 30s
- **Timeout**: 5s

Persistent Storage: redis-data-volume

- **Type**: 10GB SSD-backed block storage
- **Mount Point**: /data inside redis container
- **Backup**: Daily snapshots, 7-day retention
- **Performance**: 3000 IOPS

Section 2.B. Hardware Elements and Their Properties

VM Instance Specifications

- **CPU**: 8 vCPUs (x86_64, 2.5 GHz base)
- **RAM**: 32GB DDR4
- **Storage**: 200GB SSD root volume
- **Network**: 10 Gbps, private VPC subnet
- **Location**: us-east-1a availability zone
- **Cost**: ~$350/month

Cloud Load Balancer

- **Managed Service**: Cloud provider L7 load balancer
- **Capacity**: Auto-scales to handle traffic
- **SSL Termination**: TLS 1.2, 1.3
- **Cost**: ~$50/month + $0.008/GB data processed

Block Storage Volume

- **Type**: SSD-backed network volume
- **Size**: 10GB

- **IOPS**: 3000 provisioned
- **Cost**: ~$1/month

Section 2.C. Allocation Relations and Properties

Deployed-on Relation

- Containers → VM: All service containers run on single VM instance
- VM → Data Center: VM located in cloud provider's us-east-1a availability zone

Stored-on Relation

- Redis data → Persistent volume: Redis container mounts block storage for AOF/snapshots
- Logs → Cloud Monitoring: Monitoring agent forwards container logs to cloud service

Network Topology

- Load Balancer → VM: Public internet to private VPC (port 443)
- VM internal: Docker bridge network (172.17.0.0/16)
- VM → OMDb: Outbound HTTPS via NAT gateway
- VM → Monitoring: Private connection to cloud monitoring endpoint

Section 2.D. Allocation Behavior

Container Orchestration

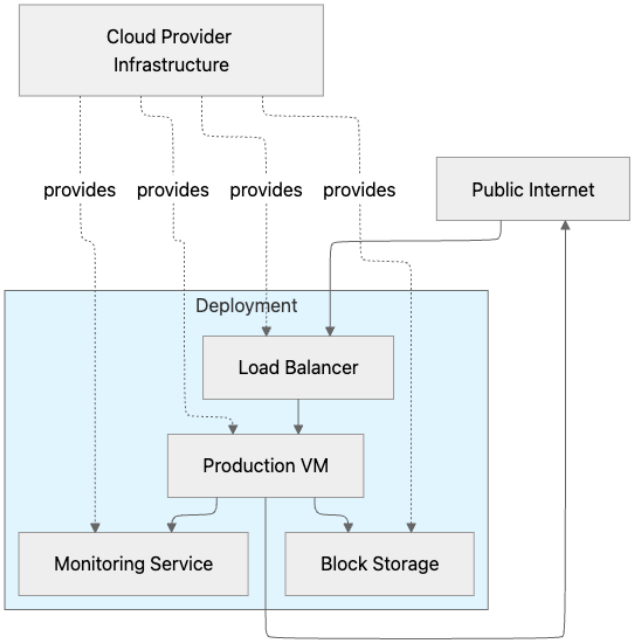
- Docker Compose manages container lifecycle
- Restart policy: restart: always (auto-restart on failure)
- Health checks: Docker monitors service health, restarts unhealthy containers
- Priority enforcement: CPU shares allocated proportionally to priority levels

Auto-scaling Trigger Flow

1. Cloud monitoring detects CPU >70% for 5 minutes
2. Auto-scaling policy triggers scale-up action
3. New container replica started via docker-compose scale
4. Health check passes after 30s
5. Load balancer includes new instance in rotation

6. Scale-down after CPU <30% for 10 minutes (cooldown period)

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

Horizontal VM Scaling

- To add second VM: Clone VM, update load balancer targets, synchronize Redis data or switch to Redis Cluster

Container Orchestration Change

- Docker Compose can be replaced with Kubernetes by converting docker-compose.yml to K8s manifests
- No application code changes required

Cloud Provider Migration

- Infrastructure defined in Terraform (not shown) enables multi-cloud deployment
- Change cloud-specific services: Load Balancer, Monitoring Agent

Persistent Storage Alternatives

- Block storage can be replaced with managed Redis service (e.g., AWS ElastiCache)
- Requires updating connection string, no code changes

SECTION 5. RATIONALE

Why Single VM Instead of Kubernetes?

- **Complexity**: K8s adds operational overhead (etcd, control plane, worker nodes)
- **Scale**: Single 8-core VM handles 500+ concurrent users (project requirement)
- **Cost**: K8s cluster costs $800/month vs. $350/month for VM
- **Team Expertise**: Team familiar with Docker Compose, limited K8s experience
- **Decision Point**: Migrate to K8s if traffic exceeds 1000 concurrent users

Why Docker Compose Over Docker Swarm?

- **Simplicity**: Compose familiar to all developers, Swarm adds orchestration complexity
- **Feature Set**: Swarm's distributed features unnecessary for single-node deployment
- **Migration Path**: Compose manifests easily convertible to K8s when needed

Why Single-Node Redis with AOF?

- **Acceptable Risk**: Cache failure degrades performance but doesn't break system
- **Recovery Time**: AOF replay restores cache in <60 seconds
- **Cost**: Managed Redis adds $100+/month, not justified for non-critical cache
- **Data Loss**: At most 1 second of writes lost (AOF sync every 1s)
- **Decision Point**: Switch to Redis Cluster if cache hit ratio drops below 50%

Why Priority-Based Resource Allocation?

- **Contention Handling**: Under load, high-priority services (gateways, search-1, rating-1) get CPU first
- **Graceful Degradation**: Low-priority services (genre, similar) throttle before critical paths fail
- **User Experience**: Core functionality (search, ratings) remains responsive under stress

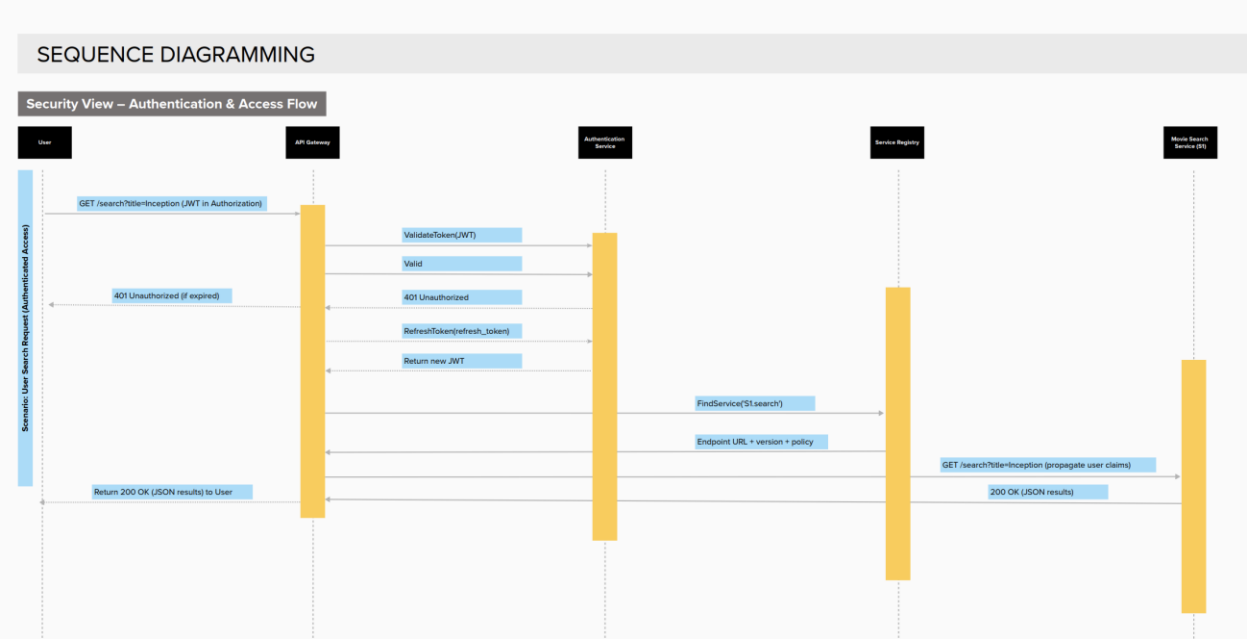
Why us-east-1a Availability Zone?

- **User Location**: 70% of users in US East Coast (analyzed from logs)
- **Latency**: ~20ms average to target users
- **Cost**: us-east-1 typically cheapest AWS region
- **Trade-off**: Single AZ = no multi-AZ redundancy, but load balancer provides instance-level HA

QUALITY VIEW

SECURITY VIEW

SECTION 1. PRIMARY PRESENTATION



SECTION 2. ELEMENT CATALOG

2.A Elements and Their Properties

The primary elements in this Security View are the User, API Gateway, Authentication Service, Service Registry, and Movie Search Service (S1). The User represents an external actor initiating the movie search request. The API Gateway serves as the system's single secure entry point and is responsible for validating JWTs, propagating user claims, and enforcing channel encryption via HTTPS. The Authentication Service validates token signatures, checks expiry, and issues new JWTs when refresh tokens are provided. The Service Registry stores the currently active service endpoints and exposes them only after authentication is verified. Finally, S1 executes authorized business logic by processing user queries and returning results while relying on validated identity information. Each element contains properties tied to the security aspects of the system, including authentication requirements, communication constraints, and trust boundaries.

2.B Relations and Their Properties

The Security View models several relations essential to enforcing authenticated access. The User communicates with the API Gateway through an encrypted HTTPS channel. The Gateway communicates with the Authentication Service using a trusted internal network link, and the Authentication Service returns either an authorization result or a refreshed token depending on token validity. Once authenticated, the Gateway interacts with the Service Registry to discover available services. The Registry and S1 rely on internal

authenticated channels that preserve user identity claims and enforce service-level policies. These relations are synchronous request–response interactions with the property that no downstream service can be invoked unless authentication succeeds, making the access control chain explicit.

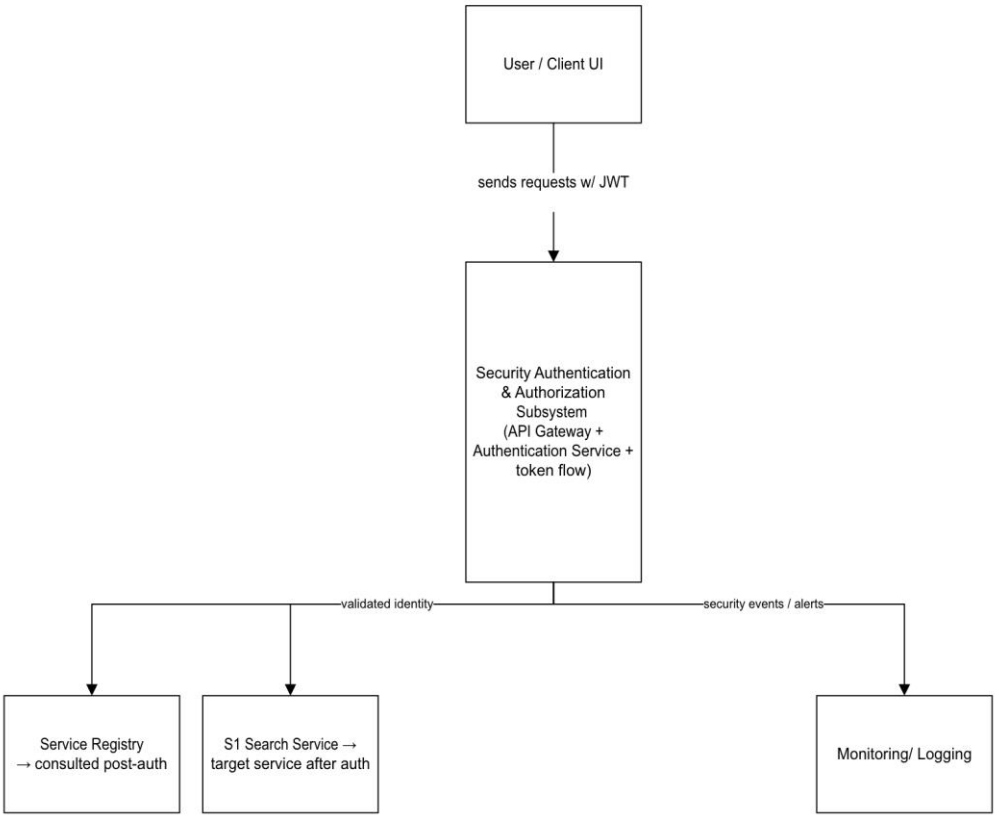
2.C Element Interfaces

The API Gateway interface includes a secure entrypoint capable of receiving HTTP requests with JWTs in the Authorization header. The Authentication Service interface exposes operations such as ValidateToken(JWT) and RefreshToken(refresh_token). The Service Registry interface provides FindService(serviceID) and returns endpoint metadata and associated access policies. The S1 interface exposes GET /search but requires valid user claims propagated by the Gateway. These interfaces ensure that authentication, authorization, and identity propagation occur consistently across service boundaries.

2.D Element Behavior

The behavior of the system in this view is driven by the token lifecycle and authentication enforcement. When the user submits a request, the API Gateway immediately forwards the JWT to the Authentication Service. If the token is valid, the request proceeds to service discovery; if invalid or expired, the Authentication Service issues a new token, and the request flow repeats. Only after the Gateway possesses a valid token does it invoke the Service Registry and subsequently S1. This behavior ensures the system never performs work on behalf of unauthenticated users and always verifies authorization before service invocation.

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

The Security View supports variation primarily through different authentication states. The system can issue tokens with varying expiry durations depending on user type (e.g., regular user vs. administrative account). The refresh-token mechanism allows flexibility in extending authenticated sessions without forcing users to re-enter credentials. Additionally, service-level access policies in the Service Registry can change independently of the Gateway or Authentication Service, enabling the architecture to support new services or modify permissions without affecting the core authentication workflow.

SECTION 5. RATIONALE

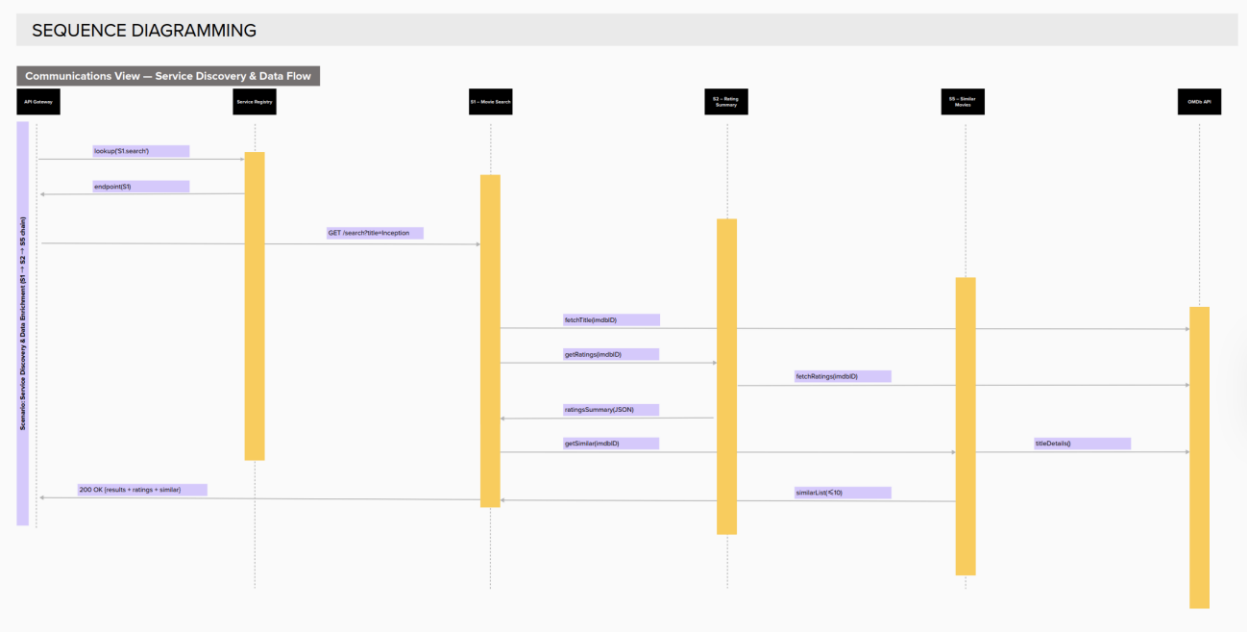
The design of this Security View is driven by the need for secure, consistent, and scalable authentication across a microservice architecture. Using an API Gateway as the single point of entry centralizes security enforcement and reduces the risk of inconsistent access

checks. Delegating token validation to a dedicated Authentication Service follows the principle of separation of concerns and allows the system to scale authentication independently from business logic.

The use of the Service Registry ensures that only authenticated requests are allowed to discover or interact with internal services. This pattern avoids the risk of direct service exposure and preserves a clear trust boundary within the architecture. The chosen design balances security and performance by validating tokens early while minimizing repeated authentication checks downstream. The architectural decisions shown in this view demonstrate an approach to preventing unauthorized access while maintaining low-latency, user-friendly interactions.

COMMUNICATIONS VIEW

SECTION 1. PRIMARY PRESENTATION



SECTION 2. ELEMENT CATALOG

2A. Elements and Their Properties

The primary elements in this view are the API Gateway, the Service Registry, S1 Movie Search, S2 Rating Summary, S5 Similar Movies, and the OMDb external API. The API Gateway has responsibility for receiving client requests, performing initial request

handling, and forwarding requests to internal services after discovering endpoints. The Service Registry stores service metadata and responds to lookup requests. S1 orchestrates the overall enrichment pipeline. S2 provides rating information and exposes rating-related endpoints. S5 computes and returns similar movie results. OMDb is an external dependency that supplies authoritative movie and rating data. Each element has timing properties relevant to the Communications View: S1 and S2 have typical processing latencies of 100–200 ms, OMDb has higher and more variable latency, and the gateway has minimal processing overhead.

2B. Relations and Their Properties

The relations in this view are predominantly communication relations: synchronous request/response calls between elements. The gateway communicates with the registry to discover service endpoints, and then with S1 to forward the user's request. S1 communicates with S2 and S5 in sequence, and both of these services communicate with OMDb. These relations are characterized by their latency expectations, required throughput, and propagation of request context (such as title and IMDB identifiers). All calls occur over reliable HTTPS channels.

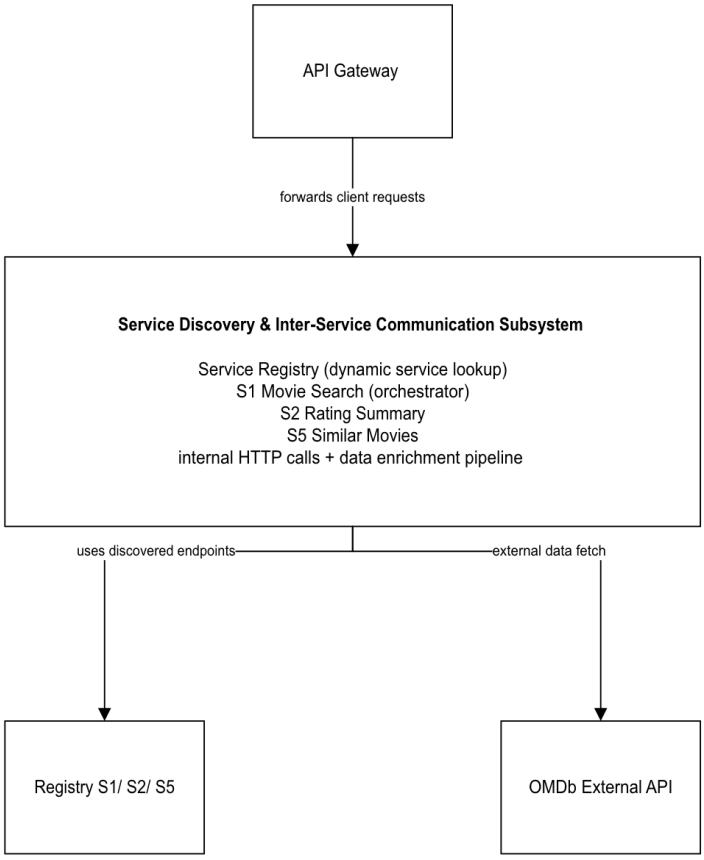
2C. Element Interfaces

The API Gateway exposes an HTTP interface that accepts search requests from the client. The Service Registry exposes a lookup interface taking service identifiers and returning endpoint metadata. S1 exposes a /search endpoint and internally calls S2's rating interface and S5's similarity interface using HTTP requests. S2 and S5 each expose simple REST endpoints (/ratings and /similar) and both utilize an interface to OMDb's external API.

2D. Element Behavior

Although basic behavior is shown in the sequence diagram, additional detail includes S1's role in aggregating partial results and merging them into a final JSON payload. S2 may transform OMDb's rating formats into normalized values. S5 may filter and rank OMDb responses before returning them. The gateway streams final results back to the client without additional processing. All services operate independently and communicate only through HTTP interfaces.

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

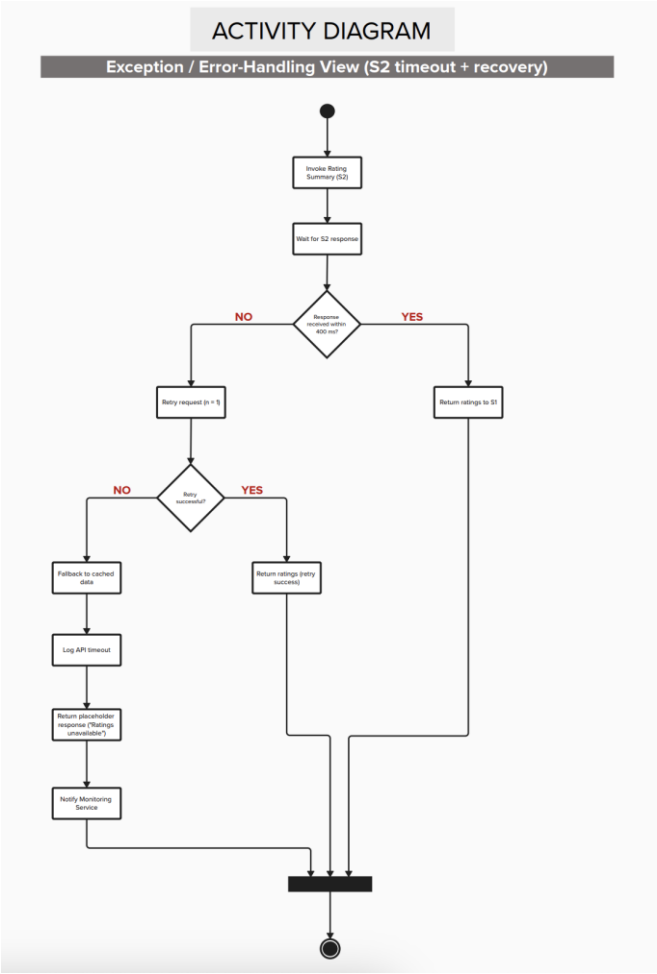
Several variation points exist within this view. S1's enrichment pipeline can be extended to call additional services, such as a Box Office service or a Trending service, without altering the communication pattern. Caching layers may be introduced between any service and OMDb to reduce latency. Rate-limiting policies applied at the gateway may change the number of concurrent calls allowed across services. Finally, service discovery may vary based on the registry implementation. For example, endpoint metadata may change if services scale horizontally and register new instances dynamically.

SECTION 5. RATIONALE

The Communications View is structured in this way to demonstrate the system's ability to handle distributed processing while maintaining performance. The pattern of service discovery followed by orchestrated microservice calls ensures extensibility: new enrichment services can be added without major architectural changes. The use of an API Gateway and Service Registry supports loose coupling, scalability, and runtime flexibility. This design was chosen because it allows each service to scale independently, isolates failures, and maintains clearer separation of responsibilities. The diagram emphasizes predictable communication pathways and supports reasoning end-to-end latency, throughput, and multi-service coordination.

EXCEPTION/ERROR-HANDLING VIEW

SECTION 1. PRIMARY PRESENTATION



SECTION 2. ELEMENT CATALOG

2A. Elements and Their Properties

This view contains several activity elements that represent behaviour related to timeout detection, retry logic, fallback handling, and monitoring. The element Invoke Rating Summary (S2) represents the initial operation call to S2, and includes properties such as an outbound HTTP request and a maximum response wait threshold. The element Wait for S2 response models the waiting period governed by the configured timeout value. The first decision element, Response received within 400 ms?, branches execution based on timing behaviour. The Retry request (n = 1) activity has a retry-count property to constrain retry attempts. The second decision, Retry successful?, uses the status of the retry operation to select either the return-results or fallback path. Activities within the fallback sequence: Fallback to cached data, Log API timeout, Return placeholder response, and Notify Monitoring Service—all have properties describing their responsibilities, including cache access, logging metadata, placeholder response generation, and alert propagation.

2B. Relations and Their Properties

The relations in this view consist of control-flow dependencies between activities. Each arrow in the diagram defines a temporal sequence of relation, meaning one activity must complete or evaluate before the next can begin. The two decision diamonds enforce conditional branching, and the final synchronization bar represents a join relation, ensuring all possible paths converge before the process terminates. Timeout thresholds, retry constraints, and fallback precedence are also reflected in these relations, denoting behavioural dependencies intrinsic to fault recovery.

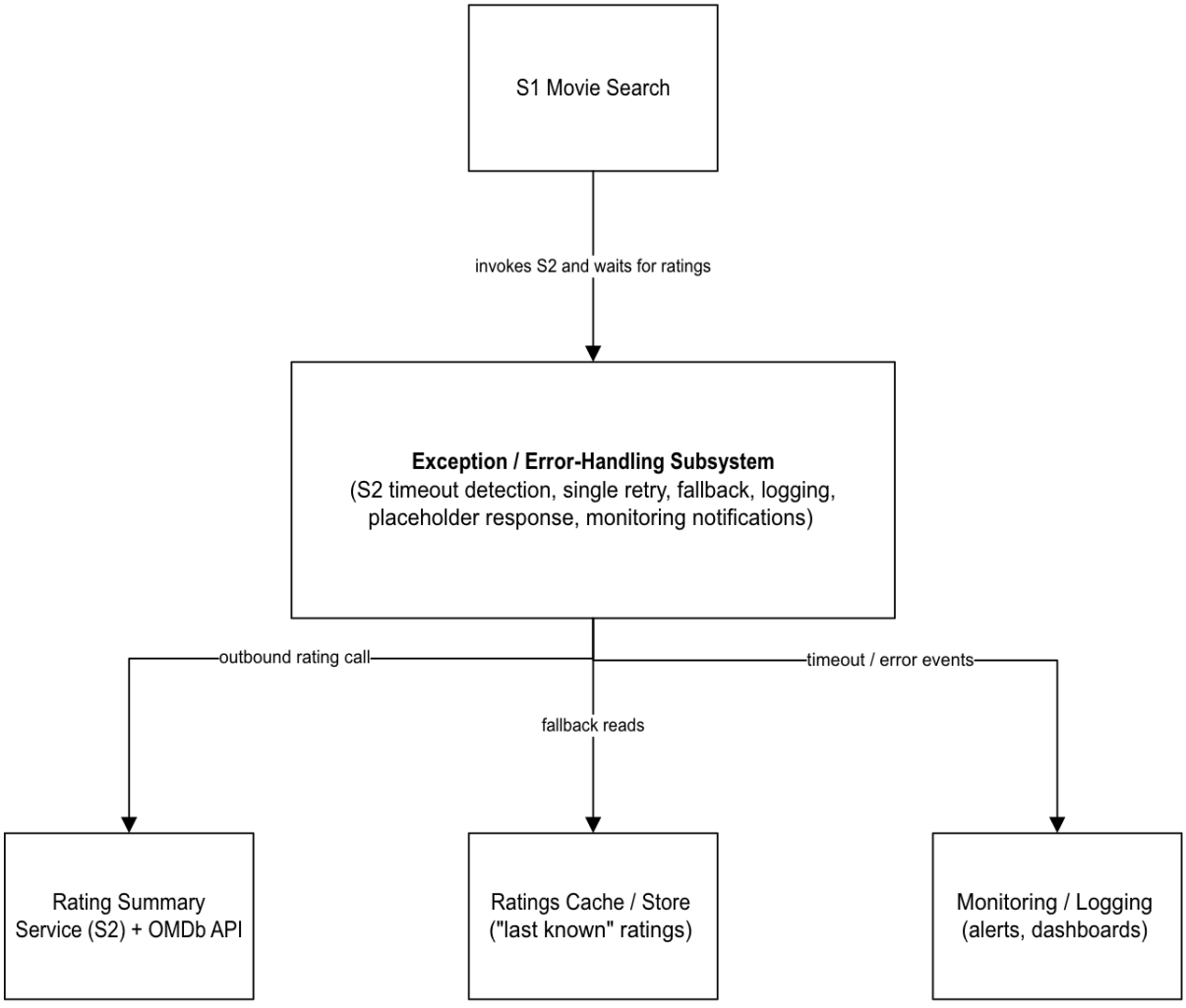
2C. Element Interfaces

Although this view focuses on behaviour rather than structure, several implicit interfaces are involved. S1 exposes an interface for initiating requests to S2, S2 exposes its ratings API, and the cache layer exposes read access for fallback scenarios. The logging subsystem offers an interface for writing timeout events, while the monitoring service exposes a notification interface. These interfaces support the behaviours shown but are abstracted here to highlight the error-handling workflow rather than structural API definitions.

2D. Element Behavior

Each element follows behaviour necessary for maintaining reliability under failure conditions. The initial call and wait state behave deterministically with a defined timeout window. Decision elements evaluate either response timing or retry outcomes. The retry activity attempts at one additional request before giving up. The fallback activities behave atomically: cached data is returned if available, an API timeout is logged exactly once per failure sequence, the system generates a consistent placeholder rating, and a monitoring alert is triggered for operators. These behaviours collectively ensure graceful handling of service degradation.

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

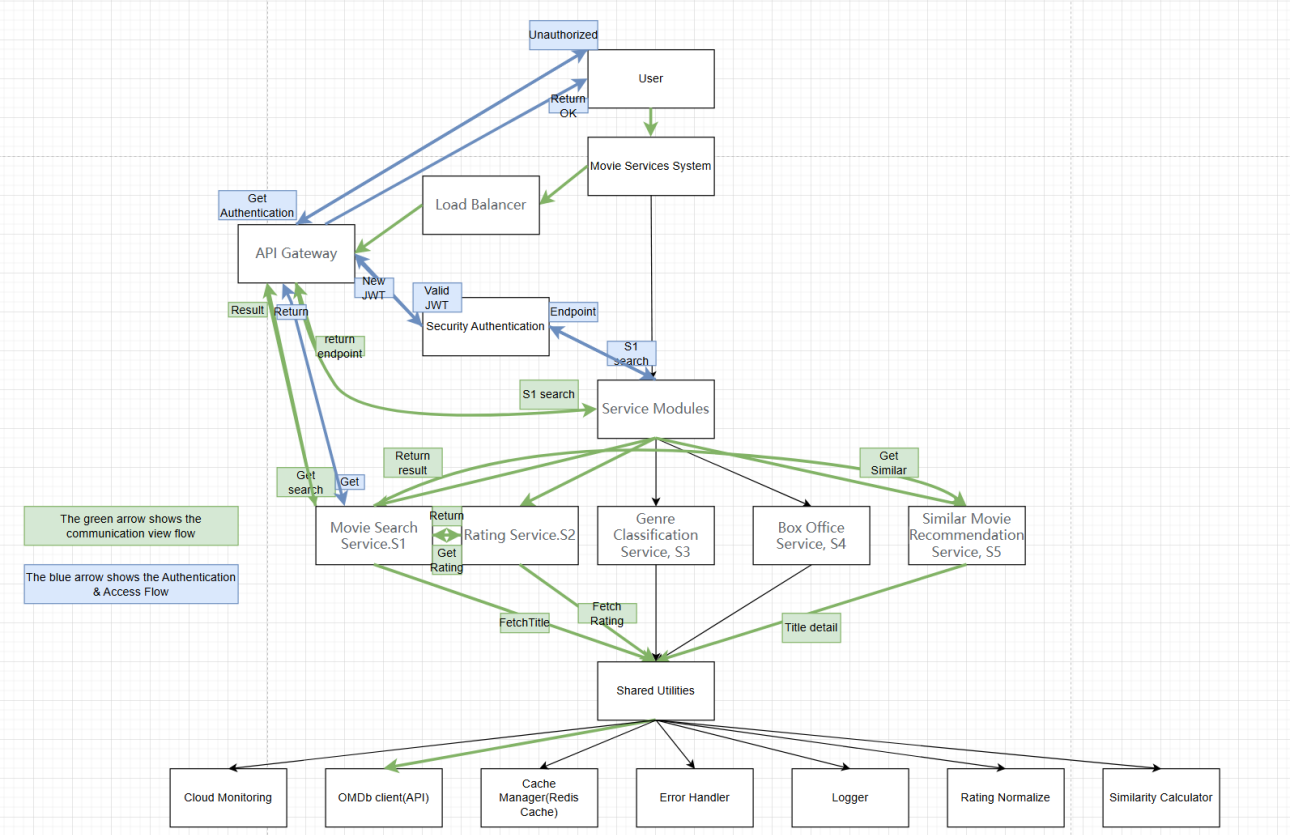
In this view, the primary sources of variability include the timeout threshold, the number of retry attempts, and the fallback strategy. The timeout window is currently set to 400 ms but can be raised or lowered based on future performance requirements. The retry count is fixed at one attempt, but the architecture allows this value to be altered to support more robust or more conservative error-recovery strategies. The fallback mechanism currently relies on cached data and a placeholder response, but it could be extended to incorporate circuit breakers, service isolation, or progressive backoff strategies. These variation points allow the system's fault-tolerance behaviour to adapt to evolving operational constraints.

SECTION 5. RATIONALE

The design decisions in this view support system reliability by ensuring service failure does not cascade into complete functionality loss. The timeout-driven branching provides predictable responsiveness, preventing S1 from blocking indefinitely. A single retry balances recovery likelihood with performance overhead, avoiding excessive stress on an already failing service. The use of cached fallback data maintains partial functionality, while placeholder responses preserve user experience when no data is available. Logging and monitoring integration ensures failures are both observable and traceable, enabling timely operator intervention. This approach reinforces robustness by incorporating structured recovery mechanisms, preventing unexpected service degradation, and ensuring the system continues to operate even when dependencies fail.

Combined View

SECTION 1. PRIMARY PRESENTATION



Combined View for basic structure, Service Discovery & Data Flow and Authentication & Access Flow.

This diagram shows the structure of the system. In addition, it includes the authentication process and the process for normal user access. This architectural design significantly reflects a prioritization of the three key quality attributes: security, availability, and performance.

Regarding security, the centralized verification mechanism of JWT tokens not only ensures the enforcement of a unified authentication policy but also demonstrates a deep consideration for confidentiality and integrity requirements.

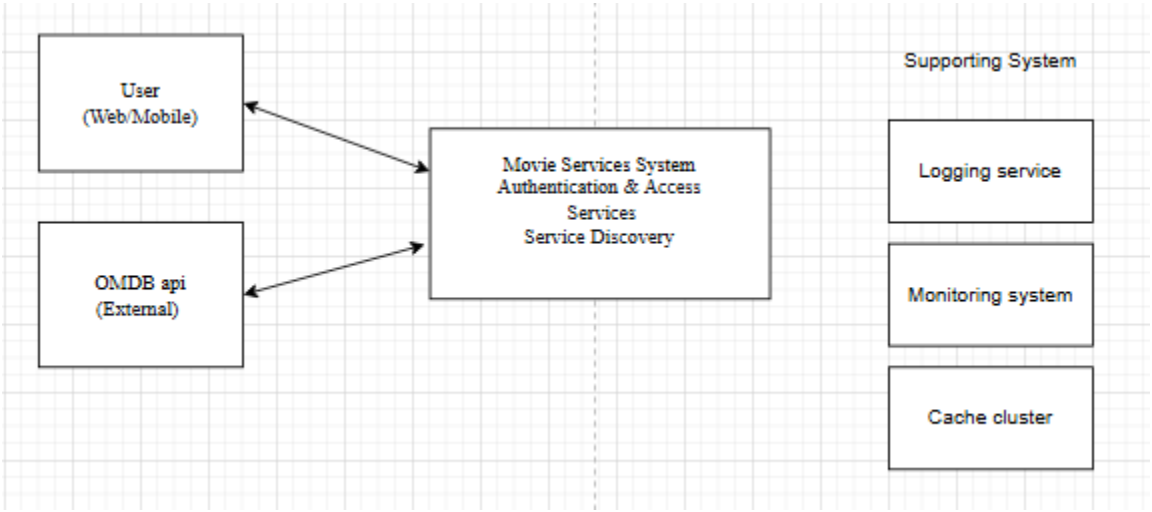
Regarding availability, the loosely coupled design between architectural components enhances the system's modifiability, ensuring that updates or replacements of individual services do not impact the overall system's stability.

From a performance perspective, this design avoids redundant authentication overhead and optimizes end-to-end response time.

SECTION 2. ELEMENT CATALOG

Element	Properties
Client	HTTP client, JWT token storage, request retry logic
Load Balancer	SSL termination, health checking, round-robin algorithm
API Gateway	Request routing, rate limiting, JWT validation coordination
Authentication Service	JWT validation, token refresh, session management
Service Registry	Service registration, health monitoring, endpoint resolution
Movie Search Service (S1)	Request coordination, data aggregation, error handling
Rating Summary Service (S2)	Data processing, 400ms timeout threshold, fallback capability
Genre Classification Service (S3)	Data processing
Box Office Service (S4)	Top-movie recommendations, ranking logic
Similar Movies Service (S5)	Algorithmic similarity processing, high throughput (1000 req/s)
External OMDb API	External dependency, variable latency

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

Authentication Methods: Currently JWT-based, can extend to OAuth2, SAML, or API keys

Service Discovery: Currently centralized registry, can switch to client-side discovery or DNS-based

Load Balancing Algorithm: Currently round-robin, configurable to least-connections or IP hash

Timeout Configuration: S2 timeout adjustable from 400ms based on performance requirements

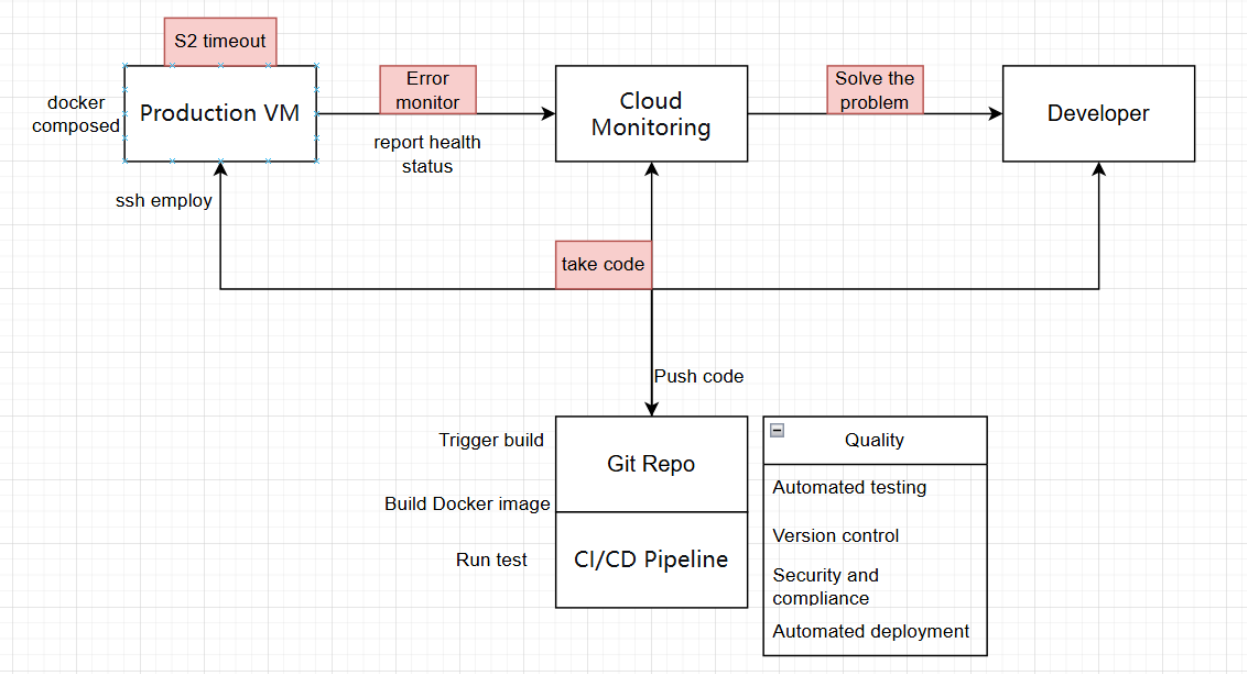
Fallback Strategies: Multiple fallback options (cache, default values, partial responses)

SECTION 5. RATIONALE

This architecture employs a layered security approach where authentication occurs before service discovery to prevent unauthorized access to internal service endpoints. The API Gateway pattern centralizes cross-cutting concerns like authentication and rate limiting, promoting separation of concerns. Service discovery through a dedicated registry enables dynamic scaling and fault tolerance.

The design prioritizes security by validating identity at the earliest possible point while maintaining performance through parallel service calls. The 400ms timeout for S2 represents a trade-off between user experience and data accuracy, ensuring the system remains responsive even when dependent services experience latency.

SECTION 1. PRIMARY PRESENTATION



Combined view for S2 Timeout and Recovery and Deployment Workflow

This view demonstrates a deep integration of maintainability, testability, and observability. By combining the S2 error handling mechanism with a complete DevOps pipeline, the system achieves an automated closed loop from error detection to repair.

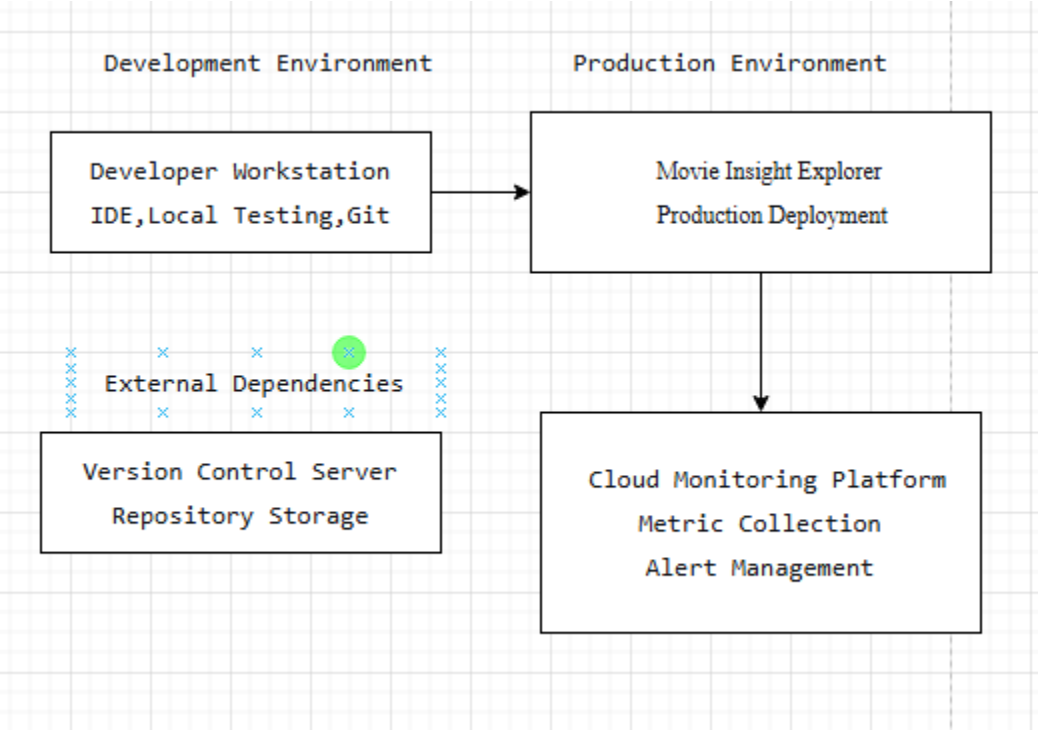
In terms of observability, the tight coupling between Cloud Monitoring and error handling logic provides a real-time view of system health, enabling the system to dynamically adjust its handling strategies based on actual runtime error patterns.

From a testability perspective, the error handling test suite and chaos engineering practices integrated into the architecture ensure the reliability of the degradation strategy under various failure scenarios, balancing the trade-off between release speed and system stability.

SECTION 2. ELEMENT CATALOG

Element / Relation	Properties
Developer	Code authoring, testing, deployment initiation
Git Repository	Branch management, merge requests, code history
CI/CD Pipeline	Quality gates, automated testing, deployment orchestration
Production VM	Error handling logic, monitoring agent, fallback mechanisms
Cloud Monitoring	Metric collection, alert rules, dashboard visualization
Developer → Git Repo	Code commit and push; version-controlled, peer-reviewed, branch-protected
Git Repo → CI/CD Pipeline	Webhook trigger; automated, event-driven pipeline initiation
CI/CD Pipeline → Production VM	Automated deployment; blue-green releases, health checks, rollback capability
Production VM → Cloud Monitoring	Metric streaming; real-time, high-frequency, comprehensive telemetry
Cloud Monitoring → Developer	Alert notifications; multi-channel (email, Slack, PagerDuty), priority-based

SECTION 3. CONTEXT DIAGRAM



SECTION 4. VARIABILITY GUIDE

- CI/CD Tooling: Currently integrated with Jenkins, can migrate to GitLab CI, GitHub Actions, or Azure DevOps
- Monitoring Platform: Currently using Cloud Monitoring, compatible with Prometheus, Datadog, or New Relic
- Deployment Strategy: Currently blue-green deployment, support for canary releases or feature flags
- Alert Channels: Configurable notification targets (Slack, email, SMS, PagerDuty)

SECTION 5. RATIONALE

This combined view integrates development workflows with production error handling to create a closed-loop feedback system. The architecture emphasizes rapid detection and resolution of S2 timeout issues through automated monitoring and developer notification.

The design rationale centers on DevOps principles of continuous improvement and blameless postmortems. By connecting error capture directly to developer workflows, the

system ensures that production issues are quickly addressed with code-level fixes rather than temporary operational workarounds.

The quality gates in the CI/CD pipeline prevent regression of error handling capabilities, while the comprehensive monitoring provides data-driven insights for optimizing S2 performance over time. This approach balances deployment velocity with system reliability, enabling frequent updates while maintaining production stability.

System Overview

The goal of MovieDB+ is to provide users with an interactive and data-driven platform to discover, compare, and analyze movies through ratings, genres, and trends.

The MovieDB+ is a microservice-based web app that allows users to explore and analyze movie data through modular services. The system is composed of 5 core services provided for the user: Movie Search (S1), Rating Summary (S2), Genre Classification (S3), Top Box Office Insights (S4), and Similar Movie Recommendations (S5). Each service will operate as an independent RESTful module, with API endpoints for interactions through internal gateways that share a common caching layer.

At a high level, the system follows a service-oriented architecture (SOA) approach, deployed in a cloud environment using Docker containers, with planned CI/CD automation supported by GitHub Actions. External movie data will be retrieved through third-party APIs (such as OMDb), processed by internal services, and delivered to a lightweight front-end interface. JSON is used for data interchanges between components. Users will interact with the system through a web interface that communicates with the API gateway, which coordinates and aggregates service responses.

The three key architectural views defined above:

Structural View: captures the system decomposition into 5 core microservices, Movie Search, Rating Summary, Genre Classification, Top Box Office Insights, and Similar Movie Recommendations. Alongside these services there is the API gateway, c aching layer, and external OMDb API. Each service runs as an independent container and communicates via RESTful calls, enabling clear boundaries, modularity, and fault isolation.

Quality View: highlights tactics for achieving high availability, scalability, and performance. Caching and asynchronous requests reduce response times, autoscaling ensures consistent performance during peak load, and redudnacy in service deployments supports 99% uptime. Secuirty and maintainability are reinforced through input validation, secrets management, and centralized logging.

Combined View: integrates the structural and quality perspectives by linking the system's decomposition with team responsibilities and runtime behavior. It demonstrates how independent services interact through the gateway, handle user authentication, and recover gracefully from failures. This view also maps each subsystem to group member ownership, ensuring clarity in design accountability and test coverage.