# Software Engineering Project 3

# UUUTorrent


## Technical Report

---

## BEST SE TEAM EVER
## (UNNAMED UNARMED UNTAMED)

Sambu Aneesh

Vidhi Rathore

Bassam Adnan

Moida Praneeth Jain

Aadi Deshmukh

# Contents

# 1 Requirements and Subsystems

## 1.1 Requirements

### 1.1.1 Functional Requirements (FR)

- **FR1: Remote Torrent Download & Transfer:** Utilize an Oracle Cloud compute instance with qBittorrent for downloading torrents. Provide authenticated users a mechanism (via TUI) to initiate downloads (directly via magnet links or indirectly via watchlist selection). Completed files reside on the server; users must **independently** use tools like SCP/SFTP to transfer files to their local devices (transfer is not managed by this application).
- **FR2: User and Access Management:** Implement user authentication (signup, login/password) and authorization. Define distinct roles: 'User' (manages personal downloads, interacts with watchlist) and 'Admin' (implied capabilities for monitoring access; specific user management endpoints beyond signup are not implemented in the current prototype but the role structure exists).
- **FR3: Watchlist-Driven Torrent Search:** The backend integrates with the Anilist API to fetch a user's 'CURRENT' watchlist. Based on user selection of an unwatched item in the TUI, the backend performs a targeted search against Nyaa.si's RSS feed using item details (title, episode number, preferred quality) to find relevant torrents and initiate downloads via qBittorrent's Web API.
- **FR4: Watchlist Integration & Display:** Allow users (via TUI) to view their 'CURRENT' media watchlist fetched from Anilist. The TUI highlights items with available next episodes, enabling users to trigger the download search process (FR3) or update progress on Anilist. Includes functionality to link their Anilist account via an access token.
- **FR5: Torrent Lifecycle Management:** Enable users (via TUI) to view the status (name, progress, status string from qBit, seeds/leechs) of **their** torrents managed by the system, and to pause, resume, and delete these torrents (optionally deleting files on the server).
- **FR6: Admin Monitoring View:** Provide administrators access to a web-based dashboard (Grafana, external to this codebase but part of the intended deployment) displaying real-time server resource usage (CPU, RAM, disk, network via node_exporter), qBittorrent statistics (if an exporter is configured), database health (via pgexporter), and API backend metrics (exposed via `/metrics`).
- **FR7: User Interface (TUI):** Provide a cross-platform Terminal User Interface (built with Textual) for user interaction: displaying torrent status, viewing the Anilist watchlist, triggering downloads for watchlist items, updating Anilist progress, adding torrents directly (magnet links not shown in current TUI prototype but API exists), managing torrents, and authenticating. Runs on the user's machine.

### 1.1.2 Non-functional Requirements (NFR)

- **NF1: Performance (Watchlist Search Latency):** The time taken from user selection of a watchlist item in the TUI to the download being successfully initiated in qBittorrent (including

backend search, RSS fetch/parse, and qBittorrent API call) should be < 20 seconds under normal load and typical external service responsiveness.

- **NF2: Performance (Concurrency):** Support at least 50 concurrent torrent downloads/seeds managed by qBittorrent without significant API/TUI responsiveness degradation from the **backend's perspective**. (Actual network throughput depends on OCI limits and peer availability).
- **NF3: Reliability (Backup & Recovery):** Perform automated daily database backups (PostgreSQL). RTO: 1 hour. RPO: 24 hours. (Backup mechanism is external to the application code).
- **NF4: Security (Authentication):** Secure user authentication (bcrypt hashed+salted passwords). Protect API endpoints with token-based authentication (JWT). Handle external service (Anilist) authentication token storage securely within the database.
- **NF5: Security (Authorization):** Enforce authorization rules: users manage only their own linked torrents and interact with their own Anilist data; admins have specific elevated privileges (e.g., accessing `/torrents/all`, implied access to monitoring).
- **NF6: Usability (TUI):** The TUI should be intuitive for terminal users, providing clear feedback (e.g., download progress, status messages), navigation between views (watchlist/torrents), and status updates for asynchronous operations like downloads.
- **NF7: Resource Constraints (OCI):** Operate within OCI Always Free tier limits (Ampere A1 CPU/RAM, storage, network egress). Monitor usage via the Monitoring Service (FR6). The on-demand search approach (ADR-001) is key to minimizing idle resource usage.
- **NF8: Maintainability:** Ensure modular, documented backend code (using type hints, FastAPI features, separation of concerns) with clear abstractions (like repositories) for easier updates and debugging.
- **NF9: Monitoring Availability & Freshness:** The Admin monitoring view (Grafana) should be highly available (best effort on free tier) and display metrics scraped by Prometheus with a latency of <= 1 minute from collection time. The API exposes metrics via `/metrics`.
- **NF10: External Service Dependency:** The system's watchlist functionality relies critically on the availability and performance of the Anilist API and the Nyaa.si RSS feed. The system handles errors from these services by raising appropriate HTTP exceptions to the TUI.

#### 1.1.3 Architecturally Significant Requirements (ASR)

The following requirements have the most significant impact on the architectural design, driving key decisions about structure, technology, and process:

- **FR3 & FR4 (Watchlist Integration/Search):** These core functional requirements dictate the primary user workflow, requiring specific interactions with external APIs (Anilist, Nyaa), influencing the API backend logic (`anilist_service`, `nyaa_service`, `torrent_orchestration_service`), database schema (`AnilistToken`, `UserTorrentLink`),

and TUI design (`card.py`, `api_client.py`). The choice for on-demand search (ADR-001) is a direct consequence.
- **NF1 (Watchlist Search Latency) & NF2 (Concurrency):** These performance requirements drive the need for an asynchronous backend (FastAPI, `httpx` for async requests), efficient external API interaction (dedicated services), optimized search/parsing logic (though Nyaa parsing is simple), and careful consideration of qBittorrent API usage (`run_in_executor` for synchronous qBit library) to avoid blocking and ensure responsiveness under load.
- **NF3 (Reliability):** Dictates the need for a robust database (PostgreSQL) and specific operational procedures like automated backups and a recovery plan, impacting deployment and maintenance processes (external to code).
- **NF4 & NF5 (Security):** Fundamental requirements demanding careful design of authentication (bcrypt hashing in `security.py`, JWT creation/validation in `security.py`, `/auth/token` endpoint) and authorization (role checks via `CurrentAdminUser` dependency, resource ownership checks via `verify_torrent_ownership` dependency) mechanisms throughout the API backend. Impacts data storage (`User`, `AnilistToken` models) and API endpoint design (`deps.py`).
- **NF7 (Resource Constraints):** The OCI Always Free tier limitation is a primary driver for technology choices (Python/FastAPI/Textual are lightweight), architectural patterns (ADR-001: on-demand processing), and necessitates resource monitoring (FR6, NF9).
- **NF8 (Maintainability):** Influences the adoption of modular design (separation into `api`, `core`, `db`, `schemas`, `services`), separation of concerns (Subsystem Overview), clear API contracts (FastAPI schemas), and implementation patterns (Repository, DI) to facilitate future development and debugging.
- **NF9 (Monitoring):** Requires incorporating a dedicated monitoring stack (Prometheus/Grafana/Exporters - external setup) and instrumenting the API backend (`prometheus-fastapi-instrumentator`) and potentially other components (DB via pgexporter, OS via node_exporter) to expose relevant metrics via the `/metrics` endpoint.
- **NF10 (External Dependency):** Necessitates designing the API backend (`anilist_service`, `nyaa_service`) and TUI (`api_client.py`) to be resilient to failures or delays in external services (Anilist, Nyaa), requiring error handling (HTTPExceptions, try-except blocks) and timeouts (`httpx` timeout parameter).

## 1.2 Subsystem Overview

The system is decomposed into the following major subsystems and external dependencies:

### 1.2.1 User Interface (TUI - Textual):

#### 1.2.1.1 Role & Functionality

The client-side application (`frontend/`) running on the user's machine. Provides views for torrent status, Anilist watchlist display (highlighting unwatched, showing progress), and mechanisms to trigger watchlist downloads, update Anilist progress, manage torrent lifecycle (pause/resume/delete), authenticate, link Anilist account, and (theoretically) get transfer info. It focuses on presenting information and capturing user intent.

#### 1.2.1.2 Interaction

Communicates exclusively with the API Backend via HTTPS RESTful API calls (using `httpx` in `api_client.py`).

### 1.2.2 API Backend (FastAPI):

#### 1.2.2.1 Role & Functionality

The central coordinating service (`backend/app/`) running on the OCI instance. Handles API requests from the TUI, manages user authentication/authorization (using JWT, bcrypt), securely stores and retrieves Anilist access tokens, interacts with Anilist API (`anilist_service.py`) to fetch watchlists/update progress, performs on-demand searches against Nyaa.si RSS (`nyaa_service.py`), orchestrates torrent management via qBittorrent's Web API (`qbittorrent_service.py`, `torrent_orchestration_service.py`), interacts with the database (`db/`), and exposes metrics (`/metrics` endpoint in `main.py`) for monitoring.

#### 1.2.2.2 Interaction

- Listens to TUI requests (`api/endpoints/`).
- Calls qBittorrent Web API (`qbittorrent_service.py` using `python-qbittorrentapi`).
- Queries/updates PostgreSQL DB (`db/repository/` using SQLAlchemy async).
- Calls external Anilist API (`anilist_service.py` using `httpx`).
- Calls external Nyaa.si RSS Search (`nyaa_service.py` using `httpx`, `feedparser`).
- Scraped by Prometheus (via `/metrics`).

### 1.2.3 Torrent Client Service (qBittorrent):

#### 1.2.3.1 Role & Functionality

The dedicated BitTorrent client running on the OCI instance (external process). Manages the low-level details of torrent downloading, uploading (seeding), peer/tracker communication, and file storage based on instructions received from the API Backend via its Web API.

#### 1.2.3.2 Interaction

- Exposes a Web API for control by the API Backend (`qbittorrent_service.py`).
- Communicates with BitTorrent network (trackers/peers).
- Reads/writes files to OCI storage.

### 1.2.4 Database (PostgreSQL)

#### 1.2.4.1 Role & Functionality

The persistent storage layer running on the OCI instance (external process). Stores user credentials (`User` model: hashed passwords, roles), mappings between users and torrents (`UserTorrentLink` model: hashes), and securely stored user-specific Anilist API tokens (`AnilistToken` model).

#### 1.2.4.2 Interaction

- Responds to asynchronous SQL queries from the API Backend (`db/repository/` via SQLAlchemy).

- Scraped for metrics by pgexporter (external monitoring setup).
- Periodic automated backups via pg-backup service (cron-based) to satisfy NF3 (Reliability).

#### 1.2.5 Monitoring Service (Prometheus + Grafana + Exporters):

##### 1.2.5.1 Role & Functionality

The operational monitoring stack running on the OCI instance (external processes). Prometheus scrapes metrics from multiple sources: system metrics (CPU, memory, disk, network) via node_exporter, database performance metrics via pgexporter, and application metrics via the API Backend's `/metrics` endpoint. Grafana provides web-based dashboards for administrators to visualize system health, resource usage, database performance, and application behavior in real-time (FR6).

##### 1.2.5.2 Interaction

- Prometheus scrapes targets at configured intervals:
  - System metrics via node_exporter (CPU, memory, disk, network utilization)
  - Database metrics via pgexporter (query performance, connections, table statistics)
  - API Backend metrics via the `/metrics` endpoint
- Grafana queries Prometheus and renders customized dashboards for administrators
- All components run as Docker containers on the OCI instance

#### 1.2.6 Watchlist API (External - e.g., Anilist):

##### 1.2.6.1 Role & Functionality

A third-party service (Anilist) providing media information and user watchlist management via a GraphQL API. Acts as the authoritative source for the user's watchlist content and progress.

##### 1.2.6.2 Interaction

Receives authorized GraphQL API requests (via HTTPS POST) from the API Backend (`anilist_service.py`).

#### 1.2.7 RSS Search Source (External - e.g., Nyaa.si):

##### 1.2.7.1 Role & Functionality

A third-party torrent indexer (Nyaa.si) providing search results via RSS feeds. Used by the API Backend on-demand (`nyaa_service.py`) to find specific torrents matching watchlist items.

##### 1.2.7.2 Interaction

Receives HTTP GET requests (with search terms) from the API Backend; returns RSS (XML) data, parsed using `feedparser`.

# 2 Architecture Framework

## 2.1 Stakeholder Identification (IEEE 42010)

Following the IEEE 42010 standard, we identify stakeholders, their concerns, and the architectural views/viewpoints addressing them:

### 2.1.1 Stakeholders

- **End User:** Individuals using the TUI to manage and download torrents, primarily based on their Anilist media watchlist.
- **Administrator:** Personnel responsible for maintaining the OCI instance, monitoring system health, ensuring service availability, and potentially managing users (though user management features are minimal in the prototype).
- **Developer:** The team building and maintaining the UUUTorrent software (BEST SE TEAM EVER).
- **Oracle Cloud Infrastructure (OCI):** Provider of the underlying compute, storage, and network resources (implicitly a stakeholder due to resource constraints outlined in NF7).

### 2.1.2 Concerns

- **End User Concerns:**
  - C1: Ease of finding and downloading desired unwatched anime from Anilist watchlist (FR3, FR4).
  - C2: Reliability and speed of downloads (dependent on qBit/peers, but API/TUI should be responsive - NF1, NF2).
  - C3: Security of user account and Anilist integration (NF4).
  - C4: Clarity of torrent status and download progress in the TUI (FR5, NF6).
  - C5: Ability to manage torrents (pause, resume, delete) (FR5).
  - C6: Understanding how to transfer completed files (addressed by clarifying FR1).
- **Administrator Concerns:**
  - C7: System stability and availability (supported by NF3, NF9).
  - C8: Resource consumption within OCI free tier limits (NF7).
  - C9: Security of the overall system (preventing unauthorized access, data breaches) (NF4, NF5).
  - C10: Ease of monitoring system health and performance (FR6, NF9).
  - C11: Backup and recovery capabilities (NF3).
  - C12: Ease of user management (currently limited to signup - FR2).
- **Developer Concerns:**
  - C13: Maintainability and extensibility of the codebase (NF8).
  - C14: Testability of components (supported by patterns like Repository, DI).
  - C15: Clear separation of concerns between subsystems (Subsystem Overview).
  - C16: Feasibility of implementation within project constraints (addressed by tech choices, ADRs).
  - C17: Performance efficiency (NF1, NF2).
- **OCI Concerns:**
  - C18: Adherence to resource usage limits (CPU, RAM, network egress, storage) (NF7).

### 2.1.3 Viewpoints and Views

- **Logical Viewpoint:** Describes the functional decomposition of the system, components (FastAPI services, repositories, Textual widgets), and their interactions. Addresses C1, C4, C5, C13, C15, C16.
  - **Views:** Component Diagram (conceptual: TUI, API Backend, DB, qBit, Ext. Services), Sequence Diagrams (e.g., Watchlist Download Flow, User Login Flow), Subsystem Description (Task 1). Code structure (`backend/app/`, `frontend/`).

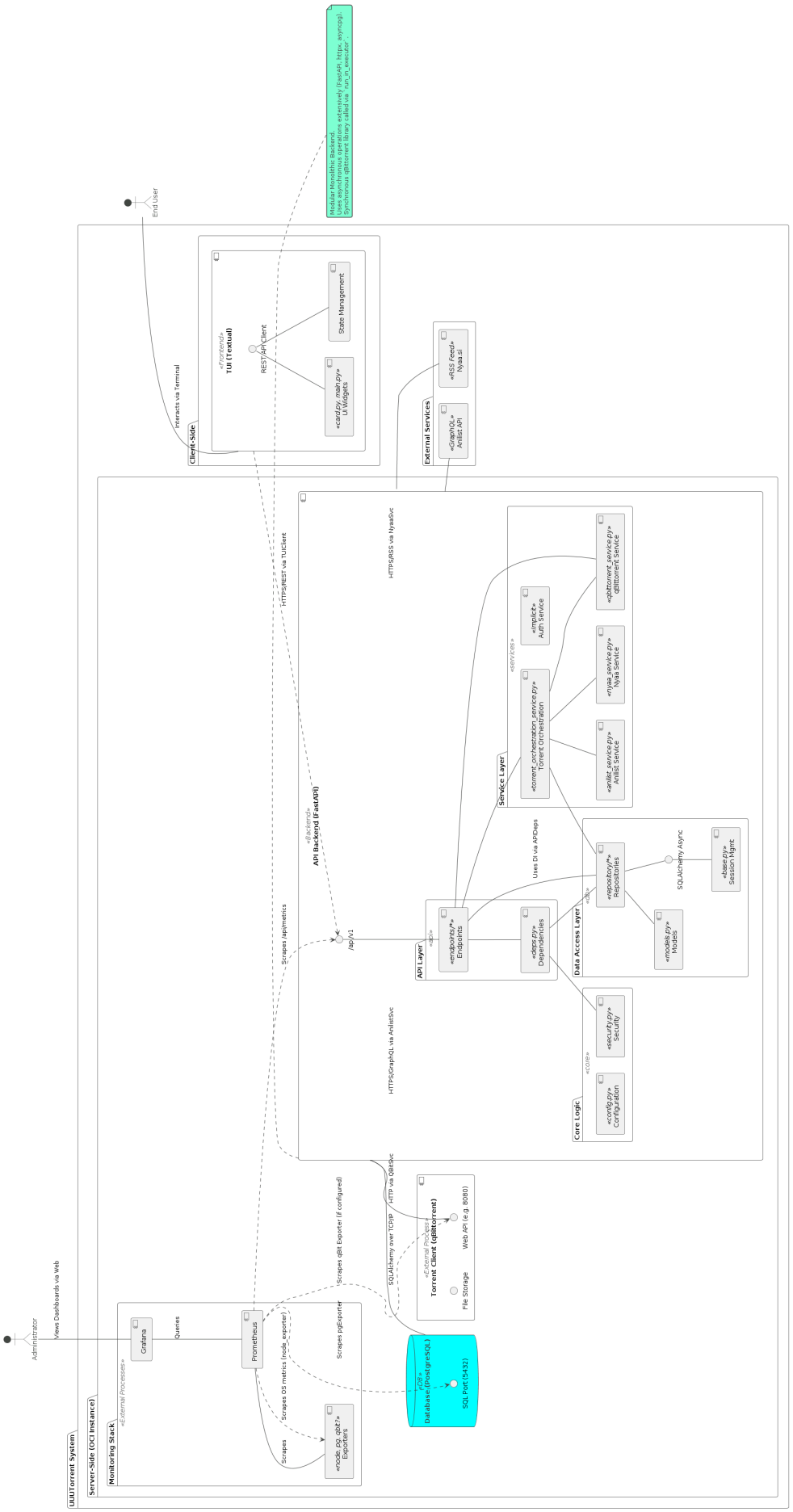


Figure 2: Logical View - Conceptual Component Diagram

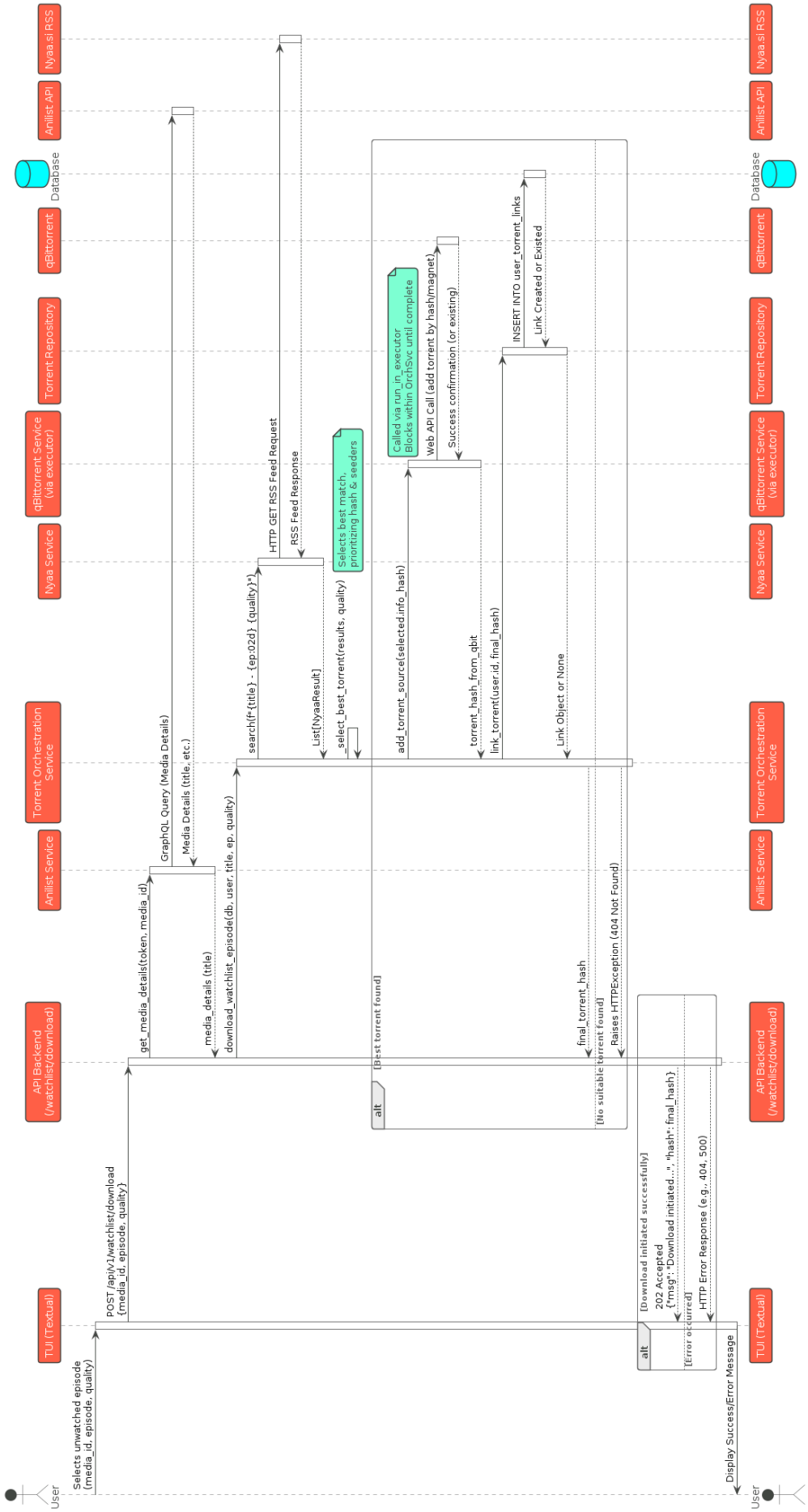


Figure 3: Logical View - Sequence Diagram: Watchlist Download Flow

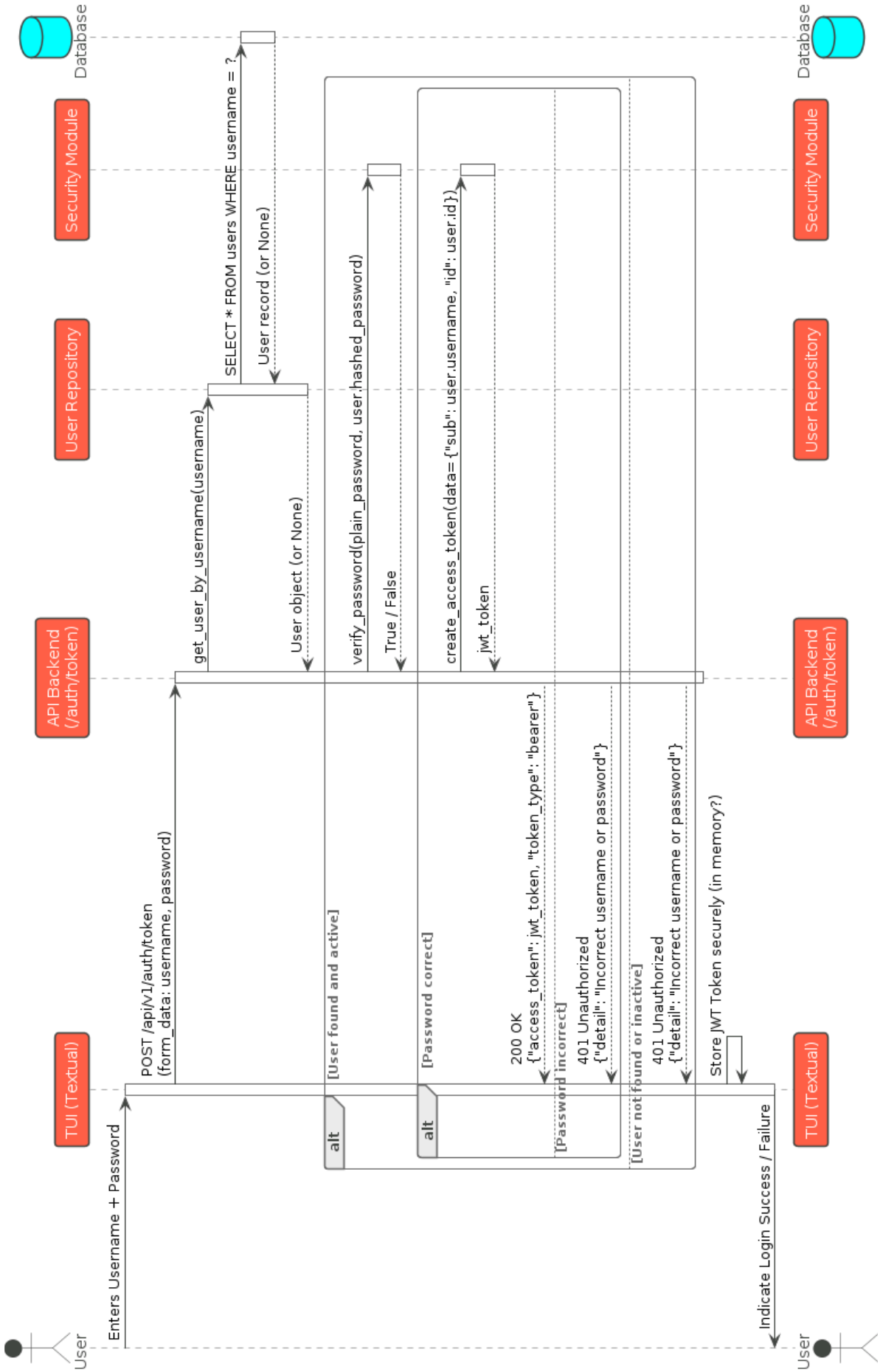


Figure 4: Logical View - Sequence Diagram: User Login Flow

- **Deployment Viewpoint:** Describes the physical environment (OCI instance), mapping of software components (Python processes, DB server, qBit process, Monitor processes) to hardware/infrastructure, and network interactions. Addresses C7, C8, C11, C18.
  - **Views:** Deployment Diagram (showing OCI instance, containers/processes for API, DB, qBit, Monitor), Network Diagram (ports: e.g., 8000 for API, 5432 for DB, qBit WebUI port, Prometheus/Grafana ports).

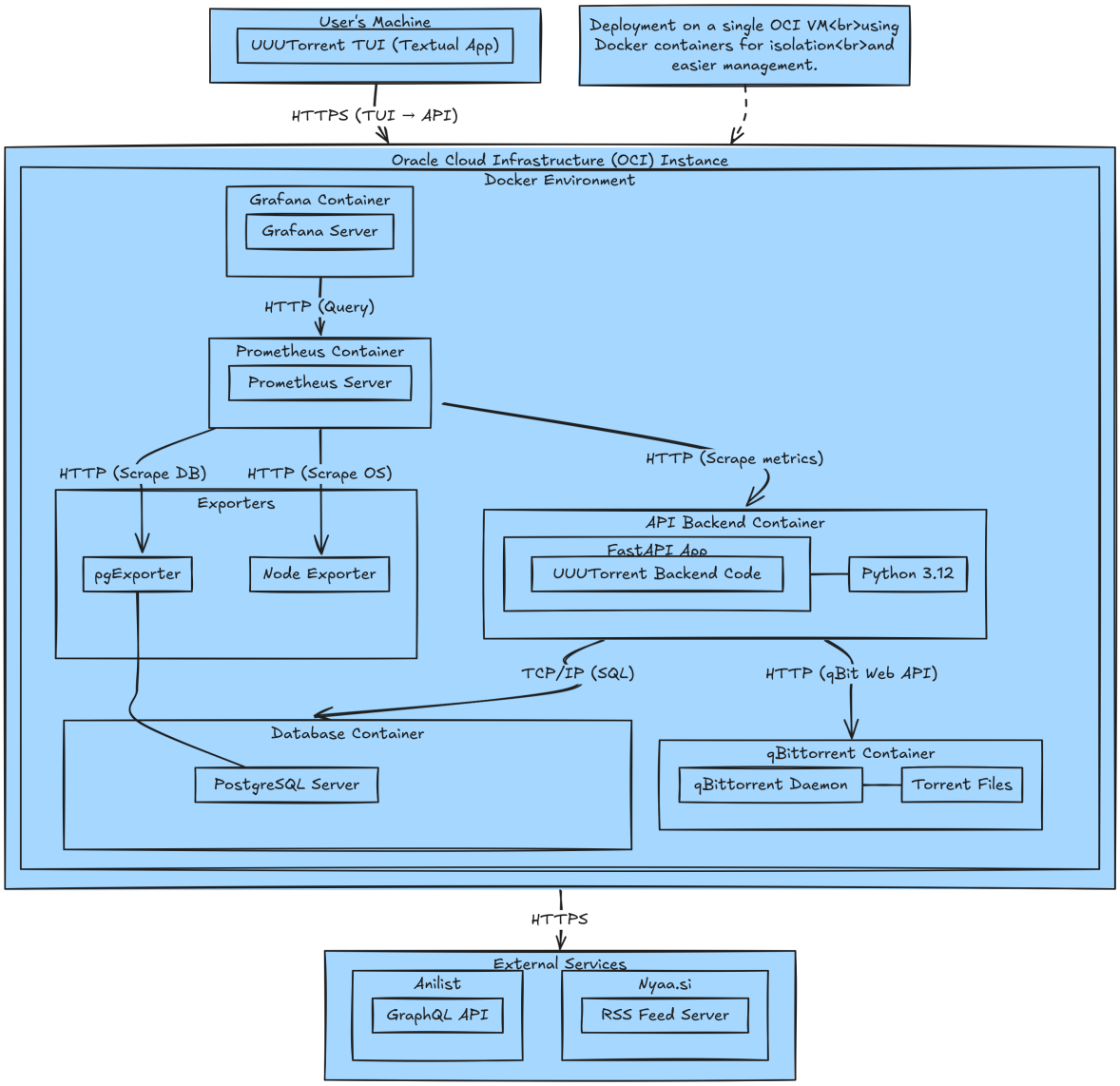


Figure 5: Deployment View - Deployment Diagram (OCI Instance with Docker Containers)

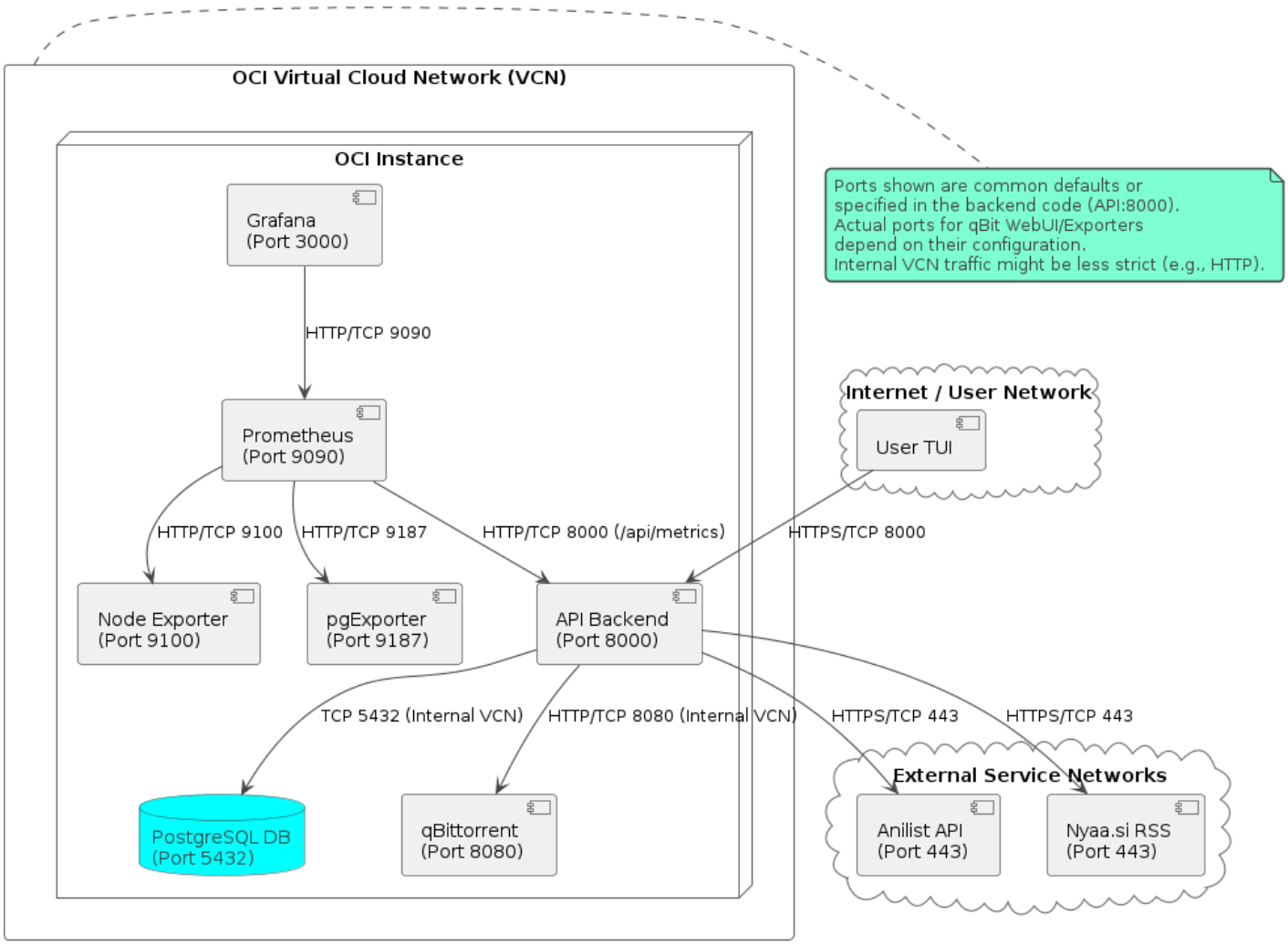


Figure 6: Deployment View - Network Diagram (Key Components and Ports)

- **Security Viewpoint:** Describes how security requirements are met, including authentication (JWT, password hashing), authorization (dependency checks), data protection (token storage), and external service integration security (Anilist token handling). Addresses C3, C9, NF4, NF5.
  - **Views:** Authentication Flow Diagram (Login -> JWT -> API Call), Access Control Description (User vs Admin roles, torrent ownership), Data Security Description (bcrypt for passwords, direct storage of Anilist token).


Figure 7: Security View - Authentication Flow Diagram (Login and Protected Route Access)

- **Operational Viewpoint:** Describes how the system is operated, monitored, and maintained. Addresses C7, C10, C11, C12, NF3, NF9.
  - **Views:** Monitoring Dashboard Description (Grafana dashboards for node_exporter and pgexporter metrics), Backup/Recovery Procedure Description (pg-backup container with cron job for daily PostgreSQL dumps), User Management (currently just signup).

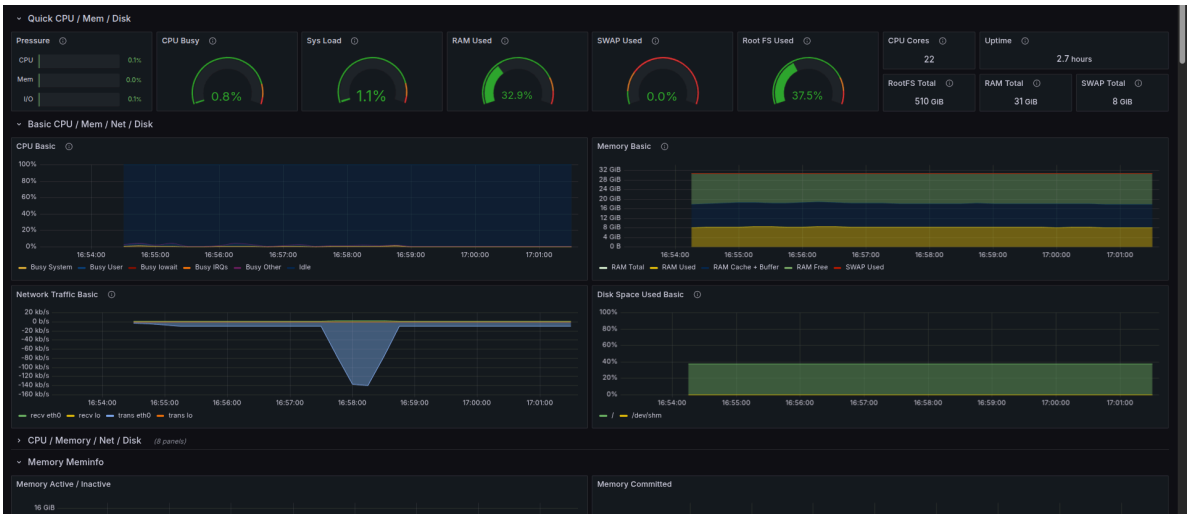


Figure 8: Operational View - Grafana Dashboard showing system metrics from node_exporter

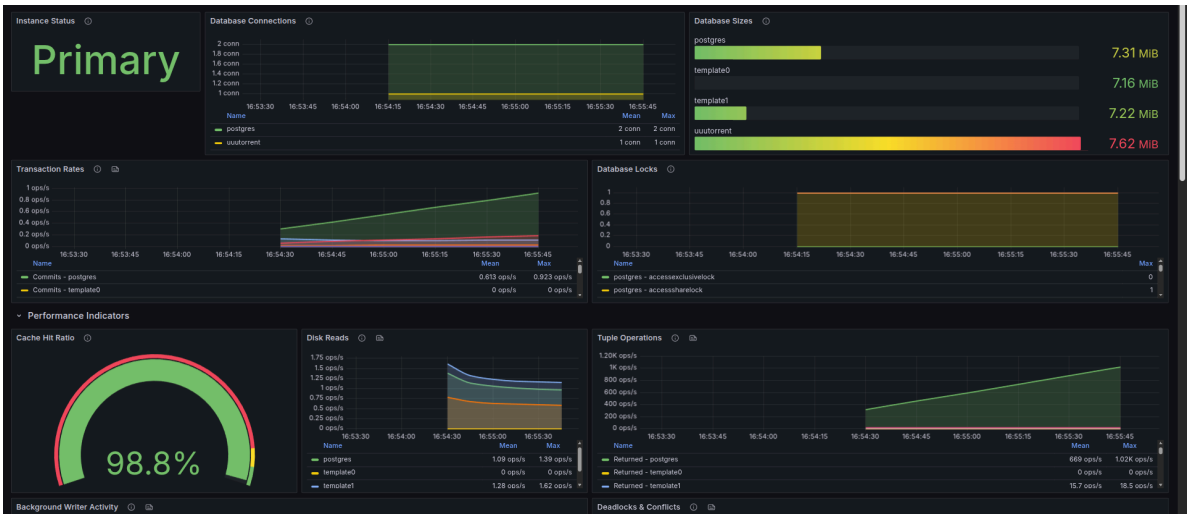


Figure 9: Operational View - Grafana Dashboard showing PostgreSQL metrics from pgexporter

- **User Experience (UX) Viewpoint:** Describes the user's interaction with the system via the TUI. Addresses C1, C4, C6, NF6.
  - **Views:** TUI Screenshots (`frontend/card.py` structure implies card-based layout), User Interaction Flow Descriptions (Login -> View Watchlist -> Select -> Download Trigger).

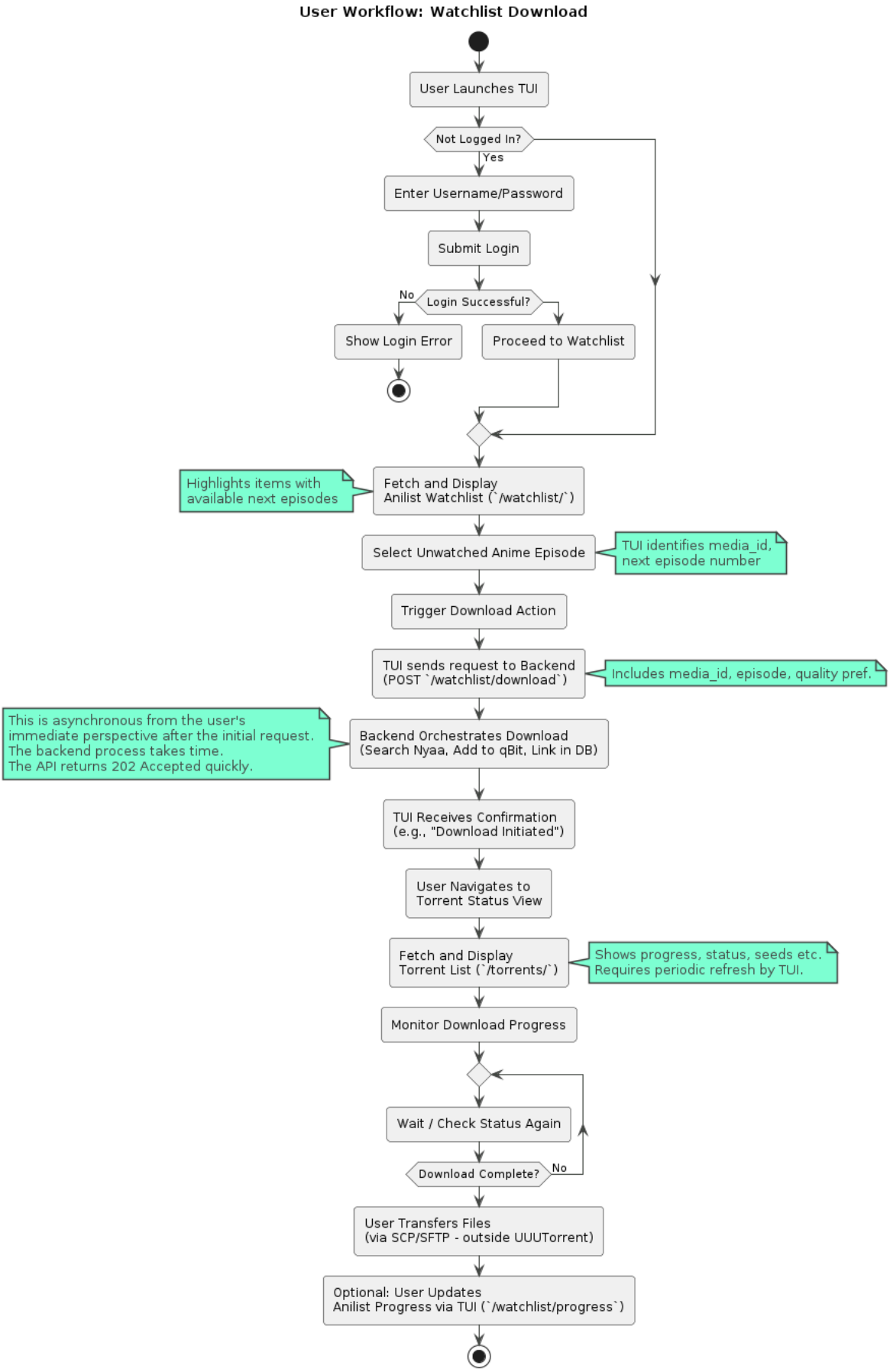


Figure 10: User Experience View - User Interaction Flow (Watchlist Download)

- **Development Viewpoint:** Describes aspects relevant to the development process, including code structure and patterns. Addresses C13, C14, C15, NF8.
  - **Views:** Code Structure Overview (`backend/app`, `frontend`), Description of Implementation Patterns (Task 3 - Repository, DI, etc.).

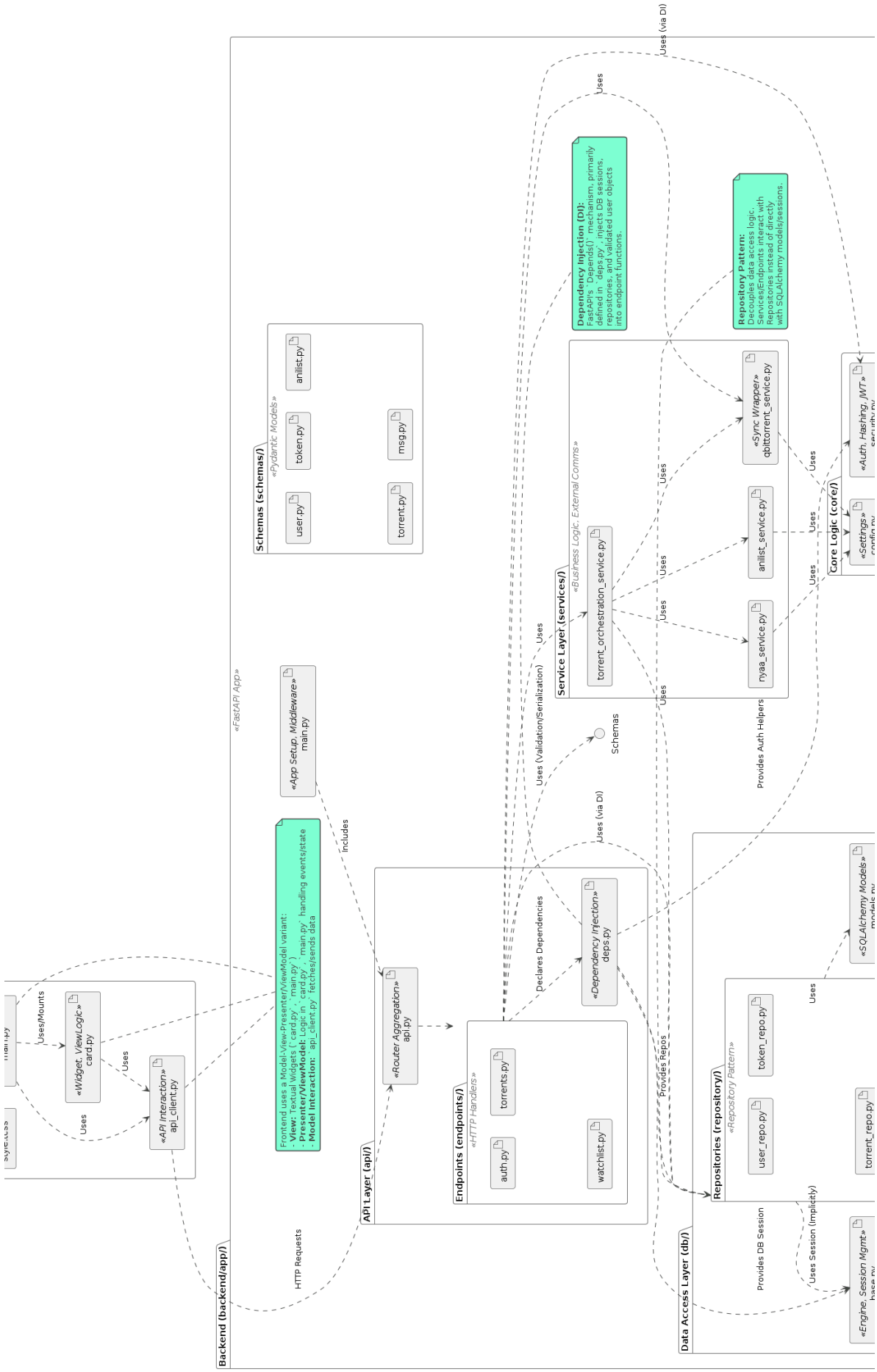


Figure 11: Development Viewpoint - Code Structure Overview and Implementation Patterns

The following figures illustrate the system architecture using the C4 model, progressing from System Context to Containers and Components.

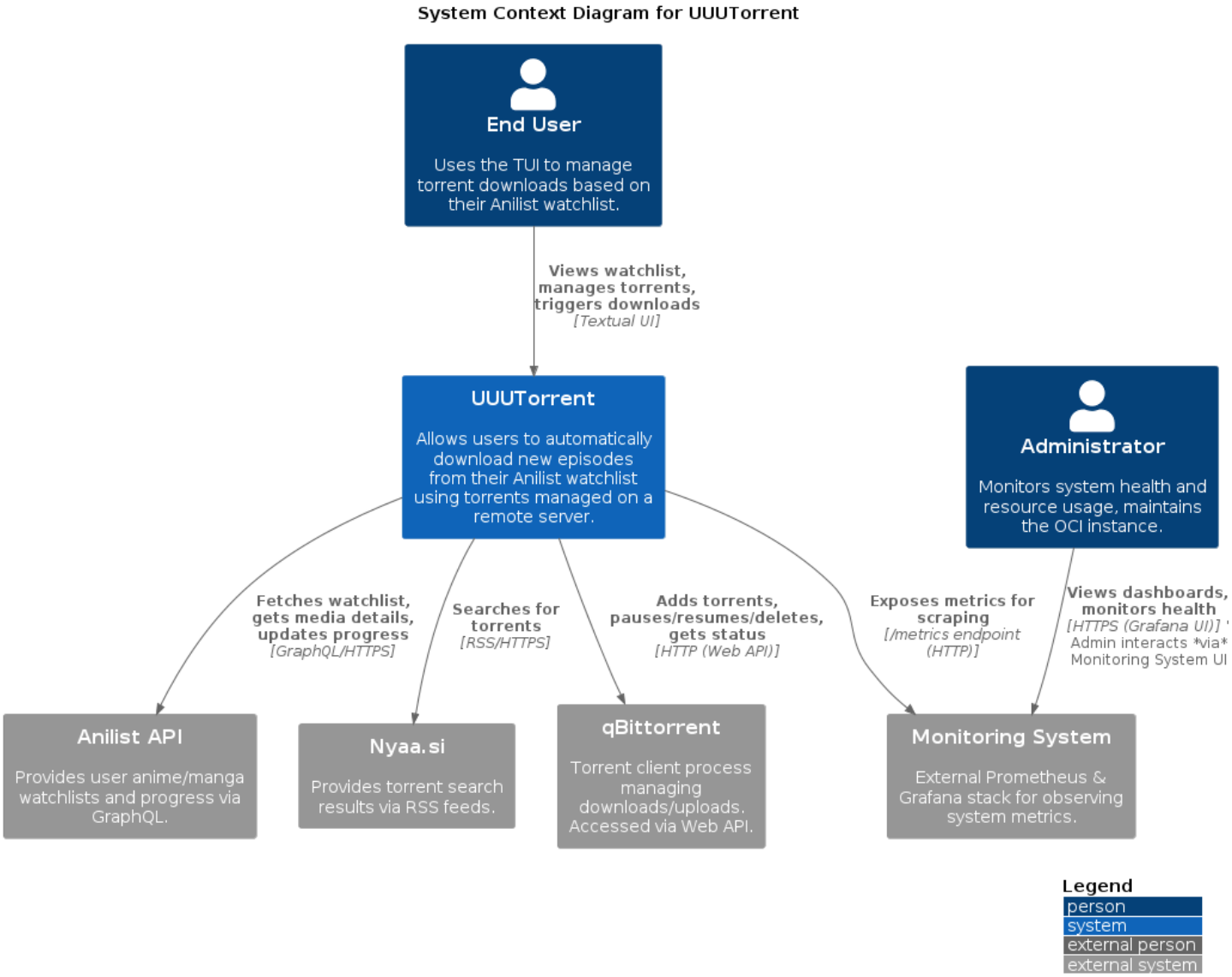


Figure 12: C4 Model - Level 1: System Context Diagram

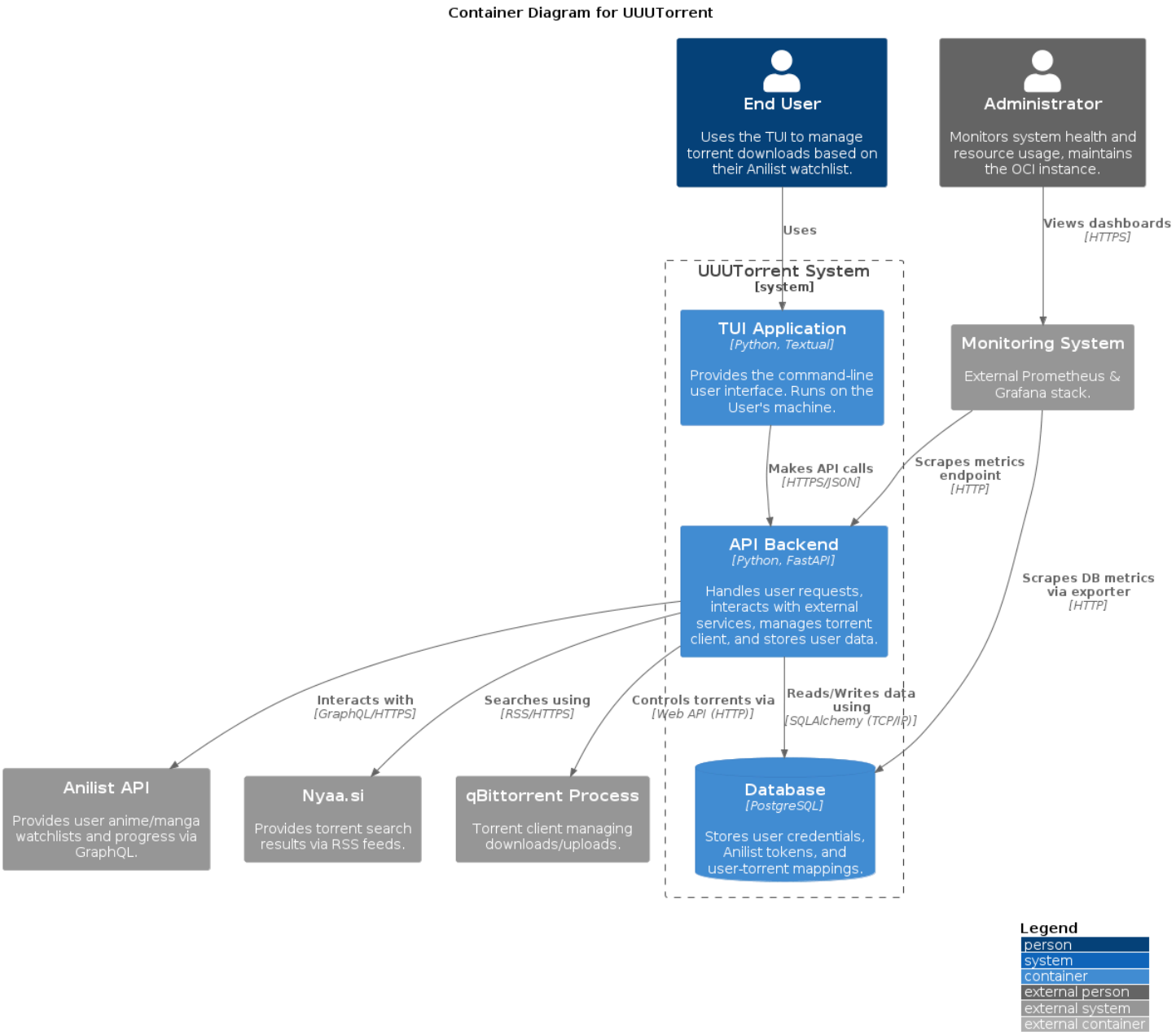


Figure 13: C4 Model - Level 2: Container Diagram

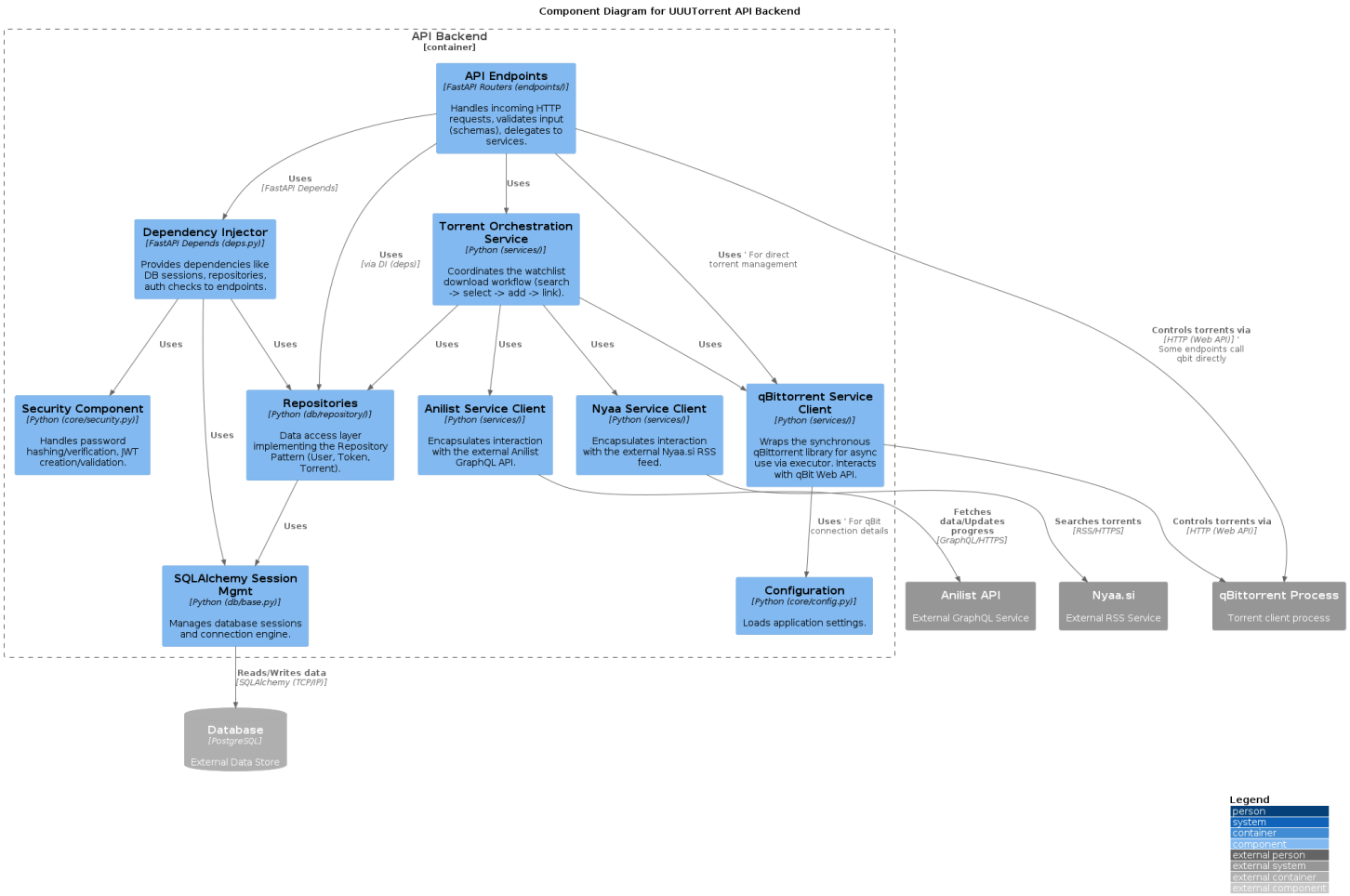


Figure 14: C4 Model - Level 3: Component Diagram (API Backend)

## 2.2 Major Design Decisions (ADRs)

### 2.2.1 ADR-001: On-Demand RSS Search for Watchlist Downloads

- **Status:** Accepted
- **Date:** 2025-11-04
- **Deciders:** BEST SE TEAM EVER
- **Context:** Need a mechanism to find torrents for unwatched Anilist watchlist items (FR3). Initial thoughts considered background RSS feed monitoring.
- **Decision Drivers:** Simplify backend logic, reduce constant resource consumption (critical for NF7 OCI Free Tier), provide a direct user-driven workflow (NF6), leverage existing external search capabilities (Nyaa.si), avoid complex state management for feeds.
- **Options Considered:**
  1. Background RSS Sync & Filtering: Continuously fetch/parse feeds, store items, match against watchlist. Complex, resource-intensive.
  2. On-Demand Watchlist-Driven RSS Search: User selects item in TUI -> Backend queries Nyaa.si RSS specifically for that item -> Backend parses results -> Add best match. Simple, low idle load.
  3. Hybrid: Some background caching, but primarily on-demand. Adds complexity.

- **Outcome:** Chose Option 2. Implemented in `torrent_orchestration_service.py` coordinating `anilist_service.py` and `nyaa_service.py`.
- **Consequences:**
  - **(+)** Significantly reduced backend complexity and idle resource usage (CPU, RAM, network, DB storage). Directly maps to user action. Faster initial development of core feature.
  - **(-)** Introduces latency during the user's download action (NF1). Creates a strong runtime dependency on Nyaa.si availability and responsiveness (NF10). Does not support discovering torrents unrelated to the Anilist watchlist via RSS.
- **Significance:** Defines the core download initiation workflow. Prioritizes resource efficiency and simplicity over proactive discovery or minimizing interactive latency. Aligns strongly with NF7.

#### 2.2.2 ADR-002: Core Technology Stack Selection

- **Status:** Accepted
- **Date:** 2025-03-25
- **Deciders:** BEST SE TEAM EVER
- **Context:** Need to choose primary technologies for the main subsystems (TUI, API Backend, Database) considering performance (NF1, NF2), maintainability (NF8), developer familiarity, and OCI constraints (NF7).
- **Decision Drivers:** OCI Free Tier compatibility (lightweight), Asynchronous capabilities for IO-bound tasks (FastAPI, Textual, httpx, asyncpg), Python ecosystem familiarity and rich libraries, modern TUI framework desired, need for relational data storage.
- **Options Considered:**
  - **API:** FastAPI (Python/Async - chosen), Flask (Python/Sync/Async), Node.js (Express/Koa - JS/Async).
  - **TUI:** Textual (Python - chosen), Rich (Python - lower level), ncurses (C - complex), Bubble Tea (Go).
  - **DB:** PostgreSQL (Relational - chosen), SQLite (File-based relational - harder for concurrent access/backup), MySQL (Relational), MongoDB (NoSQL - less suitable for user-token-torrent relations).
- **Outcome:** Chose FastAPI + Textual + PostgreSQL.
  - **FastAPI:** High performance async framework, built-in data validation (Pydantic), automatic OpenAPI docs, dependency injection. Addresses NF1, NF2, NF8.
  - **Textual:** Modern async TUI framework in Python, component-based, integrates well with async backend logic. Addresses FR7, NF6.
  - **PostgreSQL:** Mature, reliable RDBMS (NF3), handles relations and constraints well, excellent async driver (`asyncpg`).
- **Consequences:**
  - **(+)** Strong performance for IO-bound tasks, consistent language (Python) across TUI/Backend simplifies development, leverages modern Python features (async, type hints), good library support (SQLAlchemy, Pydantic, httpx).

  - **(-)** Requires developers comfortable with `asyncio`. Textual is newer than some TUI libs. PostgreSQL adds operational overhead compared to SQLite (but needed for robustness/concurrency).
- **Significance:** Defines the core development environment and runtime characteristics. Enables meeting performance and maintainability goals within resource constraints.

#### 2.2.3 ADR-003: Monitoring Stack Selection

- **Status:** Accepted
- **Date:** 2025-03-25
- **Deciders:** BEST SE TEAM EVER
- **Context:** Need a way for Administrators to monitor system health and resource usage (FR6, NF9) effectively within OCI constraints (NF7).
- **Decision Drivers:** Open-source, industry standard (Prometheus/Grafana), good visualization (Grafana), extensive ecosystem of exporters (node, postgres, potentially qBittorrent), pull-based model suitable for services exposing metrics endpoints (FastAPI), generally manageable resource footprint.
- **Options Considered:**
  - **Prometheus + Grafana:** Standard combination, wide adoption, many exporters. Chosen.
  - **ELK Stack (Elasticsearch, Logstash, Kibana):** Primarily for logs, can be resource-heavy.
  - **Datadog/New Relic:** SaaS, powerful but likely exceed free tier budget/limits.
  - **Netdata:** Real-time focus, potentially higher overhead for historical data/complex queries.
- **Outcome:** Chose Prometheus + Grafana + relevant exporters. Implemented with:
  - node_exporter for OS metrics (CPU, RAM, disk, network)
  - pgexporter for PostgreSQL database metrics
  - pg-backup service with cron scheduling for implementing backup requirement (NF3)
  - FastAPI backend instrumentation via `prometheus-fastapi-instrumentator` to expose a `/metrics` endpoint
  - Containerized deployment of the entire monitoring stack via Docker Compose
- **Consequences:**
  - **(+)** Provides powerful, standard monitoring and alerting capabilities. Aligns with cloud-native practices. Relatively low overhead for basic metrics.
  - **(-)** Adds components to manage/configure during deployment. Requires application instrumentation (`/metrics` endpoint). Grafana dashboard creation requires effort.
- **Significance:** Establishes the standard approach for observability, crucial for meeting NF9, supporting Administrator tasks (FR6), and verifying NF7.

#### 2.2.4 ADR-004: API Authentication Mechanism

- **Status:** Accepted
- **Date:** 2025-04-11
- **Deciders:** BEST SE TEAM EVER
- **Context:** The API Backend needs to securely authenticate requests from the TUI client (NF4). User credentials (passwords) require secure storage.

- **Decision Drivers:** Statelessness suitable for API/TUI interaction, standard practice (JWT), wide library support (`python-jose`), decoupling auth state from server, need for strong password security.
- **Options Considered:**
  - **JWT (JSON Web Tokens):** TUI logs in (`/auth/token`), backend validates credentials (`user_repo`, `security.verify_password`), issues short-lived access token. TUI sends token in `Authorization: Bearer` header. Backend verifies token signature (`security.decode_access_token`). Chosen.
  - **Session Cookies:** Traditional web approach. Backend manages session state. Less ideal for non-browser clients like a TUI. More stateful.
  - **API Keys:** Simpler for machine-to-machine, but less flexible for user sessions, revocation harder.
- **Outcome:** Chose JWT for API authentication between TUI and Backend. Passwords stored in PostgreSQL (`User.hashed_password`) using bcrypt via `passlib` (`security.get_password_hash`). Anilist OAuth tokens stored directly in `AnilistToken.access_token` (could be improved with encryption/vault in future).
- **Consequences:**
  - **(+)** Stateless API authentication simplifies backend scaling. Standard and well-supported mechanism.
  - **(-)** Requires secure handling/storage of JWTs on the client (TUI's `api_client.py` stores it in memory). Token expiration requires handling (currently short-lived, implicitly requires re-login). Direct storage of Anilist token is a minor security risk.
- **Significance:** Defines the core security mechanism for user interaction (NF4, NF5). Impacts TUI client (`api_client.py`), API backend (`security.py`, `deps.py`, `endpoints/auth.py`), and database schema (`User` model).

# 3 Architectural Tactics and Patterns

## 3.1 Architectural Tactics

We employ the following architectural tactics to achieve specific quality attributes, primarily non-functional requirements:

1. **Tactic: Concurrency (Performance)**
   - **Explanation:** Utilize asynchronous programming (`async`/`await`) extensively in the FastAPI backend (`main.py`, `endpoints/`, `services/`) and for I/O operations (database via `asyncpg`/SQLAlchemy, external API calls via `httpx` in `anilist_service.py`, `nyaa_service.py`). The synchronous qBittorrent library is called using `run_in_executor` within `torrent_orchestration_service.py` to avoid blocking the main async event loop.
   - **Addresses:** NF1 (Watchlist Search Latency), NF2 (Concurrency). Prevents blocking during network or disk I/O, allowing the server to handle many simultaneous requests efficiently.
2. **Tactic: Authenticate Actors & Authorize Actors (Security)**
   - **Explanation:**
     - **Authentication:** User login via `/auth/token` validates credentials against bcrypt-hashed passwords stored in the DB (`user_repo.py`, `security.py`). Successful authentication yields a JWT. Subsequent requests require a valid JWT (`oauth2_scheme` dependency). Anilist access uses a user-provided token stored per-user (`token_repo.py`).
     - **Authorization:** API endpoints are protected using FastAPI dependencies (`deps.py`). `get_current_user` ensures a valid token maps to an active user. `verify_torrent_ownership` checks if the requesting user is linked to the target torrent hash in `UserTorrentLink` before allowing actions like pause/resume/delete. `CurrentAdminUser` restricts specific endpoints (like `/torrents/all`) to admin users.
   - **Addresses:** NF4 (Security - Authentication), NF5 (Security - Authorization). Ensures only legitimate, authorized users perform actions on their own resources.
3. **Tactic: Health Monitoring & Ping/Echo (Reliability/Availability)**
   - **Explanation:** The API Backend is instrumented using `prometheus-fastapi-instrumentator` (`main.py`) exposing detailed metrics (request counts, latency histograms, etc.) at `/api/metrics` for Prometheus scraping (Health Monitoring). A simple `/api/health` endpoint (`main.py`) provides a basic liveness check (Ping/Echo). This allows the external monitoring system (Prometheus/Grafana) to track application health.
   - **Addresses:** NF9 (Monitoring Availability & Freshness), supports C7, C10. Facilitates operational visibility and early detection of issues.
4. **Tactic: Resource Management (Performance / Cost)**
   - **Explanation:** Primarily achieved via ADR-001 (On-Demand RSS Search), which avoids constant background polling/processing, minimizing idle CPU, memory, and network usage. Technology choices (Python/FastAPI) are generally resource-efficient. The system

relies on external monitoring (Prometheus/Grafana via FR6/NF9) to track actual usage against OCI free tier limits (NF7).
- **Addresses:** NF7 (Resource Constraints), indirectly supports NF1/NF2 by ensuring resources are available for active requests rather than background tasks.

5. **Tactic: Maintain Internal Model & Abstract Data Access (Maintainability)**
   - **Explanation:**
     - **Maintain Internal Model:** Pydantic models (`schemas/`) define the API contract and internal data structures, separate from the SQLAlchemy database models (`db/models.py`). Service layer logic operates on these Pydantic or domain models.
     - **Abstract Data Access:** The Repository pattern (`db/repository/`) provides a clear abstraction layer over database operations. Services (`services/`) interact with repositories (e.g., `UserRepository`, `TorrentRepository`) instead of directly executing SQL or using SQLAlchemy specifics, decoupling business logic from data persistence details.
   - **Addresses:** NF8 (Maintainability), supports C13, C14, C15. Improves code organization, testability, and makes future modifications (e.g., changing DB query implementation) easier.

## 3.2 Implementation Patterns

We utilize the following design patterns within the architecture:

1. **Pattern: Repository Pattern**
   - **Role:** Decouples the service layer (`services/`) and API endpoint handlers (`api/endpoints/`) from the data persistence logic (PostgreSQL access via SQLAlchemy).
   - **Explanation:** Interfaces for data operations are conceptually defined (e.g., get user, save token, link torrent). Concrete implementations (`UserRepository`, `AnilistTokenRepository`, `TorrentRepository` in `db/repository/`) encapsulate the SQLAlchemy async session usage and query logic. Services request repository instances via Dependency Injection.
   - **Benefit:** Centralizes data access logic, improves testability (repositories can be mocked), enhances maintainability (NF8) by isolating DB interaction code.

2. **Pattern: Model-View-Controller (MVC) / Model-View-Presenter (MVP) Variant**
   - **Role:** Structures the user-facing components (TUI) and the API request handling layer (FastAPI).
   - **Explanation:**
     - **TUI (Textual - closer to MVP/MVVM):**
       - **View:** Textual widgets (`card.py`, `main.py` widgets like `Button`, `Static`, `Markdown`) displaying data.
       - **Presenter/ViewModel:** TUI application logic (`main.py: Switcher`, `card.py: Card, Progress`) handles user input events (`on_button_pressed`, `on_click`), calls the API client (`api_client.py`), manages TUI state (e.g., `Progress.state`), and updates the View widgets (`.label`, `.disabled`, CSS classes).

    - **Model:** Data structures received from the API (`api_client.py` returns dicts, parsed into UI elements).
  - **API Backend (FastAPI - closer to MVC):**
    - **Controller:** FastAPI path operation functions (`api/endpoints/`) receive HTTP requests, use Pydantic models (`schemas/`) for validation/serialization, call appropriate services (`services/`), and return responses.
    - **Model:** Pydantic models (`schemas/`) for API contract, SQLAlchemy models (`db/models.py`) for DB structure, domain logic potentially within services. Repositories manage persistence.
    - **View:** JSON responses generated by FastAPI based on return values/models.
- **Benefit:** Enforces separation of concerns, leading to more organized, testable, and maintainable code (NF8).

3. **Pattern: Dependency Injection**
   - **Role:** Manages dependencies between components within the FastAPI backend, particularly for accessing database sessions and repositories.
   - **Explanation:** FastAPI's `Depends()` mechanism is used extensively (`api/deps.py`). Functions like `get_db`, `get_user_repository`, `get_token_repository`, `get_torrent_repository` provide instances of `AsyncSession` or repository classes. Endpoint functions declare these dependencies in their signatures (e.g., `user_repo: UserRepoDep`), and FastAPI automatically provides the required instances per request.
   - **Benefit:** Decouples components, enhances testability (easy to inject mock dependencies during tests), improves code clarity and maintainability (NF8).

4. **Pattern: API Gateway (Conceptual)**
   - **Role:** The API Backend (`backend/app/main.py` and its included routers) serves as the single, unified interface for the TUI client (`frontend/`).
   - **Explanation:** The TUI makes all its calls to the FastAPI backend. The backend then orchestrates the necessary interactions with the database (PostgreSQL), the torrent client (qBittorrent), and external services (Anilist, Nyaa.si). This shields the TUI from the complexity and specific locations/protocols of the downstream services.
   - **Benefit:** Simplifies client development (`api_client.py`), centralizes cross-cutting concerns like authentication (JWT check in `deps.py`) and request logging/monitoring (`prometheus-fastapi-instrumentator`), and provides a stable interface even if internal components change.

# 4 Prototype Implementation and Analysis

## 4.1 Prototype Development

The current state of the UUUTorrent project represents a working prototype demonstrating the core functionality and architectural design based on a **Client-Server architecture with a direct-call, Modular Monolithic backend**. It integrates the major subsystems and realizes key user workflows essential to the system's operation using synchronous-within-request orchestration for core tasks like downloads.

### 4.1.1 Implemented Core Functionality & Workflows:

The prototype successfully implements the following, showcasing the practical application of the proposed solution:

1. **User Authentication:** Users can sign up and log in (`/auth/signup`, `/auth/token`). The TUI uses JWTs for API calls (`api_client.py`). (Addresses FR2, NF4).
2. **Anilist Integration:** Users can link their Anilist account (`/auth/anilist/link`), view their 'CURRENT' watchlist (`/watchlist/`, `frontend/main.py:Cards`), and update episode progress (`/watchlist/progress`, `frontend/card.py:Progress`). (Addresses FR4, partial FR2, NF4).
3. **Direct-Call Watchlist Download:** This core workflow follows a direct request-response pattern:
   - User selects an unwatched episode in the TUI (`frontend/card.py`).
   - The TUI calls the backend endpoint (`/watchlist/download`).
   - The endpoint handler **synchronously orchestrates** the entire process within the request cycle:
     - Calls `anilist_service.get_media_details`.
     - Calls `torrent_orchestration_service.download_watchlist_episode`, which in turn:
       - Calls `nyaa_service.search`.
       - Selects the best torrent (`_select_best_torrent`).
       - Calls `qbittorrent_service.add_torrent_source` (using `run_in_executor` to handle the synchronous qBit library call without blocking the main async event loop).
       - Calls `torrent_repo.link_torrent` to update the database.
   - Once all steps are complete (or an error occurs), the endpoint returns the final result (success message with hash, or error) to the TUI. (Addresses FR1, FR3, FR4).
4. **Torrent Status Monitoring:** The TUI fetches and displays torrent status (`/torrents/`, `frontend/card.py`) as before. (Addresses FR5, NF6).
5. **Torrent Lifecycle Management:** Pausing, resuming, and deleting user-owned torrents via the TUI (`api_client.py` calling `/torrents/{hash}/...` endpoints) are functional. (Addresses FR5).
6. **API Backend Implementation:** A robust FastAPI backend provides the RESTful API, implements business logic in services (`services/`), interacts with the database via repositories (`db/repository/`), and handles external API communication asynchronously

where possible (Anilist, Nyaa via `httpx`), using `run_in_executor` for the synchronous qBittorrent library. (Addresses NF8, supports all FRs).
7. **TUI Implementation:** A functional Textual TUI (`frontend/`) provides the user interface for the core workflows mentioned above. (Addresses FR7, NF6).
8. **Basic Monitoring Hooks:** The API backend exposes `/metrics` and `/health` endpoints for integration with a monitoring system. (Addresses NF9).

#### 4.1.2 Architectural Showcase:

This prototype demonstrates key architectural decisions of the **implemented** direct-call approach:

- **Client-Server Architecture:** Clear separation between the Textual TUI (client) and the FastAPI backend (server).
- **Asynchronous Backend (with Synchronous Orchestration):** Use of `async/await` in FastAPI for handling requests and external IO (Anilist, Nyaa), but the core download workflow orchestration happens sequentially within the request-response cycle.
- **Layered Backend:** Separation into API endpoints, services, repositories, and database models.
- **Direct Service Interaction:** Services directly call other services or repositories needed to fulfill a request (e.g., endpoint calls orchestration service, which calls Nyaa service, qBit service, repo).
- **External Service Integration:** Interaction with Anilist (GraphQL), Nyaa.si (RSS), and qBittorrent (Web API) via dedicated service modules.
- **Database Interaction:** Use of PostgreSQL with SQLAlchemy and the Repository pattern.
- **Authentication & Authorization:** Implementation of JWT-based authentication and resource ownership checks.
- **Technology Stack:** Practical application of Python, FastAPI, Textual, PostgreSQL, Pydantic, SQLAlchemy, httpx, etc. (ADR-002).

The prototype effectively validates the feasibility of the core concepts using a direct-call monolithic backend structure.

#### 4.1.3 Implementation of NF3 (Reliability - Backup & Recovery)

To satisfy the NF3 requirement for automated daily database backups with an RPO of 24 hours and RTO of 1 hour, we've implemented:

1. **pg-backup Container**: A dedicated Docker container that:
   - Runs a cron job scheduled for daily execution
   - Uses `pg_dump` to create a complete backup of the PostgreSQL database
   - Stores backups in a persistent Docker volume (`pg_backups`)
   - Maintains backup history with timestamps
   - Configured with proper database credentials via environment variables
2. **Recovery Process**:
   - In case of database failure, the operations team can:
   - Access the backup files in the `pg_backups` volume

- Use `pg_restore` to recreate the database from the most recent backup
- The containerized approach ensures backups are isolated from application issues

This implementation ensures that the system can recover from database failures within the specified RTO of 1 hour, with a maximum data loss of 24 hours (RPO).

## 4.2 Architecture Analysis

We analyze the implemented **direct-call, modular monolithic backend** architecture against key non-functional requirements and compare it to an alternative.

### 4.2.1 Comparison Architecture: Event-Driven

For comparison, we consider an **Event-Driven Architecture (implemented within the Monolith)** as described previously. In this alternative:

- The `/watchlist/download` endpoint publishes a `DownloadEpisodeRequested` event to an internal queue and immediately returns `202 Accepted`.
- Background worker tasks asynchronously consume events to perform searching (`handle_download_request` worker) and torrent addition/DB linking (`handle_torrent_found` worker).
- This contrasts with the implemented approach where all these steps happen sequentially **before** the API endpoint returns a response.

### 4.2.2 Analysis of Implemented (Direct-Call) Architecture

#### 4.2.2.1 NF1: Performance (Watchlist Search Latency)

- **Quantification Method:** Measure the total time taken from the TUI sending the `POST /watchlist/download` request to receiving the **final** response (success or error). This can be timed client-side (`api_client.py`) or server-side (logging start/end times within the endpoint handler, using a request ID). Further server-side logging can break down the time spent in each major step (Anilist call, Nyaa search, qBit add call, DB link).
- **Analysis:**
  - The **total request-response time** directly reflects the sum of all operations performed sequentially within the request handler: API call overhead + Anilist fetch + Nyaa search + parsing + selection + qBit add (via executor) + DB link.
  - The user **directly perceives** this entire duration as the waiting time. While `asyncio` prevents the server from being fully blocked by external I/O (Anilist, Nyaa) and the executor handles the qBit sync library, the **request itself** is blocked until all steps complete.
  - Expected latency will be similar to the E2E time calculated for the event-driven version (e.g., **1.2s to 8.2s** typical estimate), but this entire duration is experienced by the user as wait time before getting confirmation/error.
- **Trade-offs:** Simpler transactional logic – if any step fails, the endpoint can immediately return an error, and the system state is generally consistent (no torrent added if search failed, etc.). However, this comes at the cost of poorer **perceived** user responsiveness for long-running operations. The endpoint holds resources for the entire duration. Susceptible to timeouts if external services are very slow.

- **Comparison of different architectures:**

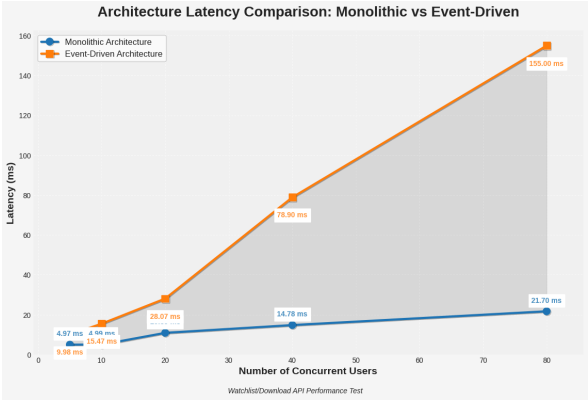


Figure 15: Total API Request Times for `/watchlist/download`.

#### 4.2.2.2 NF7: Resource Constraints (OCI Always Free Tier)

- **Quantification Method:** Deploy the application on an OCI Always Free instance. Configure Prometheus/Grafana to monitor OS, DB, and API metrics. Run simulated user load, focusing on triggering the `/watchlist/download` workflow concurrently. Monitor key metrics (CPU, RAM, Disk I/O, Network I/O) under different load levels.
- **Analysis:**
  - **Idle Load:** Expected to be very low, consisting mainly of the FastAPI process, DB, and qBit processes waiting for connections/tasks. This architecture is efficient at idle.
  - **Load Handling:** Resource usage (CPU, potentially RAM for processing data) increases directly in proportion to the number of active requests being processed. If many users trigger downloads simultaneously, multiple instances of the orchestration logic run concurrently (up to worker limits), potentially causing resource spikes (CPU for searching/parsing, qBit interaction; RAM for holding data). Since processing happens within the request cycle, there's no buffering effect like an event queue.
  - **Overall Impact:** Resource usage is tightly coupled to active user requests. The critical factors remain qBittorrent's activity and PostgreSQL load. ADR-001 (On-demand search) is still key to minimizing **idle** load. Adherence to NF7 under load depends on the efficiency of the sequential processing and the number of concurrent requests the OCI instance can handle without exceeding CPU/RAM limits.
- **Trade-offs:** Simpler resource model (usage maps directly to requests). Less resilient to sudden, large bursts of requests compared to an event-driven approach with buffering, as it might lead to resource exhaustion or request timeouts if processing takes too long. No risk of unbounded queue memory growth.
- **Visualization for different architectures:**

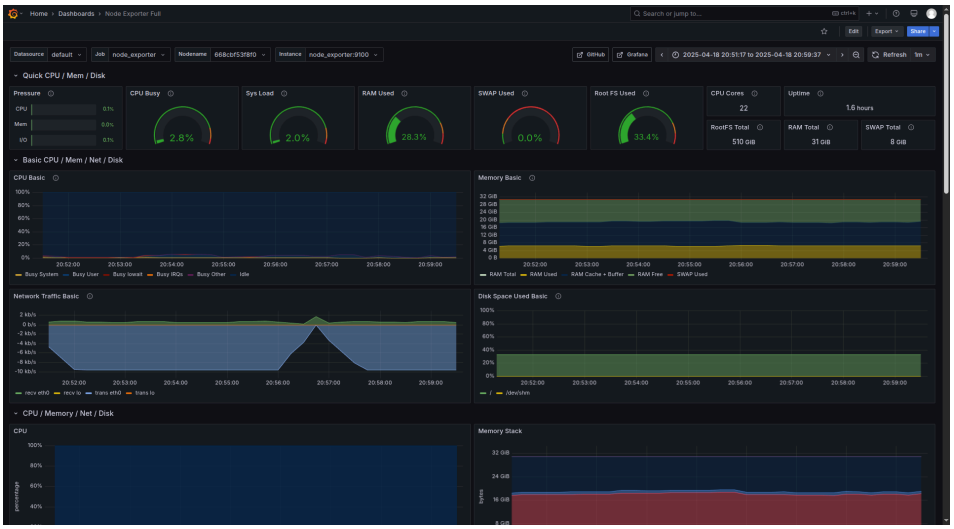


Figure 16: Grafana dashboard showing CPU, RAM, Network I/O during simulated load test (Direct-Call).

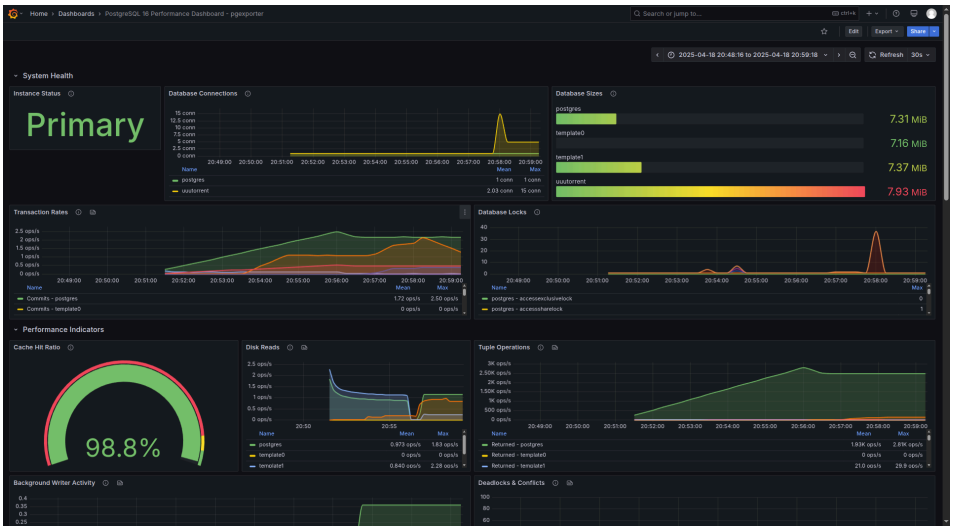


Figure 17: Grafana dashboard for monolithic system database workload

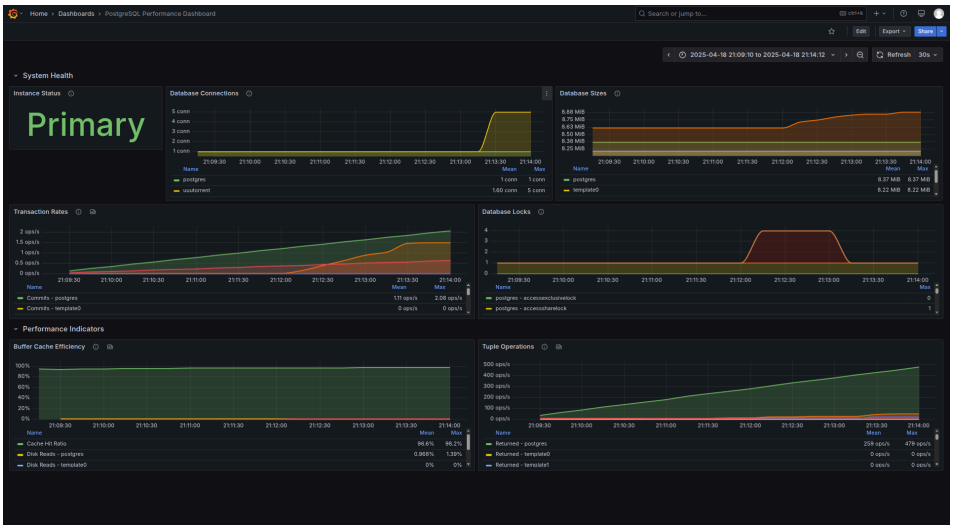


Figure 18: Grafana dashboard for event driven system database workload

#### 4.2.3 Conceptual Comparison: Direct-Call vs. Event-Driven

- **Responsiveness:** Direct-Call has higher perceived latency for initiating downloads but provides immediate success/failure confirmation. Event-Driven offers low perceived latency (fast `202 Accepted`) but delays confirmation and makes tracking harder for the user.
- **Coupling:** Direct-Call has tighter coupling between the endpoint and the orchestration steps. Event-Driven decouples the endpoint from the background processing logic.
- **Complexity:** Direct-Call has simpler control flow and error handling for a single request. Event-Driven introduces complexity with queues, workers, potential message ordering issues, and distributed error handling/compensation.
- **Scalability/Load Handling:** Event-Driven potentially handles load spikes more gracefully due to buffering, while Direct-Call might hit resource limits or timeouts faster under burst load.
- **Consistency:** Direct-Call makes transactional consistency easier to achieve within a single request. Event-Driven requires careful design (e.g., idempotency, retries, monitoring) to handle partial failures across worker stages.

# 5 Reflections on Project Process

This section reflects on the journey of developing the UUUTorrent project, covering successes, challenges, and lessons learned throughout the software engineering lifecycle.

## 5.1 What Went Well

- **Technology Stack Integration:** The chosen stack (Python, FastAPI, Textual, PostgreSQL, Docker) integrated relatively smoothly. FastAPI's async capabilities were beneficial for handling external API calls, and Textual provided a good framework for the TUI. Docker Compose significantly simplified the setup and deployment of the backend services, especially the database and monitoring components.
- **Clear Core Requirements:** The initial project goal – linking Anilist to remote qBittorrent via a TUI – provided a clear focus, helping to scope the Minimum Viable Product (MVP).
- **Modular Backend Design:** Adopting patterns like the Repository pattern and Dependency Injection early on paid off, making the backend code easier to manage, test (conceptually, though formal unit tests might be limited), and reason about as different features were added.

## 5.2 Challenges Faced

- **External API Reliability & Quirkiness:** Integrating with third-party APIs (Anilist, Nyaa.si) presented challenges. Occasional downtime, rate limits, and parsing inconsistencies (especially with RSS feeds like Nyaa.si) required careful error handling and sometimes rework. The synchronous nature of the `python-qbittorrentapi` library required using `run_in_executor`, adding a slight complexity layer to the async backend.
- **TUI State Management:** Managing state effectively within the Textual TUI, especially reflecting asynchronous backend operations (like download progress polling), proved more complex than initially anticipated. Ensuring the UI updated correctly without blocking required careful use of timers and callbacks.
- **OCI Free Tier Limitations:** Working within the strict resource constraints of the OCI Always Free tier influenced design decisions (ADR-001) and required careful monitoring during testing to ensure the application didn't exceed memory or CPU limits under load. Deployment required careful configuration.
- **Scope Creep Management:** Initially, ideas like built-in file transfer or more advanced user management were considered but had to be deferred to keep the project achievable within the timeframe and focused on the core requirements.
- **Testing Asynchronous Code:** While the design aimed for testability, thoroughly testing asynchronous interactions, especially those involving external services and timing (like the event-driven alternative), can be challenging.

## 5.3 Design Decision Process

Design decisions, documented in the ADRs, were generally made through team discussion. We typically started with the requirements (functional and non-functional, especially performance and resource constraints) and brainstormed potential solutions. Options were weighed based on:

- Alignment with requirements (especially NFRs like NF7).
- Technical feasibility and complexity.
- Developer familiarity and available libraries/tools.
- Maintainability and potential for future extension.

ADRs were drafted to capture the outcome and rationale, ensuring alignment within the team. For instance, the decision for on-demand search (ADR-001) was heavily driven by the need to conserve resources (NF7) despite the potential latency impact (NF1 trade-off).

## 5.4 Changes from Initial Plan

- **Initial Idea vs. Final Implementation:** The initial concept involved a more sophisticated background RSS processing engine, which was simplified to the on-demand search (ADR-001) due to complexity and resource concerns.
- **Feature Prioritization:** Some nice-to-have features, like direct .torrent file uploads, advanced search filtering in TUI, were de-prioritized during development to ensure the core watchlist-to-download workflow was robustly implemented.

## 5.5 Lessons Learned

- **Importance of NFRs:** Non-functional requirements, particularly resource constraints (NF7) and external dependencies (NF10), significantly shaped the architecture (driving ADR-001). Ignoring NFRs early can lead to major rework later.
- **Value of Modularity:** Even within a monolith, structuring the code logically (services, repositories) makes development and debugging much more manageable (NF8).
- **Asynchronous Programming Complexity:** While powerful for I/O-bound tasks, `asyncio` requires careful handling of tasks, error propagation, and potential race conditions (as seen when initially debugging the event-driven alternative). Bridging sync/async code (like with qBittorrent) needs specific patterns (`run_in_executor`).
- **External API Integration Risks:** Relying on external services introduces dependencies that are outside the team's control. Robust error handling, timeouts, and potentially fallback mechanisms are crucial.
- **Iterative Development:** Starting with a core workflow and iteratively adding features or refining components proved effective. Trying to build everything perfectly at once can be overwhelming.

## 5.6 What We Would Do Differently

- **More Robust Error Handling:** Implement more specific error handling and user feedback mechanisms, especially for failures during the multi-step download process (e.g., informing the TUI if Nyaa search fails vs. qBit add fails). Consider adding failure events in the event-driven alternative.
- **Formalized Testing:** Introduce more automated testing earlier in the process, particularly integration tests for API endpoints and potentially unit tests for critical service logic, to catch regressions.
- **Configuration Management:** Improve configuration handling, perhaps using a more structured approach than just `.env` files, especially for deployment-specific settings.

- **Security Enhancements:** Consider encrypting the stored Anilist token in the database rather than storing it plain-text. Implement refresh token logic for JWTs to avoid frequent re-logins.