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) and securely transfer completed files (e.g., via user-initiated SFTP/SCP) to local devices.
- **FR2: User and Access Management:** Implement user authentication (login/password) and authorization. Define distinct roles: 'User' (manages personal downloads, interacts with watchlist) and 'Admin' (manages users, system monitoring access).
- **FR3: Watchlist-Driven Torrent Search:** The backend must integrate with external services (e.g., Anilist API) to fetch a user's watchlist. Based on user selection of an unwatched item in the TUI, the backend performs a targeted search against a pre-configured external RSS source (e.g., Nyaa.si search) using item details (e.g., title, episode number) to find relevant torrents and initiate downloads via qBittorrent.
- **FR4: Watchlist Integration & Display:** Allow users (via TUI) to view their media watchlist fetched from an external service (e.g., Anilist). The TUI should clearly indicate unwatched items, enabling users to trigger the download search process (FR3). Basic management (e.g., initiating Anilist authentication/linking) is required.
- **FR5: Torrent Lifecycle Management:** Enable users (via TUI) to view download status (downloading, seeding, completed, error), pause, resume, and delete their torrents managed by the system.
- **FR6: Admin Monitoring View:** Provide administrators access to a web-based dashboard (Grafana) displaying real-time server resource usage (CPU, RAM, disk, network), qBittorrent statistics, database health, and potentially aggregated user activity/stats.
- **FR7: User Interface (TUI):** Provide a cross-platform Terminal User Interface (Textual) for user interaction: displaying torrent status, viewing the watchlist, triggering downloads for watchlist items, adding torrents directly (magnet links), and getting information for file transfers. 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 without significant API/TUI responsiveness degradation. (Network throughput depends on OCI/peers).

- **NF3: Reliability (Backup & Recovery):** Perform automated daily database backups (PostgreSQL). RTO: 1 hour. RPO: 24 hours.
- **NF4: Security (Authentication):** Secure user authentication (hashed+salted passwords). Protect API endpoints with token-based authentication (e.g., JWT). Handle external service (Anilist) authentication securely (e.g., OAuth2 flow, secure token storage).
- **NF5: Security (Authorization):** Enforce authorization rules: users manage only their own resources (torrents, watchlist interaction); admins have specific elevated privileges (user management, monitoring access).
- **NF6: Usability (TUI):** The TUI should be intuitive for terminal users, providing clear feedback, navigation, and status updates, especially for watchlist interaction and download initiation.
- **NF7: Resource Constraints (OCI):** Operate within OCI Always Free tier limits (Ampere A1 CPU/RAM, storage, network egress). Monitor usage. The on-demand search approach helps mitigate constant background load.
- **NF8: Maintainability:** Ensure modular, documented backend code with tests for easier updates.
- **NF9: Monitoring Availability & Freshness:** The Admin monitoring view (Grafana) should be highly available (best effort on free tier) and display metrics with a latency of <= 1 minute from collection time.
- **NF10: External Service Dependency:** The system's watchlist functionality relies on the availability and performance of the external Watchlist API (e.g., Anilist) and the external RSS search source (e.g., Nyaa.si). System should handle potential unavailability gracefully (e.g., error messages in 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, database schema (for user mapping/tokens), and TUI design. 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), efficient external API interaction, optimized search/parsing logic, and careful consideration of qBittorrent API usage 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.
- **NF4 & NF5 (Security):** Fundamental requirements demanding careful design of authentication (password hashing, JWT, OAuth handling) and authorization (role checks, resource ownership checks) mechanisms throughout the API backend. Impacts data storage and API endpoint design.

- **NF7 (Resource Constraints):** The OCI Always Free tier limitation is a primary driver for technology choices (favoring lightweight options like Python/FastAPI/Textual), architectural patterns (on-demand processing over background polling), and necessitates resource monitoring (NF9).
- **NF8 (Maintainability):** Influences the adoption of modular design, separation of concerns (Subsystem Overview), clear API contracts, and implementation patterns (Repository, DI) to facilitate future development and debugging.
- **NF9 (Monitoring):** Requires incorporating a dedicated monitoring stack (Prometheus/Grafana/Exporters) and instrumenting the API backend and potentially other components to expose relevant metrics.
- **NF10 (External Dependency):** Necessitates designing the API backend and TUI to be resilient to failures or delays in external services (Anilist, Nyaa), requiring error handling, timeouts, and potentially caching strategies.

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 running on the user's machine. Provides views for torrent status, Anilist watchlist display (highlighting unwatched), and mechanisms to trigger watchlist downloads, add magnet links, manage torrent lifecycle (pause/resume/delete), authenticate, and 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.

1.2.2 API Backend (FastAPI):

1.2.2.1 Role & Functionality

The central coordinating service on the OCI instance. Handles API requests from the TUI, manages user authentication/authorization (including Anilist OAuth tokens), interacts with Anilist API to fetch watchlists, performs on-demand searches against Nyaa.si RSS, orchestrates torrent management via qBittorrent's Web API, interacts with the database, and exposes metrics for monitoring.

1.2.2.2 Interaction

- Listens to TUI requests
- Calls qBittorrent Web API
- Queries/updates PostgreSQL DB
- Calls external Anilist API
- Calls external Nyaa.si RSS Search
- Scraped by Prometheus.

1.2.3 Torrent Client Service (qBittorrent):

1.2.3.1 Role & Functionality

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

1.2.3.2 Interaction

- Exposes a Web API for control by the API Backend
- 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 on the OCI instance. Stores user credentials (hashed passwords), roles, mappings between users and torrents (hashes), and securely stored user-specific Anilist API tokens/identifiers.

1.2.4.2 Interaction

- Responds to SQL queries from the API Backend
- Scraped for metrics by pgexporter.

1.2.5 Monitoring Service (Prometheus + Grafana + Exporters):

1.2.5.1 Role & Functionality

The operational monitoring stack on the OCI instance. Prometheus scrapes metrics (OS, DB, API Backend, qBittorrent) from exporters; Grafana provides web-based dashboards for Admins to visualize system health and performance. Node_exporter and pgexporter provide OS and DB metrics respectively.

1.2.5.2 Interaction

- Prometheus scrapes targets (API Backend /metrics, node_exporter, pgexporter)
- Grafana queries Prometheus and serves web UI.

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

1.2.6.1 Role & Functionality

A third-party service providing media information and user watchlist management. Acts as the authoritative source for the user's watchlist content.

1.2.6.2 Interaction

Receives authorized API requests (GraphQL/HTTPS) from the API Backend.

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

1.2.7.1 Role & Functionality

A third-party torrent indexer providing search results via RSS feeds. Used by the API Backend on-demand 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.

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 media watchlist.
- **Administrator:** Personnel responsible for maintaining the OCI instance, monitoring system health, managing users, and ensuring service availability.
- **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).

2.1.2 Concerns

- **End User Concerns:**
  - C1: Ease of finding and downloading desired unwatched media from watchlist.
  - C2: Reliability and speed of downloads.
  - C3: Security of user account and Anilist integration.
  - C4: Clarity of torrent status and download progress.
  - C5: Ability to manage torrents (pause, resume, delete).
  - C6: Ease of transferring completed files.
- **Administrator Concerns:**
  - C7: System stability and availability.
  - C8: Resource consumption within OCI free tier limits (NF7).
  - C9: Security of the overall system (preventing unauthorized access, data breaches).
  - C10: Ease of monitoring system health and performance (NF9).
  - C11: Backup and recovery capabilities (NF3).
  - C12: Ease of user management.
- **Developer Concerns:**
  - C13: Maintainability and extensibility of the codebase (NF8).
  - C14: Testability of components.
  - C15: Clear separation of concerns between subsystems.
  - C16: Feasibility of implementation within project constraints.
  - C17: Performance efficiency (NF1, NF2).
- **OCI Concerns:**
  - C18: Adherence to resource usage limits (CPU, RAM, network egress, storage).

2.1.3 Viewpoints and Views

- **Logical Viewpoint:** Describes the functional decomposition of the system, components, and their interactions. Addresses C1, C4, C5, C13, C15, C16.
  - **Views:** Component Diagram, Sequence Diagrams (e.g., Watchlist Download Flow, User Login Flow), Subsystem Description (Task 1).

- **Deployment Viewpoint:** Describes the physical environment, mapping of software components 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, protocols).
- **Security Viewpoint:** Describes how security requirements are met, including authentication, authorization, data protection, and external service integration security. Addresses C3, C9, NF4, NF5.
  - **Views:** Authentication Flow Diagram (JWT, OAuth), Access Control Matrix/Description, Data Security Description (hashing, token storage).
- **Operational Viewpoint:** Describes how the system is operated, monitored, and maintained. Addresses C7, C10, C11, C12, NF3, NF9.
  - **Views:** Monitoring Dashboard Description (Grafana), Backup/Recovery Procedure Description, User Management Procedure Description.
- **User Experience (UX) Viewpoint:** Describes the user's interaction with the system via the TUI. Addresses C1, C4, C6, NF6.
  - **Views:** TUI Mockups/Screenshots, User Interaction Flow Descriptions.
- **Development Viewpoint:** Describes aspects relevant to the development process, including code structure and patterns. Addresses C13, C14, C15, NF8.
  - **Views:** Code Structure Overview, Description of Implementation Patterns (Task 3).

(Mapping Concerns to Viewpoints/Views is implicit in the descriptions above)

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:** Initial idea involved background RSS syncing and filtering. Core need is downloading unwatched watchlist items. Background sync adds complexity, resource usage (NF7), and potential UX friction.
- **Decision Drivers:** Simplify backend, optimize for OCI free tier (NF7), direct UX for watchlist downloads, leverage existing indexer search.
- **Options Considered:**
  1. Background RSS Sync & Filtering
  2. On-Demand Watchlist-Driven RSS Search
  3. Hybrid.
- **Outcome:** Chose Option 2. TUI shows watchlist -> User selects -> Backend searches Nyaa.si RSS -> Backend parses results -> Backend adds best match to qBittorrent. Eliminates user RSS management, background polling, and related DB storage.
- **Consequences:**
  - **(+)** Reduced complexity/resource use, direct user control, faster core feature dev.
  - **(-)** Latency during search (NF1), no proactive downloads, dependency on external search source (NF10), loss of arbitrary RSS monitoring.

- **Significance:** Fundamental shift from proactive background processing to reactive on-demand user-triggered actions. Simplifies backend state, DB schema. Increases dependency on Anilist and Nyaa.si reliability/performance (NF10).

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, maintainability, developer familiarity, and OCI constraints.
- **Decision Drivers:** OCI Free Tier compatibility (NF7), Asynchronous capabilities for performance (NF1, NF2), Python ecosystem familiarity, mature libraries, ease of use for TUI, relational DB for structured data.
- **Options Considered:**
  - **API:** FastAPI (Python/Async), Flask (Python/Sync/Async), Node.js (Express/Koa - JS/Async).
  - **TUI:** Textual (Python), Rich (Python - lower level), ncurses (C - complex), Bubble Tea (Go).
  - **DB:** PostgreSQL (Relational), SQLite (File-based relational - potentially harder scaling/backup), MongoDB (NoSQL - less suitable for relations).
- **Outcome:** Chose FastAPI + Textual + PostgreSQL.
  - **FastAPI:** Excellent performance via async/await, type hinting improves maintainability (NF8), automatic docs, strong community, Python ecosystem. Addresses NF1, NF2.
  - **Textual:** Modern TUI framework in Python, good component model, async support aligns with FastAPI, easier than lower-level libraries. Addresses FR7, NF6.
  - **PostgreSQL:** Mature, reliable RDBMS (NF3), handles relations well (users, torrents, tokens), good performance, standard choice, well-supported on Linux/OCI.
- **Consequences:**
  - **(+)** Good performance potential, consistent language (Python) across TUI/Backend, leverages modern async Python features, strong library support.
  - **(-)** Requires understanding async programming, Textual is relatively new compared to some alternatives.
- **Significance:** Defines the core development environment and runtime characteristics. Impacts performance, maintainability, and required developer skills.

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) within OCI constraints (NF7).
- **Decision Drivers:** Open-source, industry standard, good visualization, ecosystem of exporters, pull-based model suitable for services exposing metrics endpoints, fits within free tier resources.

- **Options Considered:**
  - **Prometheus + Grafana:** Standard combination, wide adoption, many exporters available (node, postgres, custom /metrics).
  - **ELK Stack (Elasticsearch, Logstash, Kibana):** More focused on log aggregation, can be resource-intensive.
  - **Datadog/New Relic:** SaaS solutions, powerful but likely exceed free tier budget/limits.
  - **Netdata:** Real-time, per-second monitoring, potentially more resource-intensive than Prometheus for long-term storage/querying.
- **Outcome:** Chose Prometheus + Grafana + relevant exporters (node_exporter, pgexporter). FastAPI backend will expose a `/metrics` endpoint compatible with Prometheus scraping.
- **Consequences:**
  - **(+)** Provides powerful monitoring and alerting capabilities with standard tools. Fits well with cloud-native practices. Requires configuration of Prometheus targets and Grafana dashboards.
  - **(-)** Adds resource overhead for monitoring components, though generally manageable. Requires instrumenting the API backend.
- **Significance:** Establishes the standard approach for observability, crucial for meeting NF9 and supporting Administrator tasks (FR6). Influences operational procedures.

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 need secure storage.
- **Decision Drivers:** Statelessness suitable for API/TUI interaction, wide library support, standard practice for web APIs, decoupling auth from session management on the server. Secure password storage is paramount.
- **Options Considered:**
  - **JWT (JSON Web Tokens):** TUI logs in with user/pass, backend validates, issues short-lived access token (+ optionally refresh token). TUI sends token in subsequent requests. Backend verifies token signature. Stateless on backend (mostly).
  - **Session Cookies:** Traditional web approach. TUI logs in, backend creates session, returns session ID cookie. TUI sends cookie. Backend manages session state. More stateful, potentially less convenient for non-browser clients.
  - **API Keys:** Simpler, user generates a key, TUI uses key. Less flexible for managing sessions/scopes, revocation can be harder.
- **Outcome:** Chose JWT for API authentication between TUI and Backend. Passwords will be stored in PostgreSQL using a strong hashing algorithm with unique salts (e.g., bcrypt via `passlib`). Secure handling of Anilist OAuth tokens is also required (stored encrypted or using OCI vault if available/feasible).
- **Consequences:**
- **(+)** Stateless API auth, standard and well-supported.

- **(-)** Requires secure handling of JWTs on the client (TUI) and potentially refresh token logic for longer sessions. Need careful implementation of password hashing/salting.
- **Significance:** Defines the core security mechanism for user interaction with the system (NF4, NF5). Impacts TUI client logic and API backend implementation.

3 Architectural Tactics and Patterns

3.1 Architectural Tactics

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

1. **Tactic: Concurrency (Performance)**
   - **Explanation:** Utilize asynchronous processing in the API Backend (via FastAPI's `async/await`) to handle multiple incoming TUI requests, outgoing calls to qBittorrent, database queries, and external API calls (Anilist, Nyaa) concurrently without blocking the main execution thread.
   - **Addresses:** NF1 (Watchlist Search Latency), NF2 (Concurrency). By preventing I/O operations from blocking request handling, the system can remain responsive under load and process multiple operations seemingly in parallel.
2. **Tactic: Authenticate Actors & Authorize Actors (Security)**
   - **Explanation:** Implement robust authentication using username/password (hashed+salted) verified against the database, issuing JWTs for subsequent API calls (Authenticate Actors). On each API request requiring authorization, verify the JWT and check user roles and resource ownership (e.g., user can only manage their own torrents) before proceeding (Authorize Actors). Securely handle Anilist OAuth flow and token storage.
   - **Addresses:** NF4 (Security - Authentication), NF5 (Security - Authorization). Ensures only valid users access the system and they can only perform actions/access data they are permitted to.
3. **Tactic: Health Monitoring & Ping/Echo (Reliability/Availability)**
   - **Explanation:** The API Backend will expose a `/metrics` endpoint scraped by Prometheus (Health Monitoring). Prometheus/Grafana will monitor key indicators (API request rates/errors, DB connections, qBittorrent status via API if possible, resource usage). A simple `/health` endpoint (Ping/Echo) can provide a basic liveness check.
   - **Addresses:** NF9 (Monitoring Availability & Freshness), supports C7, C10. Allows administrators to observe system status, detect failures early, and diagnose problems.
4. **Tactic: Resource Management (Performance / Cost)**
   - **Explanation:** Primarily achieved via ADR-001 (On-Demand RSS Search). Avoids continuous background processing for RSS feeds, significantly reducing CPU, memory, and network usage compared to the alternative. Monitor resource usage via Prometheus/Grafana to stay within OCI free tier limits. Potentially implement rate limiting on external API calls if needed.
   - **Addresses:** NF7 (Resource Constraints), indirectly NF1/NF2 by preserving resources for core tasks.
5. **Tactic: Maintain Internal Model & Abstract Data Access (Maintainability)**
   - **Explanation:** Use distinct data models within the API backend service layer, separate from the database schema (Maintain Internal Model). Employ the Repository pattern (Abstract

Data Access) to decouple service logic from specific PostgreSQL query details. Define clear interfaces between subsystems (TUI-API, API-qBit).
- **Addresses:** NF8 (Maintainability), supports C13, C14, C15. Makes the system easier to understand, modify, and test by isolating changes and reducing coupling.

3.2 Implementation Patterns

We plan to utilize the following design patterns within the architecture:

1. **Pattern: Repository Pattern**
   - **Role:** Decouples the business logic/service layer of the API Backend from the data access logic (PostgreSQL). A repository interface defines data operations (e.g., `getUser`, `saveTorrentMapping`, `getAnilistToken`), and a concrete implementation handles the specific SQL queries.
   - **Explanation:** Services interact with the repository interface, making them independent of the database specifics. This improves testability (can mock repositories) and maintainability (database interactions are centralized).
2. **Pattern: Model-View-Controller (MVC) / Model-View-Presenter (MVP) Variant**
   - **Role:** Structures the user-facing components (TUI and API Backend endpoints) to separate concerns.
   - **Explanation:**
     - **TUI (Textual - closer to MVP/MVVM):**
       - **View:** Textual widgets displaying data (torrent list, watchlist).
       - **Presenter/ViewModel:** TUI application logic handling user input events, calling the API client, managing TUI state, and updating the View.
       - **Model:** Data structures representing torrents, watchlist items (often simple dicts/ Pydantic models received from the API).
     - **API Backend (FastAPI):**
       - **Controller:** FastAPI path operation functions handling HTTP requests, validating input, calling services.
       - **Model:** Pydantic models for request/response validation; Domain models used internally; Repository interactions manage data persistence.
       - **View:** JSON responses returned to the TUI.
   - **Benefit:** Improves organization, testability, and maintainability (NF8) by separating UI/API handling from business logic and data.
3. **Pattern: Dependency Injection**
   - **Role:** Manages dependencies between components, particularly within the API Backend.
   - **Explanation:** Instead of components creating their own dependencies (like database connections or repository instances), these dependencies are “injected” from an external source (e.g., FastAPI’s dependency injection system). For instance, a service class constructor or function signature will declare its need for a `UserRepository`, and the framework provides the instance.
   - **Benefit:** Increases modularity, simplifies testing (can inject mock dependencies), and improves flexibility (NF8). FastAPI has built-in support for this.

4. **Pattern: API Gateway (Conceptual)**
   - **Role:** The API Backend acts as a single entry point (gateway) for the TUI client.
   - **Explanation:** The TUI interacts only with the API Backend, which then orchestrates calls to other internal services (qBittorrent, Database) and external services (Anilist, Nyaa). This simplifies the TUI client, centralizes authentication/authorization, and hides internal system complexity. While not a separate microservice here, it fulfills the pattern's role.
   - **Benefit:** Simplifies client interaction, centralizes cross-cutting concerns like auth and logging, decouples client from internal service details.