Project 3

Team 5

Shaunak Biswas (2022111024)

Kritin Maddireddy (2022101071)

Samyak Mishra (2022101121)

Ashutosh Rudrabhatla (2022111036)

Varun Edachali (2022101029)

Introduction

This document introduces the design and scope of a **Club Recruitment Management System**, a unified platform to streamline and enhance how student organizations attract, evaluate, and onboard new members. By subscription-based updates, application tracking, and interview scheduling, the system will simplify administrative workflows for club administrators while providing candidates with a clear, engaging experience.

1 Requirements and Subsystems

1.1 Functional Requirements

Each requirement is labeled with an identifier FR-x.y for traceability, where x is the major feature category and y is the specific functionality.

FR-1: Centralized Recruitment Portal

FR-1.1	The system shall provide a unified web portal listing active recruitment opportunities across all clubs.
FR-1.2	The system shall support secure login and registration for both student and club representative accounts.
FR-1.3	The system shall support role-based access to restrict or permit functionality as appropriate.

FR-2: Application Management

FR-2.1	The system shall allow students to apply to open positions via application forms.
FR-2.2	The system shall provide a dashboard where students can view their submitted applications and statuses.
FR-2.3	The system shall allow club admins to view and manage received applications.

FR-3: Interview Scheduling

FR-3.1	The system shall allow clubs to create and publish interview slots for specific roles (each role has a different application form).
FR-3.2	The system shall send automated email reminders for upcoming interviews.

FR-4: Email Notification

FR-4.1	The system shall support email-based notifications for key events: new recruitment openings (if that user is subscribed to a club's openings), application updates, and interview schedules.

FR-5: AI-Based Recommendations

FR-5.1	The system shall provide personalized club and role recommendations on the user dashboard.
FR-5.2	The recommendation engine shall consider user profile data and expressed interests, and reported skills.

FR-6: Referral and Endorsement

FR-6.1	The system shall allow authenticated club members to endorse candidates.
FR-6.2	Endorsements shall be visible to club reviewers during shortlisting but hidden from applicants by default.
FR-6.3	The system shall associate endorsements with specific application entries.

1.2 Non-Functional Requirements

These define how the system should behave and place constraints on its functionality. Each is labeled as NFR-x.

NFR-1: Usability

NFR-1.1	The system shall support common user interactions such as canceling and undoing actions wherever relevant.

NFR-2: Performance

NFR-2.1	Interactive operations (e.g., page navigation, form submissions) shall exhibit minimal latency under normal load.

NFR-3: Security

NFR-3.1	All data in transit shall use industry-standard encryption (e.g., TLS); sensitive data at rest shall be encrypted using proven algorithms.
NFR-3.2	Authentication and authorization shall follow robust, configurable policies (e.g., RBAC).
NFR-3.3	All inputs shall be validated and sanitized to prevent CSRF, SQL injection, and similar attacks.
NFR-3.4	Security settings (e.g., session timeout, session encryption key) shall be adjustable via configuration.

NFR-4: Availability

NFR-4.1	The system shall validate readiness and health before becoming available to users.

NFR-5: Maintainability and Extensibility

NFR-5.1	The codebase shall follow modular design principles (e.g., MVC or layered architecture) to isolate changes.
NFR-5.2	APIs shall be documented using standard tools.
NFR-5.3	The architecture shall allow new features or third-party integrations to be added with minimal impact.

1.3 Subsystem Overview

The system is organized into the following main subsystems. Each subsystem encapsulates a coherent set of responsibilities and maps directly to one or more functional requirements (FR-x.y).

User Management Subsystem

Role	Handles everything related to user identities, profiles, and access control.
Responsibilities	Secure registration and authentication of students and club representatives (FR-1.2).
	Role-based access control to gate features for club admins, members and non-members (FR-1.3).
	Storage and maintenance of user profile data and interests (FR-5.2).
	Management of session configuration (NFR-3.2).

Recruitment Portal Subsystem

Role	Provides the primary interface for clubs to publish opportunities and for students to discover them.
Responsibilities	CRUD operations on recruitment posts (FR-1.1).
	Presentation layer components for listing and detailed views of created application forms.

Application Management Subsystem

Role	Orchestrates the end-to-end lifecycle of student applications.
Responsibilities	Submission of applications via dynamic web forms (FR-2.1).
	Applicant dashboard showing the application status and deadline. (FR-2.2)
	Club member/admin interface for reviewing, endorsing, and accepting/rejecting candidates (FR-2.3, FR-6).
	Interface for club members to submit endorsements tied to specific applications (FR-6.1).
	Control of endorsement visibility for reviewers vs. applicants (FR-6.2).

Interview Scheduling Subsystem

Role	Manages interview slot creation, booking, and reminders.
Responsibilities	Creation and publication of interview calendars for each role (FR-3.1).
	Automated email reminders ahead of scheduled interviews (FR-3.3, FR-4.1)

Notification Subsystem

Role	Delivers email notifications in accordance with user preferences.

Responsibilities	Sending of emails for new openings, application updates, and interviews (FR-4.1).

Recommendation Engine Subsystem

Role	Produces personalized club and role suggestions.
Responsibilities	Analysis of profile data, stated skills and stated interests to suggest opportunities (FR-5.1, FR-5.2).
	Exposing recommendation API for integration into the student dashboard.

Infrastructure & Data Persistence Subsystem

Role	Provides shared services such as storage, caching, messaging, and configuration.
Responsibilities	Relational database schemas supporting all domain entities (users, applications, etc.).
	ORM (Object-Relational Mapping) to enable seamless interactions between the relational database and codebase.
	Centralized configuration and secrets management. (NFR-3.4)

2 Architecture Framework

2.1 Stakeholder Analysis

2.1.1 Stakeholders & Concerns

Students (Non-members)

- **Discover & Subscribe:** Easily find and follow clubs of interest
- **Apply & Track:** Complete application forms and monitor their status
- **Notifications:** Receive timely emails to whatever they apply/subscribe.
- **Recommendations:** See relevant club/role suggestions based on profile
- **Privacy:** Control over personal data and visibility.

Club Members

- **Review & Endorse:** Access assigned applications for review, provide endorsements easily.
- **Interview Participation:** View assigned interview slots and applicant details.
- **Partial Visibility:** Access only the portions of the recruitment workflow they're involved in

Club Admins

- **Recruitment Management:** Create/update posts and custom application forms
- **Interview Control:** Schedule, modify, and resolve conflicts for interview slots
- **Applicant Review:** Shortlist, provide feedback, and view endorsements

Developers & Maintainers

- **Modularity & Extensibility:** Build a clean, decoupled codebase with clear API contracts
- **Documentation:** Provide up-to-date docs for all the routes, with detailed examples for each of them.

- **Deployment & Monitoring:** Automate builds, deployments, and health checks with minimal downtime
- **Technical Viability:** Ensure the chosen technologies are appropriate and scalable.

2.1.2 Architectural Viewpoints & Views

2.1.2.1 Interaction Viewpoint

Stakeholders: Students, Club Members

Concerns Addressed: Discover & Subscribe, Apply & Track, Recommendations, Review & Endorse, Interview Participation, Partial Visibility, Privacy (as perceived by user).

Purpose	Focuses on how users interact with the system's features through the user interface. Uses UI/UX conventions, flow descriptions.
Views	*UI Prototypes:* Visual representations of key screens like the Dashboard (showing My Clubs & Recommendations), Form Application Page, My Applications list, Application Detail view (showing status and endorsement count). Addresses: *Discover, Apply, Track, Recommendations, Endorse.*
	User Journey Narratives describing subscription, application, and endorsement workflows. Example: **Student Application Journey** 1) Student logs in. 2) Navigates to 'Clubs' or 'Dashboard'. 3) Finds 'Coding Club Recruitment Form'. 4) Clicks 'Apply'. 5) Fills out questions Q1, Q2, Q3. 6) Clicks 'Submit'. 7) Later, navigates to 'My Applications' to see status 'Ongoing'.

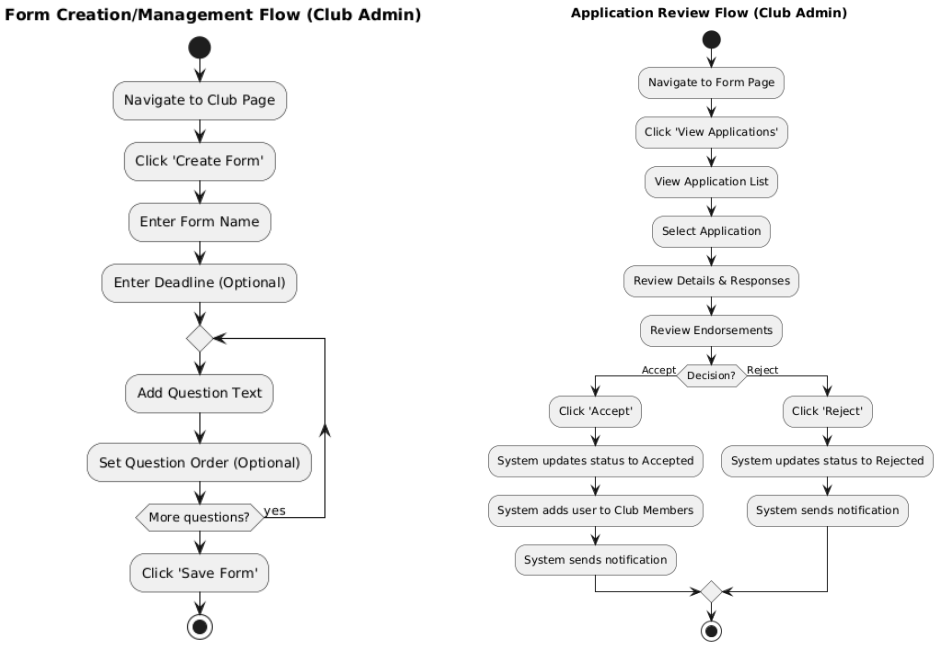
2.1.2.2 Administration Viewpoint

Stakeholders: Club Admins

Concerns Addressed: Recruitment Management, Applicant Review, Interview Control, Club Configuration.

Purpose	Detail the workflows and controls used by club admins.

Views	*Process Narratives & Flowcharts* for recruitment post creation, application review, and interview scheduling.

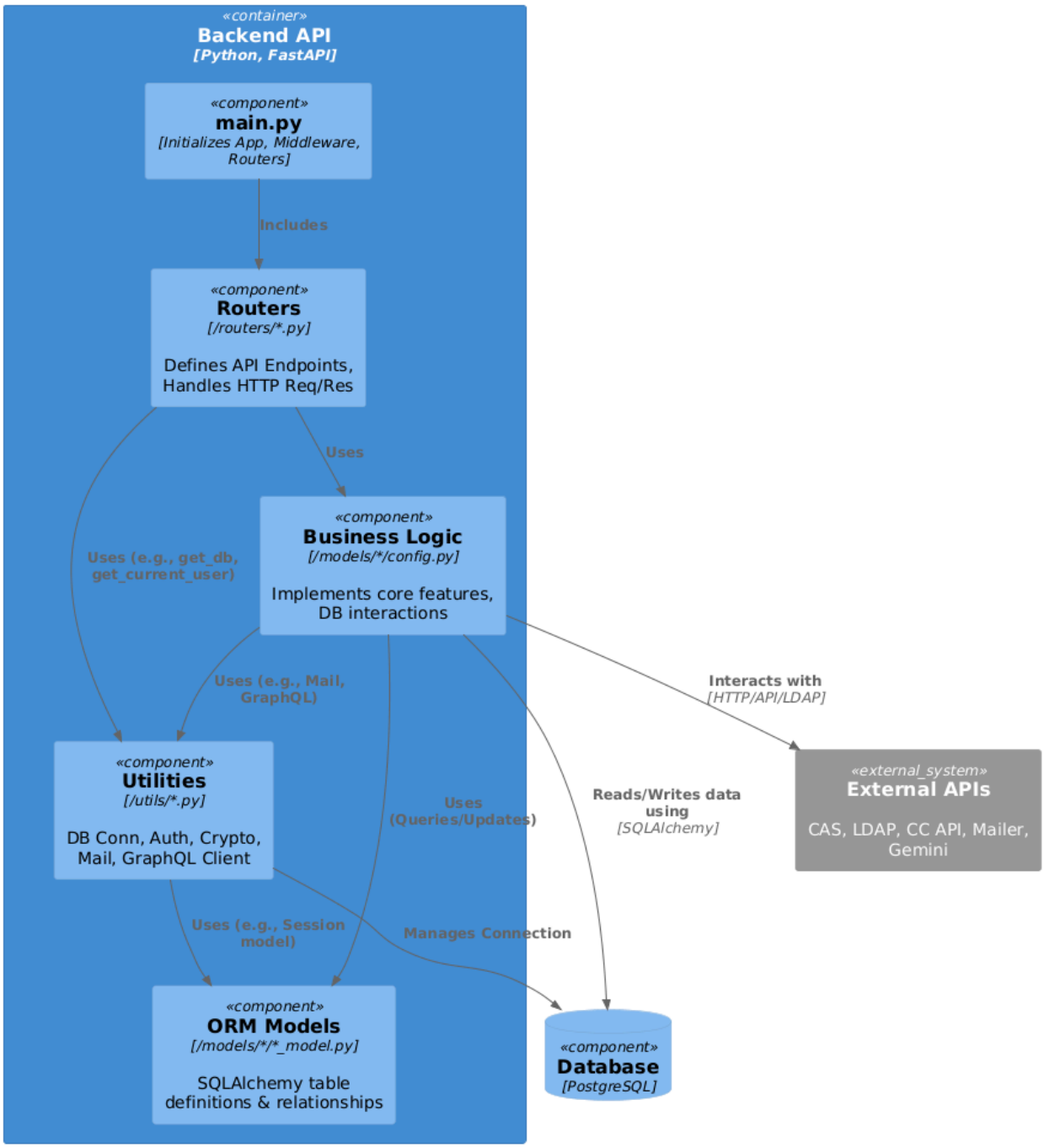


2.1.2.3 Development & Evolution Viewpoint

Stakeholders: Developers, Maintainers

Concerns Addressed: Modularity & Extensibility, Documentation & Testing, Deployment & Monitoring, Technical Viability.

Purpose	Focuses on the internal structure, code organization, APIs, and processes relevant for building and maintaining the system. Uses software design diagrams and technical documentation conventions.
Views	Component Diagrams showing module boundaries and service interactions.
	API Specifications (REST/GraphQL schemas) and data contract definitions.
	CI/CD Pipeline Documentation and code repository structure guidelines.



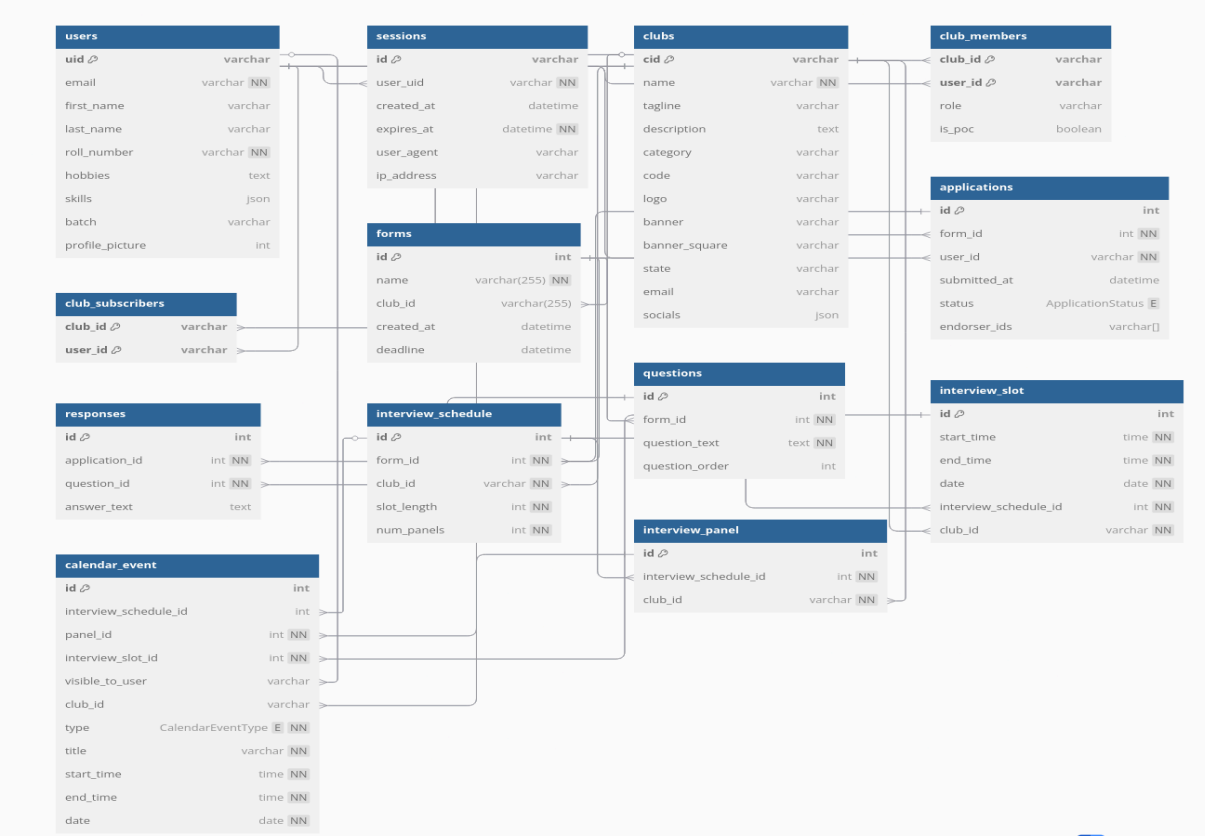
Backend Component Diagram

2.1.2.4 Data & Security Viewpoint

Stakeholders: All (implicitly) and Developers, Maintainers

Concerns Addressed: Privacy, Reliability (data integrity), Compliance, Modularity (data contracts).

Purpose	Focuses on the structure, persistence, integrity, privacy, and security of the system's data. Uses data modeling and security policy conventions.
Views	Data Model Description (entities, relationships, privacy annotations).
	Access Control Matrix mapping roles to resource permissions
	Security Policy Documents covering encryption, authentication/authorization, and input validation.



Database Diagram

Resource/Endpoint	Non-Member (Authenticated)	Club Member	Club Admin	Anonymous
General				
GET /	✓	✓	✓	✓
User Authentication & Profile				
POST /api/user/login	✓	✓	✓	✓
GET /api/user/login	✓	✓	✓	✓
GET /api/user/user_info	✓	✓	✓	✗
GET /api/user/user_role/{club_id}	✓	✓	✓	✗
GET /api/user/user_club_info	✓	✓	✓	✗
GET /api/user/is_authenticated	✓	✓	✓	✓
POST /api/user/logout	✓	✓	✓	✗
PUT /api/user/update_profile	✓	✓	✓	✗
Clubs				
GET /api/club/all_clubs	✓	✓	✓	✓
GET /api/club/{cid}	✓	✓	✓	✓
GET /api/club/is_subscribed/{cid}	✓	✓	✓	✗
POST /api/club/subscribe/{cid}	✓	✓	✓	✗
POST /api/club/unsubscribe/{cid}	✓	✓	✓	✗
Applications				
GET /api/application/autofill-details	✓	✓	✓	✗
POST /api/application/submit-application	✓	✓ (other clubs), ✗ (own club)	✓ (other clubs), ✗ (own club)	✗

GET /api/application/form/{form_id}	✗	✓ (own club)	✓ (own club)	✗
GET /api/application/user	✓	✓	✓	✗
GET /api/application/{application_id}/status	✓ (if applicant)	✓ (own club)	✓ (own club)	✗
PUT /api/application/{application_id}/status	✗	✗	✓ (own club)	✗
GET /api/application/{application_id}	✓ (if applicant)	✓ (own club)	✓ (own club)	✗
DELETE /api/application/{application_id}	✓ (if applicant, before approval/rejection)	✗	✗	✗
PUT /api/application/{application_id}/endorse	✗	✓ (own club)	✓ (own club)	✗
PUT /api/application/{application_id}/withdraw-endorsement	✗	✓ (if endorsed)	✓ (if endorsed)	✗
Recruitment Forms				
POST /api/recruitment/forms	✗	✗	✓ (own club)	✗
GET /api/recruitment/forms/club/{club_id}	✓	✓	✓	✓
GET /api/recruitment/forms/{form_id}	✓	✓	✓	✓
PUT /api/recruitment/forms/{form_id}	✗	✓ (own club)	✓ (own club)	✗
DELETE /api/recruitment/forms/{form_id}	✗	✗	✓ (own club)	✗

GET /api/recruitment/forms/{form_id}/applicants/emails	✗	✓ (own club)	✓ (own club)	✗
Calendar				
GET /api/calendar/events	✓ (own events)	✓ (own + club)	✓ (own + club)	✗
Recommendations				
GET /api/recommendations/clubs	✓	✓	✓	✗
Interviews				
POST /api/interviews/schedule_interviews	✗	✗	✓ (own club)	✗

2.2 Major Design Decisions

ADR 001: System Architecture – Monolithic

Status: Accepted

Context:

Our recruitment platform is being built by a small student team under a tight schedule. The core domain (user profiles, applications, endorsements, scheduling, notifications) is relatively cohesive, and we need rapid development, simple deployment, and straightforward testing. We must meet usability (NFR-1), performance (NFR-2), and maintainability (NFR-6) goals without introducing undue operational overhead.

Alternatives Considered:

- Microservices
- Service-Oriented Architecture (SOA)
- Serverless / Function-as-a-Service

Decision:

Adopt a modular **monolithic** architecture: all subsystems run in a single deployable unit, organized internally via clear package/module boundaries (e.g., MVC or layered).

Rationale:

- **Simplicity & Speed (NFR-6.1):** One codebase and one deployment pipeline accelerates development, testing, and onboarding.
- **Performance (NFR-2.1):** Intra-process calls avoid network hops, reducing latency.
- **Team Expertise:** Team lacks prior microservices experience; avoids "distributed monolith" pitfalls.
- **Deployment Overhead:** No need for service discovery, API gateway, or complex orchestration initially.

Consequences:

+ **Faster time-to-market** with minimal infrastructure complexity.

+ **Easier end-to-end testing** since components share memory space.

– **Scalability Constraints (NFR-4.1):** The entire app must scale together; true horizontal scaling requires full instance replication. Acceptable for initial load.

– **Future Refactoring:** Splitting into microservices later will require careful extraction of modules and APIs.

ADR 002: Database Technology – PostgreSQL

Status: Accepted

Context:

Our domain has relational entities (users, clubs, applications, endorsements, interview slots) with ACID consistency requirements (preferred) and complex queries (joins, filters, transactions). We also want builtin full-text search and JSON support for flexible form fields.

Alternatives Considered:

- MySQL / MariaDB
- NoSQL (MongoDB, DynamoDB)

Decision:

Use **PostgreSQL** as the primary data store.

Rationale:

- **ACID Compliance:** Guarantees data consistency across application workflows.
- **Rich Query Capabilities:** Joins, window functions, full-text search for recruitment portal filters (FR-1.2).
- **JSONB Support:** Flexible storage of dynamic form fields without sacrificing relational integrity.
- **Ecosystem & Tooling:** Mature migration tools, ORMs, backup/restore utilities (NFR-5.3).
- **Community & Support:** Widely adopted in academia and industry.

Consequences:

+ **Strong consistency and transaction support** for critical operations (applications, endorsements).

+ **Efficient complex queries** and indexing strategies for performance (NFR‑2.1).

– **Operational Complexity:** Requires dedicated DBA or managed service to handle backups, tuning, and upgrades.

– **Vertical Scaling Limits:** Large tables may require partitioning or sharding strategies in the future.

ADR 003: Authentication – Apereo CAS

Status: Accepted

Context:

We must integrate with the university's existing identity store so that students and staff use campus credentials. This enables automatic fetching of user attributes (name, email, department) and enforces institutional policies (password rules, MFA). Our system must satisfy security (NFR‑3) and privacy concerns.

Alternatives Considered:

- Custom Auth (store credentials in our database)
- OAuth2 / OpenID Connect via a third‑party IdP

Decision:

Use the university's **Apereo CAS** service for SSO.

Rationale:

- **Seamless Campus Integration:** Users authenticate with campus credentials; no separate password to manage.
- **Auto‑Provisioning:** Fetch user details from LDAP on first login, reducing manual profile setup (FR‑1.3).

- **Security Policies (NFR-3.2, NFR-3.4):** Inherits campus password complexity, session timeout, and MFA rules.
- **Reduced Attack Surface:** No local credential store to secure or rotate.

Consequences:

+ **Single Sign-On** experience and centralized user management.

+ **Offloads password policy and MFA enforcement** to institutional IdP.

– **Dependency on campus CAS availability:** We must implement fallback or maintenance-window handling.

– **Integration Effort:** Requires mapping CAS attributes to our user model and handling CAS ticket flows.

ADR 004: Rendering Strategy – Client-Side Rendering (CSR)

Status: Accepted

Context:

Our user interface demands dynamic interactivity (filtering, live endorsement counts, form validations) and a rich, responsive experience (NFR-1). SEO is secondary since the portal is behind SSO. We need rapid frontend development with hot-reload and a modern toolchain.

Alternatives Considered:

- Server-Side Rendering (Next.js, Nuxt)
- Static Site Generation

Decision:

Implement a **CSR** architecture using **React** bundled by **Vite**, consuming a FastAPI JSON API.

Rationale:

- **Developer Experience:** Vite's fast HMR accelerates UI iteration.
- **Responsiveness (NFR-1.1, NFR-1.3):** Immediate UI updates without full-page reloads.

- **Performance (NFR-2.1):** After initial load, client-side caching and diff-based updates keep interactions snappy.
- **Simplified Backend:** No need to manage server-side templates or hydration.

Consequences:

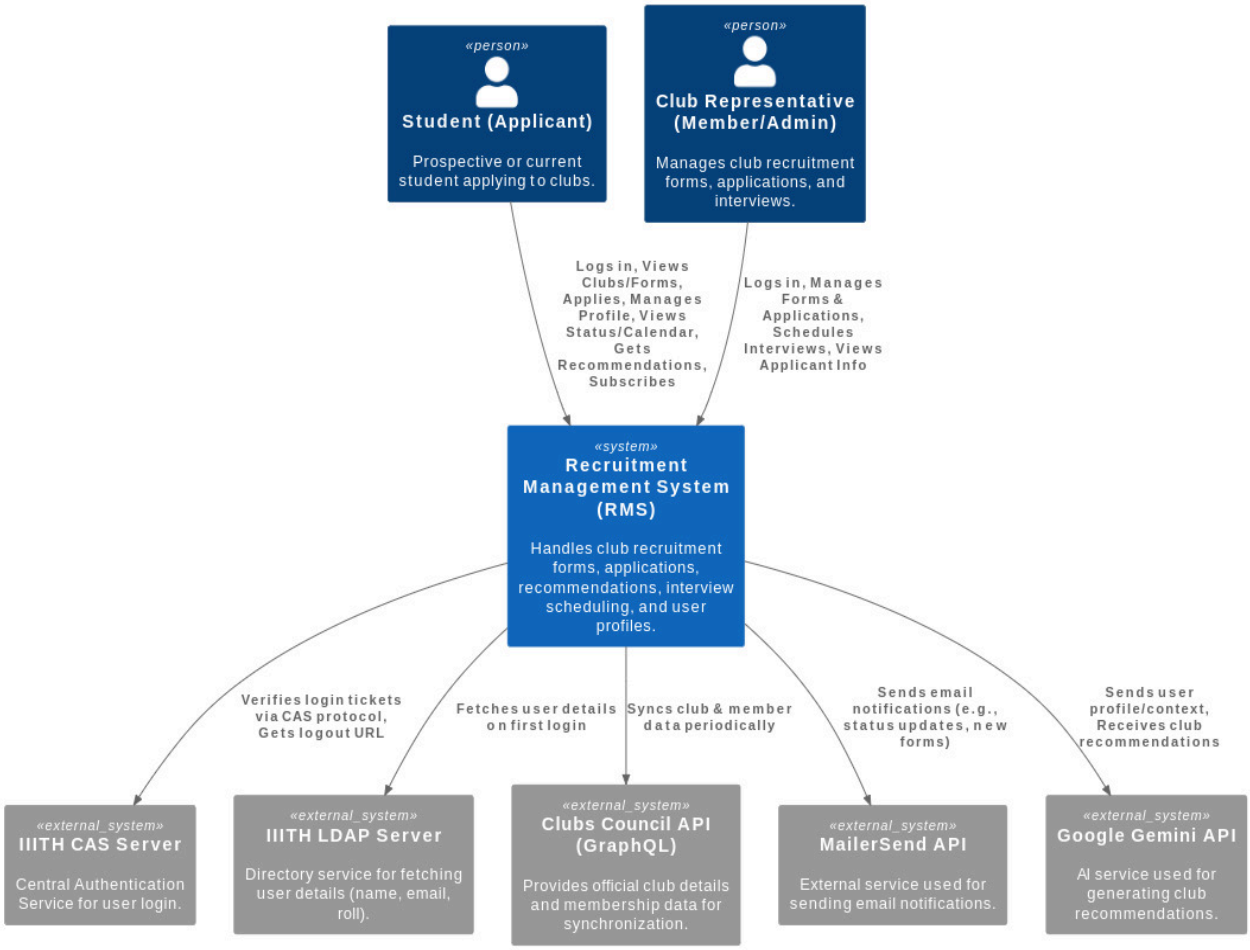
\+ **Rich app-like experience** with minimal latency on interactions.

– **Larger initial bundle:** Mitigated via code splitting and lazy loading.

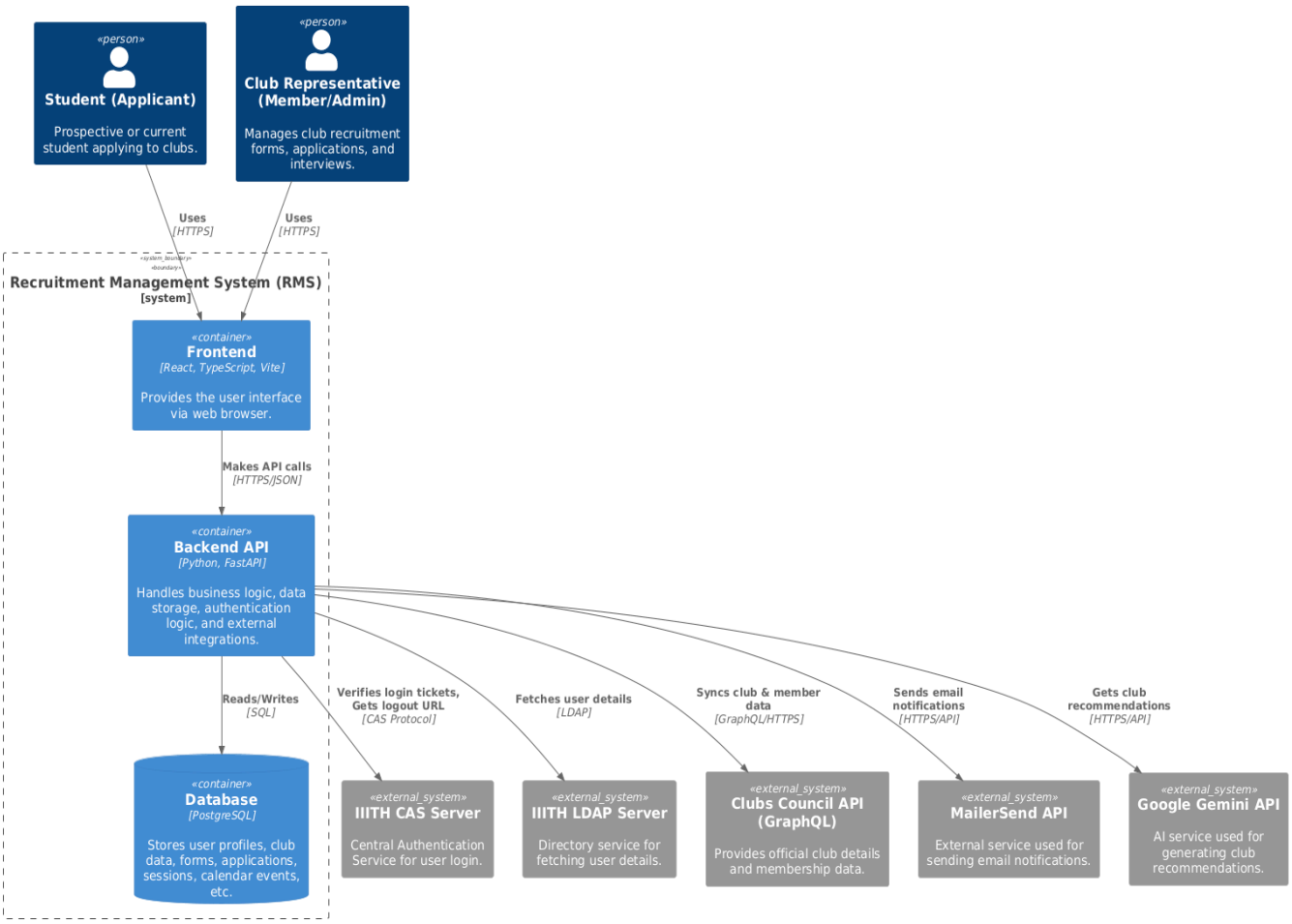
– **SEO Limitations:** Acceptable since the portal is behind SSO; public crawlers not critical.

C4 Models

Context Diagram:

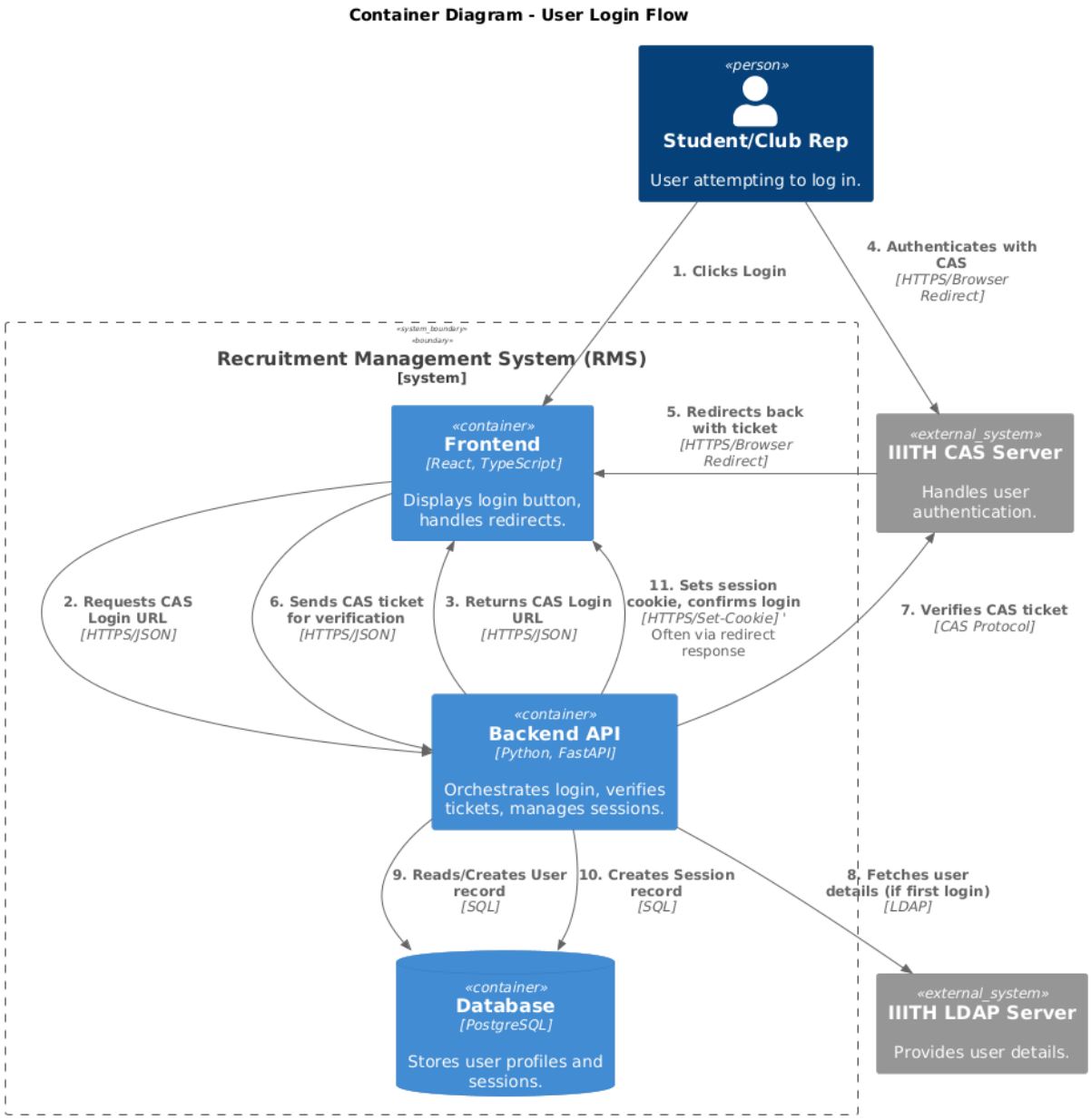


Container Diagram:

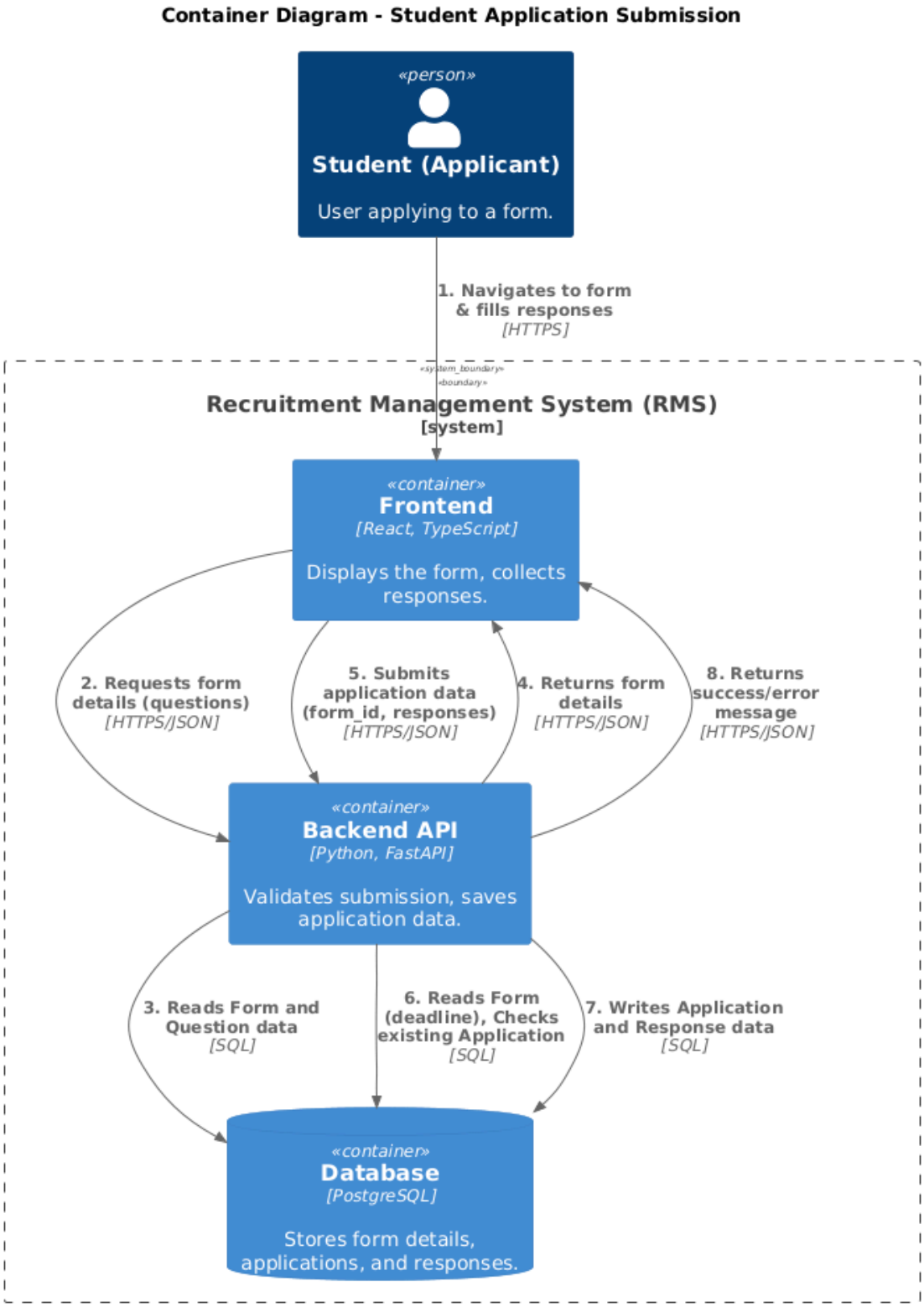


Component Diagrams:

1) User Login:



2) Student Application Submissions:



3 Architectural Tactics and Patterns

3.1 Architectural Tactics

Below are the key tactics we'll employ to satisfy our NFRs in the Club Recruitment Management System. Each tactic is explained in context.

3.1.1 Availability

Addresses: NFR-5 (Reliability & Availability)

3.1.1.1 Exception Handling

What: Catch and handle all unexpected errors at defined boundaries (API layer, background jobs).
How it helps: Prevents unhandled exceptions from crashing the service, returns meaningful error responses, and triggers fallback behavior (e.g., retry or degraded mode).
In our system: If the recommendation engine fails, we fall back to a default "most popular" list instead of taking the portal offline.

3.1.1.2 Health Pings (Liveness & Readiness Checks)

What: Expose /health endpoints that report on service health and dependencies (DB).

How it helps: Orchestrators (Docker Compose) can detect failures and restart only the unhealthy component, minimizing downtime.
In our system: The backend service waits for the Postgres service to start up first. Then, once the healthcheck passes, only then the backend service starts up. This is implemented in docker-compose.yml.

3.1.2 Performance

Addresses: NFR-2 (Performance), NFR-4 (Scalability)

3.1.2.1 Introduce Concurrency

What: Use asynchronous request handling (FastAPI's async endpoints), background workers (Celery/RQ), and database connection pooling.
How it helps: Maximizes resource utilization, handles I/O-bound tasks (emails, recommendations) in parallel, and keeps interactive operations responsive (NFR-2.1).
In our system: All endpoints which require the use of DB are asynchronous, to ensure that

fetching/writing to the DB does not become a bottleneck for the rest of the system, thereby letting it serve other requests while it accesses the DB in the background.

3.1.3 Security

Addresses: NFR-3 (Security)

3.1.3.1 Authentication

What: Integrate with Apereo CAS for single sign-on using campus LDAP.
How it helps: Ensures only valid campus users can log in, leverages institutional MFA and password policies (NFR-3.2, NFR-3.4).
In our system: All student and admin sessions are backed by CAS tickets, removing the need for local credential storage.

3.1.3.2 Authorization

What: Enforce Role-Based Access Control (RBAC) at every API endpoint and UI action.
How it helps: Prevents unauthorized access—e.g., club members can endorse but cannot approve applications(NFR-3.2).
In our system: A user with "Club Admin" role for Club A can manage Club A's interviews but cannot access Club B's settings.

3.1.3.3 Data Confidentiality

What: Encrypt all data in transit (TLS) and sensitive fields at rest (AES-256).
How it helps: Protects PII and application data from eavesdropping or database leaks (NFR-3.1).
In our system: Endorsement comments and application attachments are stored encrypted; backups also encrypted.

3.1.4 Modifiability

Addresses: NFR-6 (Maintainability & Extensibility)

3.1.4.1 Semantic Coherence

What: Group related functionality into well-defined modules (e.g., Application, Interview, Notification).
How it helps: Changes in one module (e.g., adding a new notification type) have minimal ripple effects on others.
In our system: The Recommendation Engine is a standalone package with its own interface—swapping algorithms won't affect core application flows.

3.1.4.2 Information Hiding

What: Expose only necessary interfaces; keep internal data structures and helper functions private.
How it helps: Reduces coupling and accidental misuse of internal APIs (NFR-6.1).
In our system: The Notification Subsystem exposes a single sendEmail(event, payload) API; it hides SMTP details and retry logic.

3.1.5 Testability

Addresses: NFR-5.2 (Reliability), NFR-6.2 (Documentation)

3.1.5.1 Built-In Monitors

What: Instrument key operations with health metrics and traces (e.g., application submission rate, email queue length).
How it helps: Enables automated checks, alerts on abnormal behavior, and supports end-to-end tests that validate service health.
In our system: A monitor ensures that if the Application Management subsystem's average processing time exceeds a threshold, alerts are raised for investigation.

3.1.6 Usability

Addresses: NFR-1 (Usability)

3.1.6.1 UI Separation

What: Keep the React/Vite front-end completely decoupled from the FastAPI backend via a well-defined JSON API.
How it helps: Front-end and back-end teams can iterate independently; UI can be replaced or upgraded without touching server code (NFR-1.1).
In our system: All form definitions, validation rules, and content come via API, enabling dynamic UI updates.

3.1.6.2 Cancel & Undo

What: Server-side deletion of applications.
How it helps: Improves user confidence and reduces error impact (NFR-1.3).
In our system: A student can withdraw (delete) their application anytime before their application gets accepted/rejected.

3.2 Implementation Patterns

3.2.1 Strategy Pattern [Design Pattern]

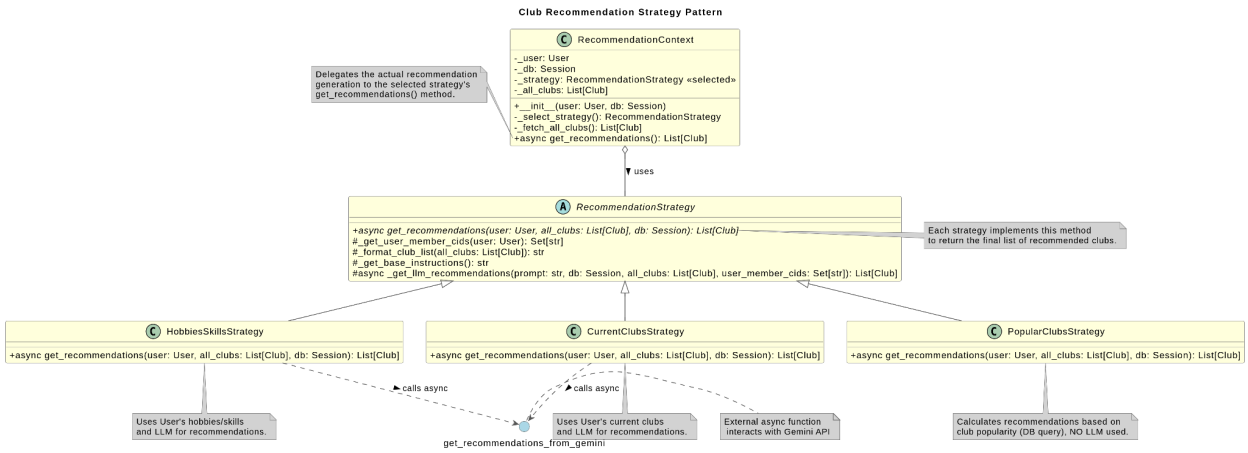
The Strategy design pattern is employed within the club recommendation subsystem to manage different algorithms for generating personalized club suggestions. The core idea of the Strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. This allows the algorithm to vary independently from the clients that use it.

The "family of algorithms" represents the different methods used to determine club recommendations:

- Based on the user's Hobbies and Skills.
- Based on the user's Current Club memberships.
- Based on general Club Popularity.

The RecommendationContext acts as the "client" that needs a recommendation list but doesn't need to know the specifics of *how* it's generated. The pattern enables the RecommendationContext to select the most appropriate recommendation algorithm (Strategy) at runtime based on the user's available profile data and then delegate the actual recommendation generation task to the selected strategy object.

1. Class Diagram



2. Implementation Details

The implementation follows the classic structure of the Strategy pattern, as illustrated in the UML diagram:

RecommendationStrategy (Abstract Strategy):

- This abstract base class defines the common interface for all concrete recommendation algorithms.
- Its key method is async get_recommendations(user: User, all_clubs: List[Club], db: Session): List[Club], which must be implemented by all subclasses.
- It may also contain shared helper methods (like _get_user_member_cids, _format_club_list, _get_llm_recommendations) utilized by one or more concrete strategies.

Concrete Strategies:

- These classes inherit from RecommendationStrategy and provide specific implementations of the get_recommendations method.
- **HobbiesSkillsStrategy**: Implements the algorithm using user hobbies/skills, formatting a prompt, and calling the external get_recommendations_from_gemini function (LLM).
- **CurrentClubsStrategy**: Implements the algorithm using the user's current clubs, formatting a different prompt, and also calling the LLM via get_recommendations_from_gemini.
- **PopularClubsStrategy**: Implements the algorithm by directly querying the database for popular clubs based on subscriber counts, filtering the results, and does *not* involve an LLM call. This difference is clearly shown in the UML notes and dependencies.

RecommendationContext (Context):

- This class maintains a reference to a RecommendationStrategy object (_strategy attribute).
- Its _select_strategy() method determines which concrete strategy instance to create based on the user's data (e.g., presence of hobbies, skills, or current clubs).
- The main async get_recommendations() method in the RecommendationContext delegates the core work by calling self._strategy.get_recommendations(...), executing the chosen algorithm without needing explicit knowledge of which one it is.
- It holds contextual information needed by the strategies (_user, _db, _all_clubs).
- The UML shows a "uses" relationship (aggregation) from RecommendationContext to RecommendationStrategy.

3. Rationale for Using the Strategy Pattern

The primary reason for choosing the Strategy pattern was to handle the requirement that recommendations should be generated differently based on the available user data (FR-5.2 implicitly requires varied approaches).

- **Multiple Algorithms:** There isn't a single "one-size-fits-all" algorithm for club recommendations. The best approach depends on whether the user has provided hobbies/skills, is already in clubs, or is new/has minimal data.
- **Avoiding Complex Conditionals:** Implementing these different algorithms using large if/else if/else blocks within the RecommendationContext would lead to complex, hard-to-maintain code. Each time a new condition or algorithm was added, the central logic would need modification, violating the Open/Closed Principle.
- **Decoupling:** The pattern decouples the algorithm selection logic (RecommendationContext._select_strategy) from the algorithm implementation details (each Concrete Strategy class). The context only needs to know the common RecommendationStrategy interface.

4. Quality Attributes (Benefits) Provided

Using the Strategy pattern yields several significant advantages related to software quality:

1. **Flexibility & Extensibility:** New recommendation algorithms (e.g., based on user-followed events, collaborative filtering) can be introduced easily by creating a new class implementing RecommendationStrategy and updating the _select_strategy logic in the context. Existing strategies remain untouched.
2. **Maintainability:** Each recommendation algorithm is encapsulated within its own class. This isolation makes the code easier to understand, debug, test, and modify. If the logic for recommending based on hobbies needs changes, only the HobbiesSkillsStrategy class is affected.
3. **Adherence to SOLID Principles:**
    - **Single Responsibility Principle:** Each strategy class has a single responsibility, implementing one specific recommendation algorithm. The context's responsibility is orchestration and strategy selection.
    - **Open/Closed Principle:** The system is open for extension (adding new strategies) but closed for modification (the core context logic and existing strategies don't need to change to add a new one).
4. **Improved Testability:** Each concrete strategy can be tested independently, potentially mocking dependencies like the database session or the LLM API call (get_recommendations_from_gemini). The context can also be tested by providing mock strategy objects.
5. **Simplified Context:** The RecommendationContext remains relatively simple, as the complexity of each specific recommendation algorithm is delegated to the respective strategy objects.

3.2.2 Observer Pattern [Design Pattern]

We apply the observer design pattern within the Recruitment Management System to handle club subscriptions and subsequent notifications to users. While the RMS backend, built with FastAPI and SQLAlchemy, doesn't strictly adhere to classical object-oriented implementations with explicit Subject and Observer classes, the underlying principles and behavioral characteristics of the Observer pattern are effectively realized through its functional and data-driven approach.

In the RMS, the club subscription feature allows users to "follow" clubs and receive email notifications about events related to those clubs, such as the creation, update, or deletion of recruitment forms. This mechanism functionally implements the Observer pattern as follows:

Subject: The **Club** entity acts as the conceptual Subject. Events related to a specific club (like a new form being added *to that club*) represent the "state change" that Observers care about. There isn't a single Club *object* in memory holding state and notifying; rather, the *system events associated with a club ID* trigger the notification process.

Observer: A **User** who has subscribed to a specific Club acts as an Observer for events related to that Club.

Attach/Detach Mechanism:

- **Attach:** When a user calls the POST /api/club/subscribe/{cid} endpoint, the subscribe function in models/clubs/clubs_config.py adds the user.uid and club.cid association to the club_subscribers database table. This table effectively represents the list of registered Observers for each Club (Subject).
- **Detach:** When a user calls the POST /api/club/unsubscribe/{cid} endpoint, the unsubscribe function removes the association from the club_subscribers table.

Notification Trigger & Logic (notify() equivalent): This responsibility is distributed within the code that handles the relevant events (the "state changes"). For example, in routers/recruitment_router.py:

- Inside create_new_form, update_existing_form, and delete_existing_form functions, *after* the primary action (creating/updating/deleting the form in the database) is completed:
    a. The relevant club_id is identified from the form data.
    b. The function get_all_subscribers(db, club_id) (from models/clubs/clubs_config.py) is called. This function queries the club_subscribers table to retrieve the list of User objects (Observers) associated with that club_id (Subject).

c. The function inform_users(subscribers, subject, content) (from models/users/users_config.py) is called with the list of subscribers.

Update Mechanism (update() equivalent):

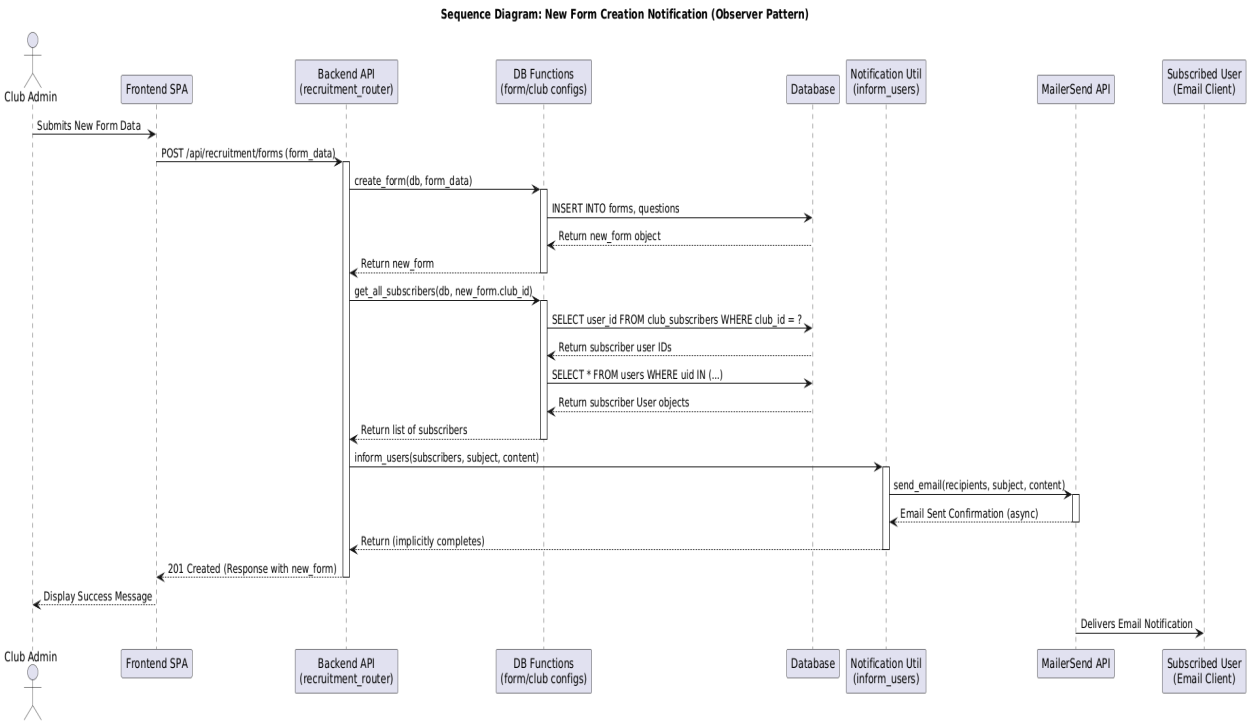
- The inform_users function iterates through the list of subscriber User objects.
- It formats the recipient list for the email service.
- It calls the send_email(recipients, subject, content) utility (from utils/mail_utils.py).
- The send_email function interacts with the external MailerSend API to dispatch the actual email.
- **Receiving the email** is the effective "update" for the User (Observer). They are informed about the event related to the Club (Subject) they subscribed to.

While the RMS doesn't use explicit Subject and Observer classes inheriting from base interfaces, the *behavioral contract* of the Observer pattern is fulfilled:

1. **One-to-Many Dependency:** The club_subscribers table establishes a clear one-to-many relationship between a Club (one) and its Subscribed Users (many).
2. **Automatic Notification:** When a relevant event occurs (e.g., form creation), the system *automatically* identifies the dependents (subscribers) and triggers a notification mechanism (inform_users -> send_email) without the event-originating code needing to know the specifics of *how* each user is notified.
3. **Loose Coupling:**
    - The form management logic (create_form, update_form, etc.) doesn't need to know *who* is subscribed or *how* emails are sent. It only needs to know the club_id and trigger the notification process (get_all_subscribers + inform_users).
    - The user subscription logic (subscribe, unsubscribe) is separate from the event-triggering logic.
    - The notification mechanism (inform_users, send_email) is separate and can be modified (e.g., switch email providers, add different notification types later) without changing the form management code.

Therefore, the system achieves the *decoupling* and *automatic notification* goals of the Observer pattern through database relationships, function calls, and API endpoints, effectively mirroring the pattern's intent and structure in a functional/procedural style.

1. Sequence Diagram



2. Rationale for Using the Observer Pattern Principles

The core reason for adopting this pattern (even implicitly) is to **decouple** the components involved in club events from the components responsible for notifying users about those events.

- **Separation of Concerns:** The logic for creating/updating/deleting forms should focus solely on form data management. It shouldn't be burdened with the details of finding subscribers or sending emails.
- **Maintainability:** If the notification mechanism changes (e.g., switching email providers, adding in-app notifications), the changes are localized primarily within the inform_users and send_email functions (and potentially get_all_subscribers if the storage method changed), without requiring modifications to every piece of code that *triggers* a notification (like form creation, updates, etc.).
- **Extensibility:** Adding new types of events that require notification (e.g., a club posting a general announcement) becomes easier. The new event-handling code simply needs to identify the relevant club_id and call the existing get_all_subscribers and inform_users functions. Similarly, adding new *types* of Observers (e.g., sending notifications to a webhook) could be implemented by modifying inform_users without changing the Subjects (event triggers).

3.2.13 Observer Pattern [Architectural Pattern]

Layer 1: Presentation Layer

Components	Frontend Single-Page Application (SPA) built with React and TypeScript (frontend/).
Responsibility	Renders the user interface in the web browser, handles user input events, manages UI state, and communicates with the Backend API to fetch data and trigger actions. This includes all pages (pages/), components (components/), layout elements (layout/), and routing (CreateRouter.tsx).
Interaction	Interacts only with the Application Layer (Backend API) via HTTP requests (typically JSON). It has no knowledge of the backend's internal logic, persistence mechanisms, or the database itself.

Layer 2: Application Layer (Business Logic)

Components	Backend API built with FastAPI (backend/). This includes:
	main.py: Application entry point, middleware setup.
	Routers (routers/*.py): Define API endpoints, handle HTTP request/response validation (using Pydantic schemas from schemas/), parse requests, and call appropriate business logic functions.
	Business Logic/Configuration (models/*/config.py): Contain the core functions implementing system features (e.g., create_form, process_submitted_application, user_login_cas, get_recommendations, allocate_calendar_events). These functions orchestrate operations, enforce business rules, and utilize the Persistence Layer for data operations.
Responsibility	Exposes the system's functionality via API endpoints. Implements business rules and workflows (e.g., checking deadlines, validating user roles, calculating recommendations, triggering notifications). Orchestrates data operations by interacting with the Persistence Layer. Handles authentication and session management logic. Interacts with external systems (CAS, LDAP, CC API, MailerSend, Gemini).
Interaction	Receives requests from the Presentation Layer. Uses services from the Persistence Layer (SQLAlchemy models and session management) to interact with data. Calls external systems via utility functions. It does not directly interact with the Database Layer (e.g., executing raw SQL).

Layer 3: Persistence Layer (Data Access)

Components	SQLAlchemy ORM Models (models/*/*_model.py): Define the structure of the data entities (Users, Clubs, Forms, Applications, etc.) and their relationships, mapping them to database tables.
	Database Utilities (utils/database_utils.py): Manages the database connection (engine), session lifecycle (SessionLocal, get_db), and provides functions for initializing/resetting the database schema based on the ORM models.
Responsibility	Renders the user interface in the web browser, handles user input events, manages UI state, and communicates with the Backend API to fetch data and trigger actions. This includes all pages (pages/), components (components/), layout elements (layout/), and routing (CreateRouter.tsx).
Interaction	Interacts only with the Application Layer (Backend API) via HTTP requests (typically JSON). It has no knowledge of the backend's internal logic, persistence mechanisms, or the database itself.

Layer 4: Database Layer

Components	PostgreSQL Database Server.
Responsibility	Persistently stores and manages the application's data (users, clubs, forms, applications, sessions, etc.). Executes SQL queries received from the Persistence Layer (via the database driver). Ensures data integrity through constraints, handles transactions, concurrency, and backups.
Interaction	Interacts only with the Persistence Layer (SQLAlchemy engine/driver).

Rationale for Using Layered Architecture in RMS

Adopting a layered structure, even implicitly, provides several advantages for the RMS:

1. **Clear Separation of Concerns:** It cleanly divides the system into distinct areas of responsibility: UI (React), API & Business Rules (FastAPI), Data Mapping & Access (SQLAlchemy), and Data Storage (PostgreSQL). This makes the system easier to understand and reason about.
2. **Improved Maintainability:** Changes within one layer have a reduced impact on others, provided the interfaces (API contracts, function signatures between Application and

Persistence) remain stable. For instance, a significant UI redesign in the React frontend should not require changes to the backend business logic or database schema. Similarly, optimizing a database query within the Persistence Layer shouldn't affect the API endpoint definitions.

3. **Enhanced Testability:** Layers can be tested more independently. The Frontend SPA can be tested against mock API responses. The Backend API (Application Layer) can be tested by mocking the Persistence Layer (SQLAlchemy calls) or using an in-memory database, allowing business logic validation without relying on a full database setup.
4. **Technology Flexibility:** It allows for replacing technologies within a layer with less disruption. For example, the frontend could potentially be rewritten using a different framework (like Vue or Angular) as long as it interacts with the same Backend API. The database could be migrated from PostgreSQL to another SQL database with changes primarily localized to the Persistence Layer (SQLAlchemy configuration and potentially minor model adjustments).

5 Prototype Implementation and Analysis

5.1 Prototype Development

Repo link: github

5.2 Architecture Analysis

Our primary system operates as a containerized monolithic backend with a frontend querying it on a single domain. In a separate branch, we also developed and maintained a separate microservice focused specifically on generating club recommendations for users. This allows us to draw direct comparisons based on our practical experience with both architectures within our specific context.

Qualitative Comparison:

1. **Development Simplicity and Speed:**
    - **Monolith:** The development process for the main monolithic system is generally simpler and faster. Components communicate through direct function calls within the same codebase, reducing the need for complex inter-service communication protocols or API designs. This familiarity allows our team to iterate quickly and reduces the potential for errors often associated with less familiar architectures. Deployment is straightforward as it involves a single unit.

- **Microservices:** While developing the recommendation microservice, we observed increased complexity. Managing separate deployment pipelines, API versioning, and inter-service communication (even if relatively simple for this one service) added overhead compared to extending the monolith. Scaling and managing just this one microservice required specific attention and tooling.

2. **Performance and Resource Utilization:**

    - **Monolith:** For core recruitment functions, the monolithic architecture demonstrates strong performance. Direct integration of components minimizes network latency and communication overhead. This efficiency is crucial for a system that needs to handle potentially high loads during peak recruitment periods, ensuring a responsive user experience without excessive resource consumption.
    - **Microservices:** Our recommendation microservice, while isolated, introduces inherent network latency for its interactions with the main system to fetch user data (like skills/hobbies) and club information. While acceptable for its specific, less time-critical function, extending this model to core features like form submission or application processing would likely degrade the perceived performance due to the cumulative effect of inter-service communication overhead. We noted in a similar system analysis that microservices incurred additional overhead affecting response times compared to simpler function calls in a monolith.

3. **Operational Complexity and Monitoring:**

    - **Monolith:** Monitoring and debugging the monolithic system is relatively centralized. Tracing requests and identifying issues typically occurs within a single codebase and logging system. Managing a single containerized application simplifies operational tasks.
    - **Microservices:** Even with just the recommendation microservice, we require separate monitoring for its resources, logs, and performance. Expanding to multiple microservices would significantly increase the complexity of our monitoring infrastructure, distributed tracing, and managing potential points of failure across the network.
4. **Consistency and Data Management:**

    - **Monolith:** Ensuring data consistency across different parts of the recruitment process (e.g., user profiles, applications, club details) is easier within the monolith, often managed through shared database transactions.
    - **Microservices:** With the recommendation service, data consistency is less critical as it primarily reads data. However, if core functionalities were split, managing distributed transactions or ensuring eventual consistency across separate service databases would introduce significant challenges and potential complexities, increasing the risk of data-related issues.

5. **Extensibility and Flexibility (Trade-offs):**

    - **Monolith:** While often cited as a drawback, modifying or extending the monolith for closely related features within the recruitment domain has been manageable due to the unified codebase. However, making large structural changes or adopting completely new technologies for specific parts can be more challenging.
    - **Microservices:** For our core recruitment features, which are tightly coupled, the benefits of independent scaling offered by microservices seem outweighed by the increased development and operational overhead.

Justification for Monolithic Design:

Based on our experience, the monolithic architecture currently provides the best balance for our core recruitment management system. The key justifications are:

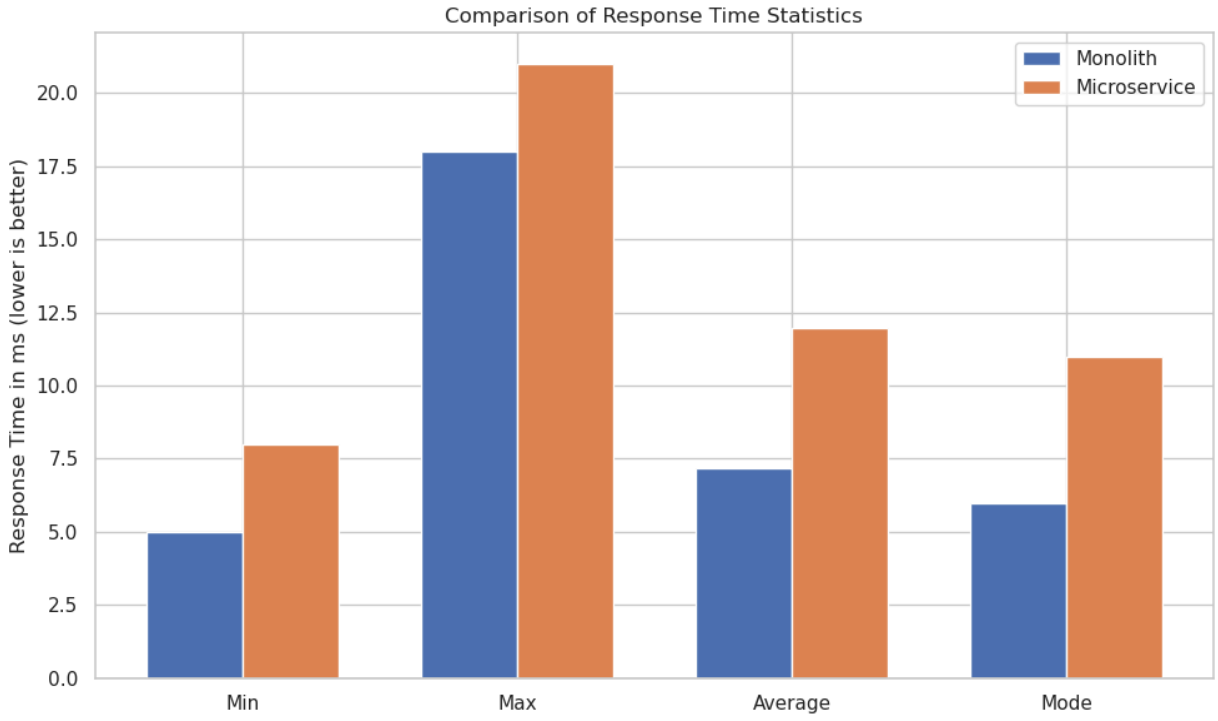
- **Performance:** The low latency and efficiency of the monolith are critical for the user-facing recruitment workflows.
- **Development Velocity:** The simplicity of the monolithic codebase allows our team to develop and deploy features for the core system more rapidly and with fewer potential integration issues compared to managing multiple services.
- **Operational Simplicity:** Managing a single application is less complex in terms of deployment, monitoring, and debugging.
- **Team Familiarity:** Our team is well-versed in developing and maintaining the monolithic application, reducing risks associated with adopting less familiar architectural patterns for core functionalities.

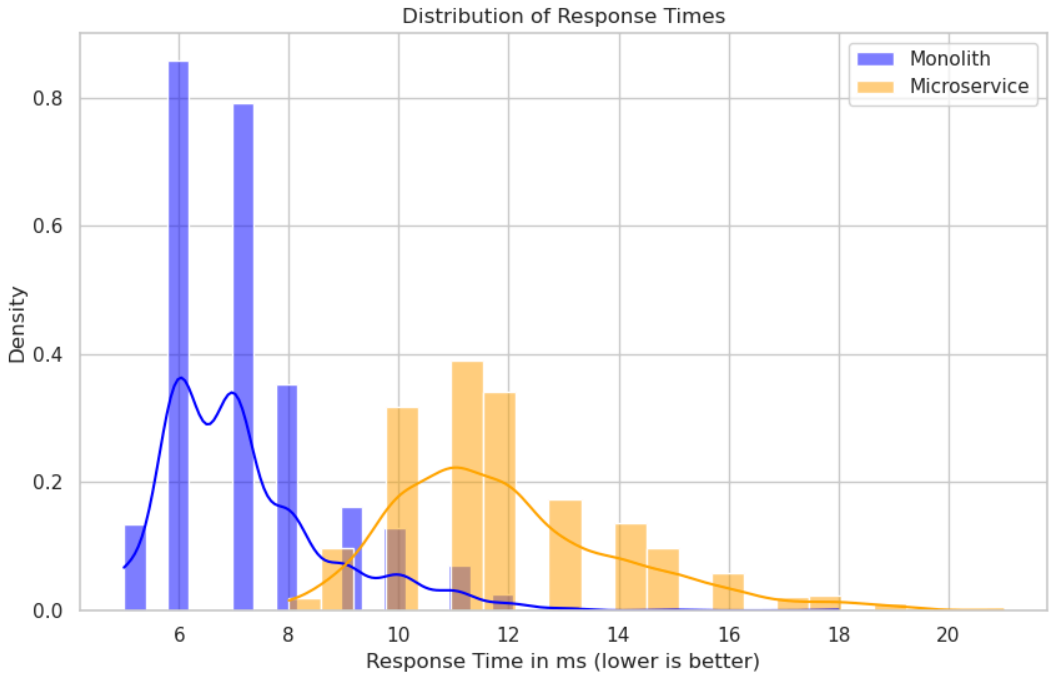
While the recommendation microservice proves the value of a service-based approach for *specific, isolated, and computationally distinct* functionalities, the core recruitment processes benefit significantly from the coherence, performance, and simplicity of the monolithic design. The observed overheads associated with even a single microservice reinforce the decision to maintain the monolith for the system's primary responsibilities.

Quantitative Comparison:

We compared the response time for both the monolith and microservice architectures, in fetching the “recommendations” data. We have done this without calling the Gemini API, as we didn’t want the non-deterministic nature of calling an LLM to influence the response time. For the purposes of this test, we wrote a deterministic function that just picks 5 clubs at random.

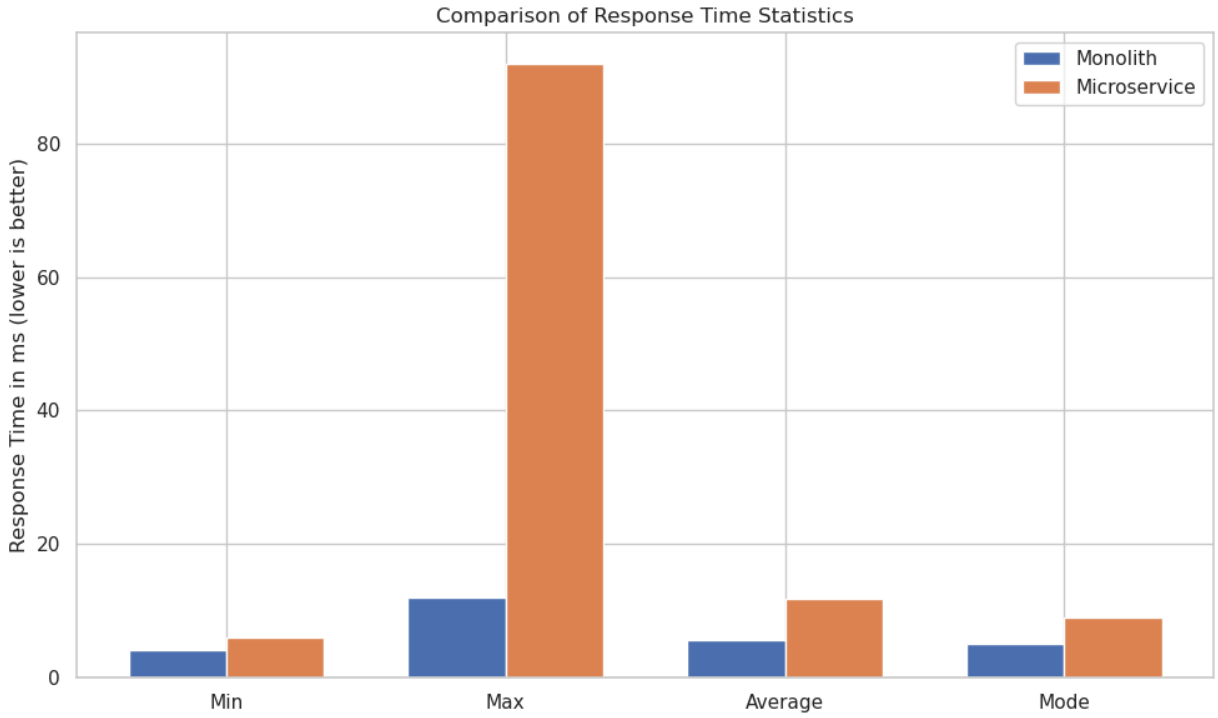
The following two graphs show the distribution of response times across 1000 tests, with a delay of 5 ms between tests.





As can be seen above, the **microservice** architecture is much worse compared to the **monolith** architecture for our system in **every conceivable way.** This is due to the additional latency introduced due to the requests in the microservice architecture having to communicate amongst themselves in order to obtain all the necessary data before producing an answer. Monolith however, does not have this additional latency as all the necessary functions are present on the same service.
In the distribution of response times graph above, we can see that the monolith architecture takes much lower time and is much more consistent with the time it takes for each request. This is not the case on both counts with the microservice architecture.

Additional statistics: The inconsistency in response times can be observed better in the graph below, when we ran each of the architectures 10000 times. Clearly, the microservice architecture here performed **much worse**, as in a couple of instances, it took upwards of 80 ms to fetch the required details a single time! The density graph for this was extremely skewed as a result, and we have omitted showing it here for the sake of sanity.



6 Reflections

The major learning from this project was that creating an end-to-end product is much more than just coding it up. Moreover, We found that laying the groundwork before beginning implementation goes a long way in helping streamline the process of development. We identified the importance of ADRs in bigger projects, as they help us effectively document our decisions that we made and help us be conscious in our architectural decisions.