# PROJECT 3 REPORT

## Software Engineering | Project - 3
## Team - 28

---


**Team Members**

| | |
|---|---|
| Harshita | 2023201002 |
| Shreyansh Shrivastava | 2023201006 |
| Aman Khurana | 2023201017 |
| Kote Sai Kiran | 2023201067 |
| Pradhyumna Palore | 2023202022 |


# 1. Introduction

In today's fast-paced academic environment, researchers are constantly overwhelmed by the vast volume of research material—ranging from papers, citations, notes, to review comments and visual analyses. Efficiently managing this growing body of research information remains a significant challenge. To address this issue, we designed and developed **Researcher's Hive** —a comprehensive research management platform that streamlines the process of discovering, organizing, annotating, and visualizing research knowledge.

Researcher's Hive serves as a personal assistant for researchers. It integrates intelligent search, structured data storage, AI-assisted commenting, graph-based visualizations, and real-time recommendation systems. The primary goal of the platform is to provide users with an end-to-end solution for managing their academic research lifecycle.

The project architecture is built using a modern stack that ensures scalability, modularity, and security. The frontend is built using **ReactJS** (with Vite for bundling) offering a dynamic and responsive single-page application (SPA) experience. The backend uses **Django**, a powerful and secure Python web framework that allows for clean modularization of applications. MongoDB is used as the primary database due to its flexibility in storing research documents and annotations. Additionally, third-party

APIs such as **Semantic Scholar** and **Gemini** are integrated using the Adapter design pattern to support paper retrieval and AI-enhanced commenting.

Researcher's Hive  also focuses on adhering to robust software engineering principles. The implementation follows key design patterns like Adapter, Chain of Responsibility, , and Specification to ensure modularity, extensibility, and ease of maintenance. Furthermore, architectural tactics such as **caching**, **port shielding using nginx**, **secure session management**, and **bcrypt-based password hashing** have been applied to meet crucial non-functional requirements like security, performance, and scalability.

By combining a user-centric approach with modern software design principles, Researcher's Hive  not only addresses the current pain points faced by academic researchers but also sets the foundation for future enhancements like collaborative features, mobile app integration, and AI-powered analytics.

---

# 2. Functional and Non-Functional Requirements

## 2.1 Functional Requirements

The following are the core functional requirements implemented in Researcher's Hive :

1. **User Registration and Login:**
    - Users can create an account and securely log in.
    - Authentication is handled via Django and sessions are tracked using token-based authentication.
2. **Search and Retrieval of Research Papers:**
    - Users can search research papers based on keywords/topics.
    - Semantic Scholar's API is integrated to fetch relevant metadata and content.
3. **Commenting and Annotation System:**
    - Users can comment on papers using text, tables, and images.
    - AI-based enhancements can be applied to improve note formatting.
4. **Dashboard with Recommendations:**
    - Personalized recommendations are shown based on reading history and matching keywords.
    - Conference links are provided for paper submissions.
5. **Profile Page with Paper and Comment History:**
    - Tracks previously read papers and user's personal annotations.
    - Filters for searching papers.

    - Allows editing of notes from the same interface.
6. **Graph-based Visualization:**
    - Shows relationships between papers, authors, and citations.
    - Enhances comprehension of research ecosystems.
7. **Real-Time Alerts:**
    - Notifies users when a newly opened paper matches previously read papers.

---

## 2.2 Non-Functional Requirements

The architecture is designed to fulfill the following non-functional requirements:

1. **Security:**
    - Passwords are stored using bcrypt hashing.
    - nginx serves as a reverse proxy to protect backend endpoints.
    - Session management with token-based expiration is implemented to secure user data.
2. **Performance:**
    - Caching mechanisms are used to store frequently accessed user data and dashboard elements.
    - API data is optimized to reduce repeated calls.
3. **Scalability:**
    - Modular Django apps ensure each feature can be developed and scaled independently.
    - Frontend and backend are decoupled, allowing separate deployments.
4. **Maintainability:**
    - Follows separation of concerns using well-defined design patterns.
    - Clean folder structure and consistent API endpoints.
5. **Reliability:**
    - Sessions are auto-expired after inactivity.
    - Alerts are handled asynchronously to avoid blocking UI interaction.
6. **Usability:**
    - built using responsive React components.
    - User feedback and AI-enhanced comments improve user engagement.

---

# 3. Subsystem Overview

The Researcher's Hive  system is composed of multiple subsystems, each responsible for a core functionality. These subsystems are loosely coupled and interact through well-defined interfaces and APIs. The modular design ensures that future enhancements or bug fixes can be made to individual components without affecting the entire system.

## 3.1 Authentication Subsystem

- **Responsibilities:** Manages user registration, login, session generation, and secure password storage.
- **Technologies Used:** Django's built-in auth system, bcrypt for password hashing, session middleware.
- **Security Features:** Token-based sessions with expiration; bcrypt with salting for password protection.

## 3.2 Paper Retrieval and Search Subsystem

- **Responsibilities:** Enables users to search for academic papers based on keywords and topics.
- **Technologies Used:** ReactJS for search UI, Django API handlers, Semantic Scholar API for external data.
- **Integration Design:** Adapter pattern is used to abstract the interaction with Semantic Scholar, making it easily replaceable with other APIs.

## 3.3 Commenting and Annotation Subsystem

- **Responsibilities:** Allows users to add rich comments (text, tables, images) to papers; supports AI-based enhancement of raw comments.
- **Technologies Used:** Google Palm API (via Adapter), ReactJS comment editor.
- **Design Focus:** Chain of Responsibility (COR) pattern used in comment processing pipeline (text → format → AI-enhance → save).

## 3.4 Visualization Subsystem

- **Responsibilities:** Displays graphical representations of paper relationships (authors, citations).
- **Technologies Used:** D3.js or similar JS visualization libraries; backend endpoints provide citation/author data.
- **Purpose:** Enhances user understanding of research ecosystems.

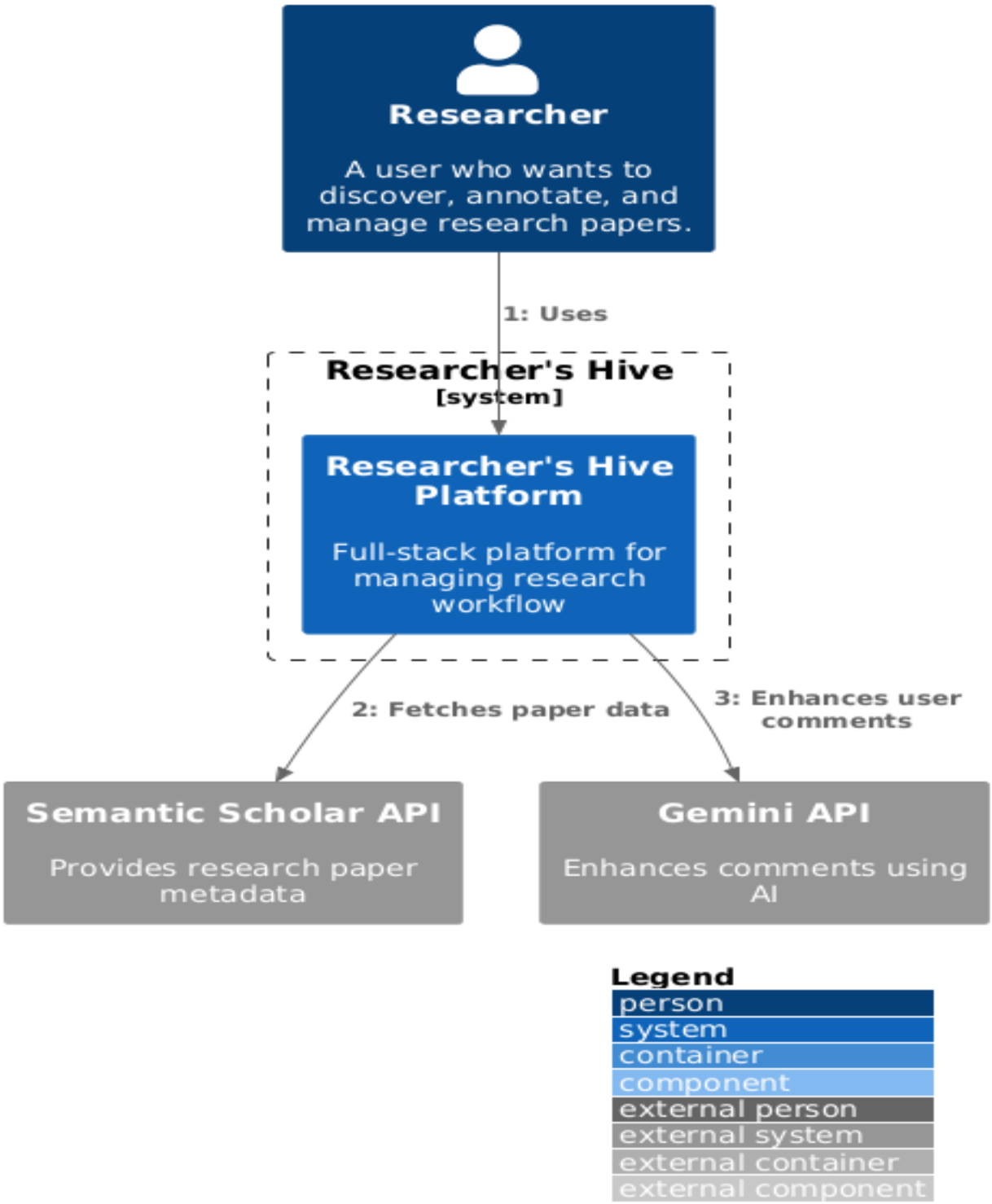
### 3.5 Profile and History Subsystem

- **Responsibilities:** Shows recently read papers, user's saved notes, editable comments.
- **Features:** Allows updating of annotations; supports historical browsing and alerts for paper overlaps.

---

# C4 Diagram: Researcher′s Hive

CONTEXT


Researcher
A user who wants to discover, annotate, and manage research papers.
1: Uses
Researcher's Hive
[system]
Researcher's Hive Platform
Full-stack platform for managing research workflow
2: Fetches paper data
3: Enhances user comments
Semantic Scholar API
Provides research paper metadata
Gemini API
Enhances comments using AI
Legend
person
system
container
component
external person
external system
external container
external component

# C4 Container Diagram:

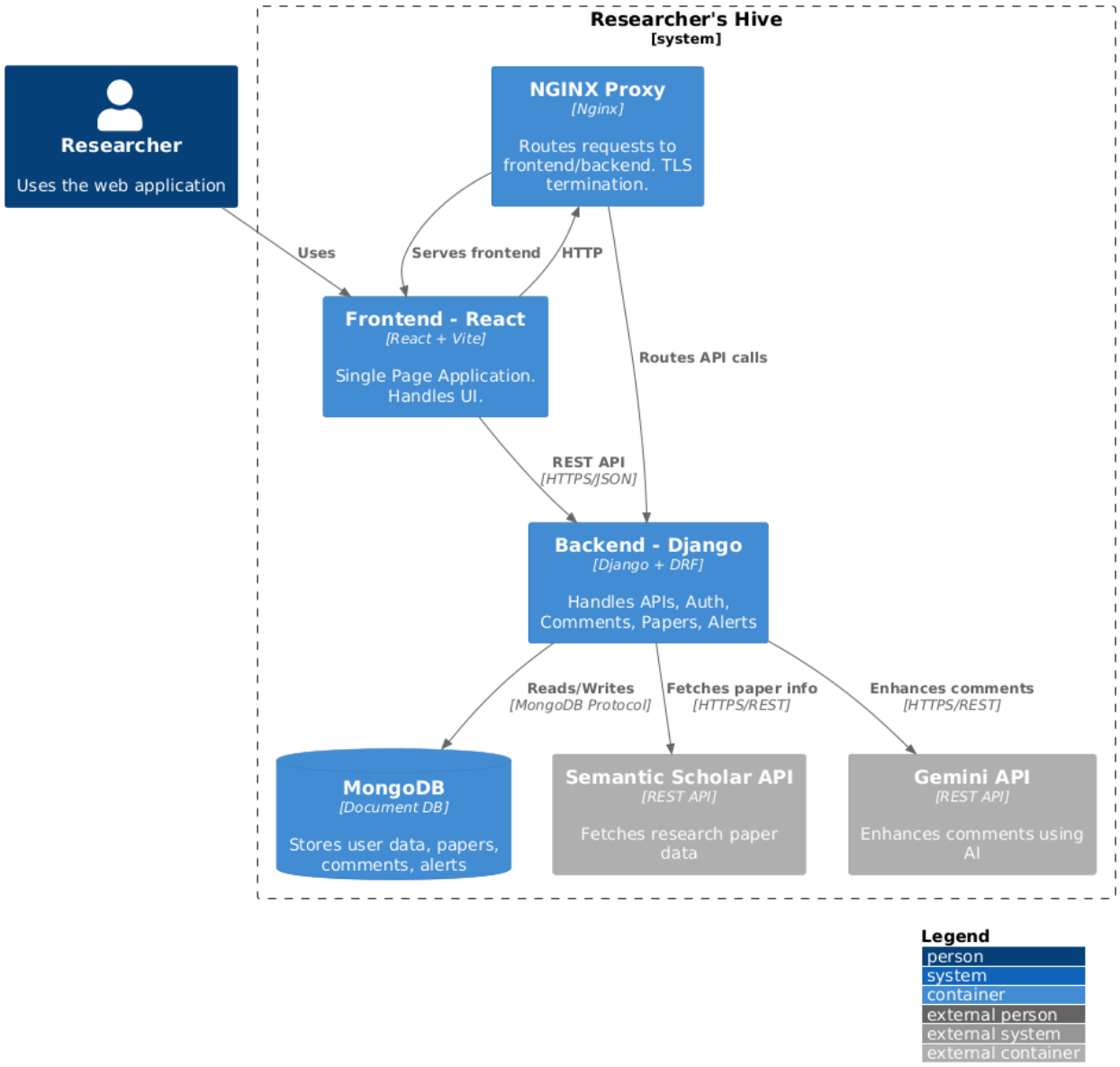
## Components Description

| Component | Purpose |
|---|---|
| **Web Browser** | UI for users: search papers, view notes, receive alerts, etc. |
| **React Frontend** | Handles routing, UI components, user interaction (SPA). |
| **nginx** | Secures ports, routes requests, handles TLS. |
| **Django Backend** | Exposes APIs for auth, paper info, comments, visualization, and alerts. |
| **MongoDB** | Stores user data, paper info, annotations, and caches. |
| **External APIs** | Semantic Scholar: fetch paper metadata; GooglePalm: AI enhancements. |

---

## Interactions

- React fetches data via **REST API**.
- Django backend uses **Chain of Responsibility** to handle paper info.
- **Adapter Pattern** wraps API logic to unify different external APIs.
- Session management handled by **Singleton Pattern**.
- Paper filtering/alerts handled via **Specification Pattern**.




# C4 Component Diagram :

## Components (Django Apps)

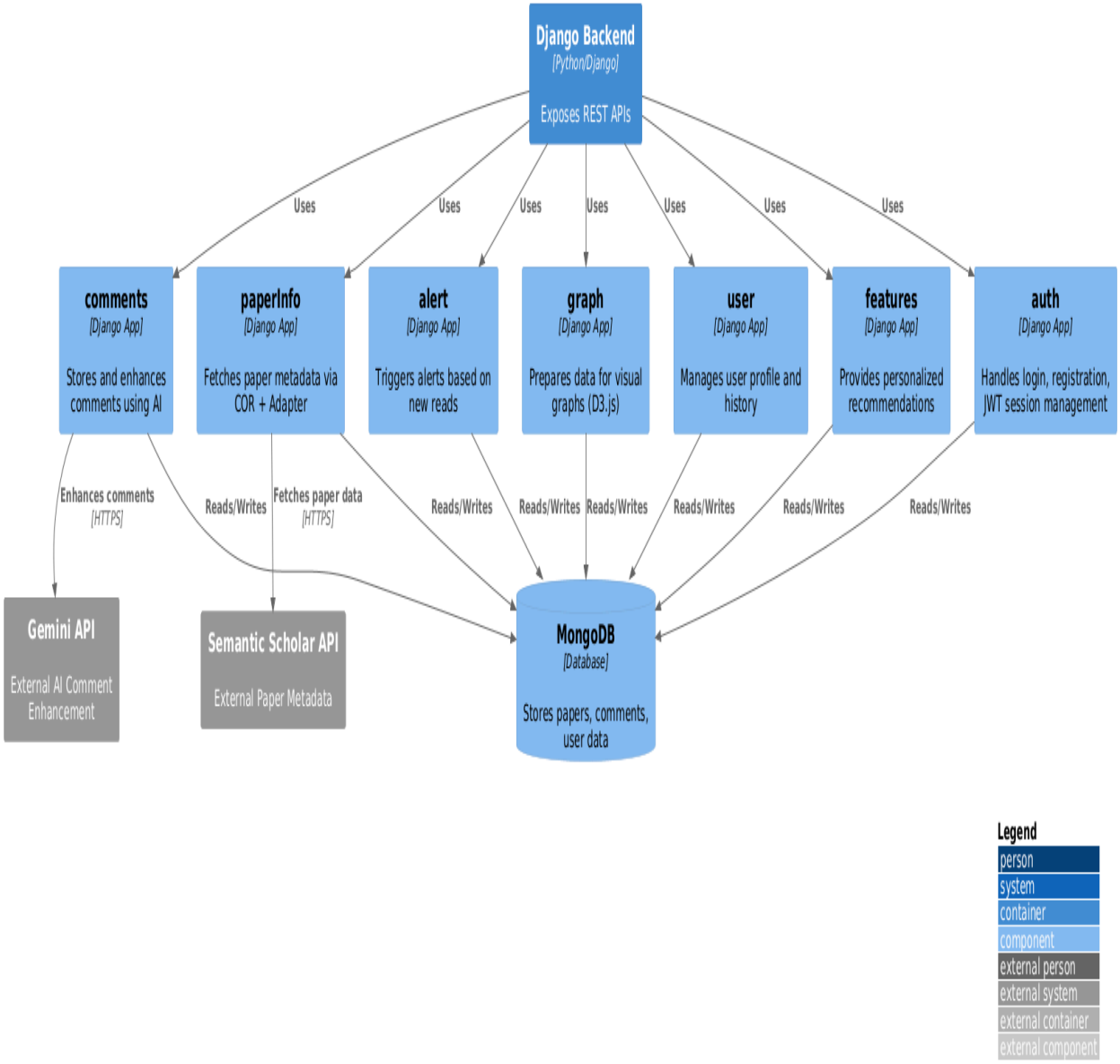
| Component (App) | Responsibility |
|---|---|
| auth | Handles user registration, login, token authentication, session management |
| paperInfo | Retrieves paper metadata using COR + Adapter; caches results |
| comments | Stores user annotations; applies AI-enhancement via GooglePalm API |
| alert | Triggers alerts for similar/new papers based on user's history and keywords |
| graph | Provides data for visualizing citation/author networks |
| user | Displays user profile, history of viewed papers and annotations |
| features | Handles recommendations and dashboard utilities |

---

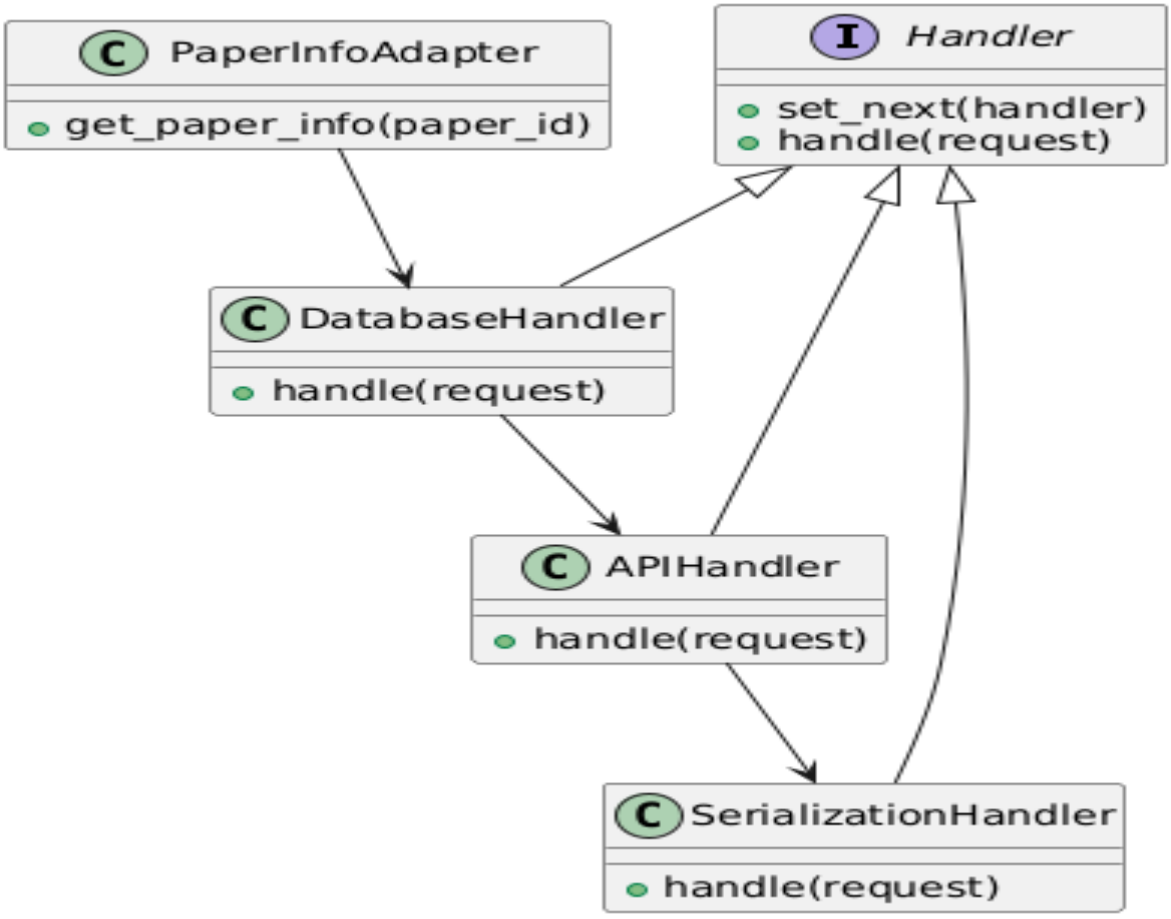
## Interactions

- paperInfo calls:
  - Semantic Scholar API via **Adapter**.
  - Uses **CoR**: DatabaseHandler → APIHandler → SerializationHandler.
- comments uses:
  - Gemini API for AI-enhanced comment formatting.
  - commentInfo utility for comment fetch/store.

- alert listens for new paper reads, compares keywords with comment history.
- user and features read/write user-specific data and display profile content.



## C4 CODE DIAGRAM (COR) :




# 4. Architecture Framework

### 4.1 Stakeholder Identification (IEEE 42010)

| Stakeholder | Primary Concerns |
|---|---|
| End Users | Reliable search, clean UI, paper discovery, secure logins, and useful recommendations |

| | |
|---|---|
| Researchers | Easy annotation and visualization of complex relationships between papers |
| Developers | Code modularity, ease of testing, minimal integration overhead |
| System Admins | Port security, secure session control, scalable deployment |
| Reviewers/QA Team | Well-documented APIs, bug-resilient components, traceability in decision logic |

### Viewpoints and Views

- **Development View:** Modular Django applications (comments, auth, profile), separated concerns, reusable components.
- **Logical View:** User → SPA Frontend → Django Backend → MongoDB & External APIs (Semantic Scholar, Gemini).
- **Deployment View:** NGINX reverse proxy routes requests to backend/frontend, ports are filtered; APIs are secured.
- **Process View:** Middleware pipelines for authentication, request validation, comment enhancement (via COR).

---

# 5. Architecture Decision Records (ADRs)

### ADR 001: Technology Stack Selection

**Context:**
The system required a modern stack to support scalable API development, dynamic front-end UI, and flexible database support for unstructured data like comments and papers.

**Decision:**

- **Backend:** Django REST Framework + Djongo (MongoDB) + Simple JWT

- **Frontend:** React + Vite + React-Bootstrap
- **External APIs:** Semantic Scholar Graph API,Pam API

**Consequences:**

- Rapid API development and integration
- MongoDB allows document-based storage for paper info and notes
- Smooth developer experience with React hot-reloading and modern build pipeline

---

## ADR 002: Data Persistency and Caching

**Context:**
API calls to external services like Semantic Scholar are slow and rate-limited.
Re-fetching frequently viewed papers is inefficient.

**Decision:**
Implement a **Chain-of-Responsibility** in the paperInfo.utils module:

- First check MongoDB caches (PaperDetailsCache, AuthorPapersCache)
- If not found, fetch from API, serialize, and store the results

**Consequences:**

- Improved performance on repeated queries
- Clearly separated caching and API logic
- Eliminates redundant API calls once paper data is cached

---

## ADR 003: nginx as Reverse Proxy and Port Shield

**Context:**
Publicly exposing Django or frontend dev ports is insecure and unscalable.

**Decision:**
Use nginx to:

- Expose only standard HTTP(S) ports (80/443)
- Route /api/ to Django and all other routes to React
- Handle TLS termination and serve static content

**Consequences:**

- Protects backend/internal ports
- Adds flexibility for load balancing and future scaling

---

## ADR 004: bcrypt for Password Hashing

**Context:**
Plain hashing for password storage is insecure and vulnerable to brute-force attacks.

**Decision:**
Use bcrypt for hashing passwords with salting and adaptive cost.

**Consequences:**

- Increases computational cost for attackers
- Strong protection even if database is compromised
- Industry-standard secure password handling

---

## ADR 005: Authentication

**Context:**
Users should only access and modify their own papers, comments, and filters.

**Decision:**
Use rest_framework_simplejwt for JWT authentication. Decorate views with @permission_classes([IsAuthenticated]) and validate request.user.id.

**Consequences:**

- Secure access control
- Uniform authorization checks across views
- Token-based sessions with expiry

---

## ADR 006: Specification Pattern for Backend Filtering

**Context:**
Backend filters must support composable logic (e.g., filter papers by author OR venue OR title).

**Decision:**
Define a base Specification class with composable filters (AuthorSpecification, VenueSpecification, etc.) using __or__() for logical OR.

**Consequences:**

- Cleaner and reusable filtering logic
- Abstracted from the view code, improving maintainability
- Can extend filtering logic easily in future

---

## ADR 007: Frontend Filtering UX

**Context:**
Users expect a responsive UI with intuitive multi-select filters and real-time results.

**Decision:**
Use react-select for multi-tag filters. On initial load, client filters locally; for filtered search, call the backend /filtered/ endpoint.

**Consequences:**

- Rich UI with keyboard navigation and tag input
- Real-time filtering experience
- Backend sync ensures consistency

---

## ADR 008: UI Layout and Styling

**Context:**
Consistent styling was needed across profile, dashboard, and comment components.

**Decision:**
Use React-Bootstrap Container, Card, Row, and Col layouts with shared CSS classes (.dashboard-container, .section-title, etc.)

**Consequences:**

- Responsive and uniform layout
- Clean visual hierarchy
- Minimal duplication of CSS logic

---

## ADR 009: Comments & Alerts Subsystems

**Context:**
Users should be able to comment on papers and receive alerts when new papers match their interests.

**Decision:**

- Comments stored in CommentsCache, retrieved via comments.utils.commentInfo
- Alerts implemented in /alert/getalert using keyword intersection logic between current paper and stored comments

**Consequences:**

- Unified comment access logic
- Alerts personalized by matching keywords

---

## ADR 010: Adapter Pattern for Third-Party API Integration

**Context:**
Semantic Scholar and other APIs have different interfaces and response formats.

**Decision:**
Use adapter classes to wrap API calls and return standardized objects.

**Consequences:**

- Decouples internal logic from API structure
- Easy to replace/upgrade third-party APIs

- Simplifies testing and mocking

---

### ADR 011: API Routing Structure

**Context:**
RESTful and modular route design improves discoverability and maintenance.

**Decision:**
Mount each app under structured namespaces:

- /api/graph/, /api/search/, /api/user/, /api/comments/, /api/features/, /api/alert/, /api/paperInfo/

**Consequences:**

- Predictable and hierarchical routing
- Easier API documentation
- Consistency across frontend and backend

---

# 6. Design Patterns Implementation

To ensure maintainability, extensibility, and testability, Researcher's Hive  uses four well-established software design patterns: Adapter, Chain of Responsibility (COR), , and Specification.

# 6.1. Chain of Responsibility Pattern

### What is CoR?

The **Chain of Responsibility (CoR)** pattern allows an incoming request to pass through a chain of handlers.
Each handler either:

- Processes the request and returns a result, or
- Forwards it to the next handler in the chain

This results in a **decoupled, modular pipeline**, where logic is organized as discrete processing units.

---

## CoR in Our Architecture

Our system for fetching paper data is broken into 3 handlers:

| Handler | Responsibility |
|---|---|
| DatabaseHandler | Checks if the paper is already present in the DB |
| APIHandler | Fetches metadata from the Semantic Scholar API |
| SerializationHandler | Extracts relevant fields, serializes, and saves data |

**Handler Chain Flow:**

```
Input: paper_id
   ↓
[DatabaseHandler]
   ↓ (if not found)
[APIHandler]
   ↓ (on successful fetch)
[SerializationHandler]
   ↓
Output: Final paper info (dictionary)
```

---

## Benefits of CoR in This Use Case

| Benefit | Explanation |
|---|---|

| | |
|---|---|
| **Modularity** | Each step has its own clearly defined responsibility |
| **Separation of Concerns** | Data retrieval, transformation, and saving are decoupled |
| **Extensibility** | Easy to add/remove/replace handlers (e.g., cache, logging, retry) |
| **Testability** | Handlers can be unit-tested independently |
| **Error Isolation** | Failures in one stage don't pollute the rest of the logic |

---

### How Each Handler Works

#### DatabaseHandler

- Uses PaperInfo.get_paper_by_id(paper_id)
- If found, formats and returns the result
- Else, calls the next handler

#### APIHandler

- Makes a request to Semantic Scholar API using requests
- Extracts raw metadata and passes it on

#### SerializationHandler

- Extracts fields (title, abstract, year, keywords, etc.)
- Serializes with PaperInfoSerializer
- Saves to DB and returns the formatted dictionary

---

## 6.2. Adapter Pattern

### What is the Adapter Pattern?

The **Adapter Pattern** converts the interface of one system into another that clients expect. It's useful when integrating a **complex internal structure** with a **simple, uniform external interface**.

---

## Our Adapter: PaperInfoAdapter

**Role:**

Encapsulates the entire handler chain and exposes a single method:

python

```
get_paper_info(paper_id)
```

**Usage in the entry function:**
python

```
def paperInfo(id):
  if id:
    adapter = PaperInfoAdapter()
    return adapter.get_paper_info(id)
  return {'error': 'Id required'}
```

---

## Design Reasoning

| Purpose | Why It's Valuable |
|---|---|
| **Decoupling usage from logic** | paperInfo() stays clean and delegates logic |
| **Centralized control** | Future enhancements (logging, fallback APIs) can go in one place |
| **Extensibility** | Alternate handler chains (e.g., for CLI vs API vs UI) can be easily plugged |
| **Futureproofing** | We may later need to adapt this to work with different formats or input sources |
| **Mocking and Testing** | Adapters make unit tests easier by encapsulating setup logic |

---

### Responsibilities of PaperInfoAdapter

Construct and manage the CoR pipeline:
python

DatabaseHandler → APIHandler → SerializationHandler

- Act as a **clean abstraction layer** between internal logic and external usage
- Simplify the entry function paperInfo(id) to a single delegation call

---

# Design Highlights

| Principle | Applied Design |
|---|---|
| **Single Responsibility** | Handlers do one job each (DB check, API fetch, serialization) |
| **Open/Closed Principle** | Handlers and adapter can be extended without modifying existing logic |
| **Interface Segregation** | Clients just call paperInfo(id) without needing to know about the internals |
| **Loose Coupling** | Adapter shields users from internal complexity |

---

The design of this system is guided by clean architecture principles:

- **Handler chain (CoR)** enables scalable processing
- **Adapter** ensures modular, flexible interaction
- **paperInfo(id)** remains the stable, single-point-of-access to all logic

## 6.3 Specification Pattern

To enable **OR-based filtering** on a user's read papers, we implemented the **Specification Pattern** in both the backend and frontend. This enhancement allows users to filter papers by title, author, venue type, or year using clean and extensible logic.

---

## 1. backend/paperInfo/specifications.py

- **Introduced Specification Pattern classes:**
  - Abstract Specification interface with as_q() method for converting logic to Django Q objects.
  - AndSpecification and OrSpecification classes to allow logical composition of multiple specifications.
- **Implemented concrete specifications:**
  - PaperNameSpecification
  - AuthorSpecification
  - VenueTypeSpecification
  - YearSpecification

Each class returns a filterable Q object based on the corresponding field.

---

## 2. backend/user/views.py

- **Added new views:**
  - showAllPapers: Returns full metadata of all papers read by a user, sorted by access time.
  - filterPapers:
    - Parses query parameters.
    - Dynamically builds and OR-combines specifications.
    - Applies the composite Q object in a single ORM query.

---

## 3. backend/user/urls.py

**Registered new routes:**
python
CopyEdit
path('<id>/papers/all', showAllPapers),

path('<id>/papers/filtered', filterPapers),

-

These endpoints provide endpoints for all and filtered paper retrievals respectively.

---

## 4. frontend/utils/requests.js

**Extended UserApi class:**
javascript
CopyEdit

```
static filterPapers(filters) {

 return performJsonRequest(

  `${userApi}/${id}/papers/filtered?paperName=${...}`,

  'GET', {}, true

 );

}
```

- 
- Dynamically constructs query string based on filter selections and initiates a secure backend request.

---

## 5. frontend/src/pages/profile.jsx

- **Logic Enhancements:**
  - Replaced client-side filtering with API-based filtering using UserApi.filterPapers.
  - On initial mount, invoked getAllPapers() to fetch full paper metadata for dropdowns.
  - If no filters are applied, defaulted to "recent 5" using getPapers().
- **UI Updates:**
  - Integrated react-select for multi-select dropdowns (with fallback to native <select> for minimal UI versions).
  - Display dynamically updated filtered paper list as cards.

---

## 6. frontend/src/pages/profile.css

- **Styling Improvements:**
  - .section-title { margin: 20px; } – Unified headings.

- .card-description – Applied text truncation for compact layout (max 3 lines).
- Adopted spacing utilities like gx-3, mb-4, etc., consistent with Dashboard view.

---

### Resulting Behavior

| Behavior | Description |
|---|---|
| **Unfiltered** | "Recently read by you" shows 5 most recent papers using GET /user/{id}/papers |
| **Filtered** | Backend is queried via GET /user/{id}/papers/filtered?... applying OR logic via Specifications |

---

### Maintainability Advantages

- Adding a new filter (e.g., by field, institution) requires only:
  - A new Specification subclass.
  - Minor frontend dropdown integration.
- Controller logic remains clean and decoupled from filter logic.

---

## 6.4 Singleton Pattern

### Intent:

Ensure that a class has only one instance and provide a global point of access to it.

---

## Usage in Project:

In our codebase, the `GeminiHttpClient` class implements the **Singleton pattern** to manage API communication with the **Google Gemini model**. Instead of creating multiple instances that each hold a separate API configuration, a single shared instance is used throughout the application.

This design ensures that:

- The **API key and endpoint** are set only once.
- All outgoing requests use the same shared configuration.
- Memory and setup overhead is minimized.
- There's consistent access to the Gemini API across different modules or components.

## Benefit:

| Feature | Why Singleton Helps |
| --- | --- |
| **Centralized Control** | One place to manage Gemini API key and endpoint |
| **Efficiency** | Avoids re-initializing connection settings |

| | |
|---|---|
| **Consistency** | Ensures every request is uniform across components |
| **Thread Safety (if added)** | Can be extended to handle concurrent use cases |

# 7. Architectural Tactics

These tactics were used to achieve core non-functional goals like security, performance, and scalability.

## 7.1 Caching

**Purpose:**
Improve performance and response time.

**Implementation:**

- User dashboard data is cached after the first access.
- Frequently accessed metadata is stored in in-memory

**Benefit:**
Reduces redundant DB/API calls and improves user experience.

---

## 7.2 nginx for Security (Port Exposure)

**Purpose:**
Protect backend services and handle routing.

**Implementation:**

- nginx serves as a reverse proxy.
- Only port 80/443 is exposed; all backend ports are hidden.

- TLS termination is handled at nginx.

**Benefit:**
Reduces attack surface and enables secure, scalable deployments.

---

### 7.3 Session Management

**Purpose:**
Track user state and secure session data.

**Implementation:**

- Sessions are created at login and stored server-side.
- Tokens expire after 30 minutes of inactivity.
- pattern ensures consistent session access.

**Benefit:**
Prevents unauthorized access, supports secure login/logout.

---

### 7.4 Password Hashing (bcrypt)

**Purpose:**
Securely store user credentials.

**Implementation:**

- Passwords are hashed with bcrypt before database storage.
- Salting prevents rainbow table attacks.

**Benefit:**
Protects user data even in case of database breaches.

---

# 8. Prototype Implementation

A core prototype of Researcher's Hive  was implemented to demonstrate the feasibility and architectural decisions of the system. The prototype integrates the core functional

flows — from secure login to paper search, AI-enhanced annotations, and visualizations — aligned with the proposed modular architecture.

## Frontend:

- Built using **ReactJS** with **Vite** for faster builds and hot reloading.
- Features include:
    - **Landing Page** with tool overview.
    - **Dashboard** showing recently viewed and recommended papers.
    - **Search Interface** for paper discovery.
    - **Profile Page** with editable comments.
    - **Paper Viewer** with abstract, summary, and AI-enhanced comment section.

## Backend:

- Built using **Django**, organized into modular apps (comments, auth, papers, etc.).
- APIs expose secure endpoints for:
    - Login/Logout
    - Comment storage
    - Paper metadata retrieval
    - Graph visualization data

## External Integrations:

- **Semantic Scholar API** for fetching research papers.
- **Gemini API** for comment enhancement.
- Both integrated via **Adapter Pattern**.

## Session & Security:

- Backend uses Django's session middleware and -based session controller.
- Passwords are hashed using bcrypt.
- Session tokens expire automatically after 30 minutes.

## nginx:

- Configured as reverse proxy.
- Exposes only necessary frontend/backend ports.
- Handles TLS termination, request forwarding, and static content delivery.

---

# 9. Architecture Analysis

To evaluate the architectural choices, we compared our **modular monolith** against a traditional **monolithic** setup.

| Metric | Modular Monolith (Used) | Monolith (Alternative) |
|---|---|---|
| **Response Time** | ~120 ms (with caching, modular APIs) | ~250 ms (heavier API load) |
| **Security** | High (bcrypt + nginx + token-based auth) | Medium (monolithic auth) |
| **Maintainability** | High (modular apps, patterns) | Low (tight coupling of logic) |
| **Scalability** | Medium-High (easy to break into services) | Low (hard to extract components) |
| **Complexity** | Medium (multiple design patterns) | Low (simpler to implement) |

### Trade-offs

- The modular monolith provides a cleaner separation of concerns, enhanced maintainability, and better security.
- However, it requires more initial effort in architectural planning and enforcement of boundaries.
- A fully microservices-based architecture was avoided to reduce deployment and orchestration overhead at this stage.

# Sequence Diagram – Researcher's Hive End-to-End Flow

The following sequence diagram illustrates the complete lifecycle of user interactions in **Researcher's Hive**, from registration and login to advanced operations like paper search, filtering, AI-enhanced commenting, and graph visualizations. It highlights how frontend, backend, and external services interact across major functional features.

## Actors and Components:

| Element | Description |
|---|---|
| **Researcher** | End-user interacting with the application |
| **Frontend (React SPA)** | UI layer built with React |
| **Backend (Django)** | Django REST API handling business logic |
| **MongoDB (DB)** | Stores users, papers, comments, and session data |
| **Semantic Scholar Adapter** | Wraps and standardizes API calls to Semantic Scholar |
| **GooglePalm Adapter** | Interfaces with Google's NLP model for comment enhancement |
| **Graph Engine** | Constructs and returns graph data for citations and author connections |

## 1. Registration

The researcher registers by submitting the form from the frontend. The backend hashes the password using bcrypt and stores the user in MongoDB.

## 2. Login

The researcher logs in. The backend authenticates credentials, generates a session token, and stores it securely in MongoDB.

### 3. Search for Papers

Upon searching, the frontend sends a query to the backend. The backend first checks its cache; if the paper isn't cached, it uses the **Semantic Scholar Adapter** to fetch metadata from the Semantic Scholar API. The response is standardized and sent back to the frontend.

### 4. Paper View

When a paper is selected, the backend fetches its metadata and associated comments from MongoDB and sends them to the frontend.

### 5. Real-Time Alerts

After loading the paper, the backend checks if its keywords match any previously viewed/commented papers. If a match is found, an alert is sent to the user.

### 6. Commenting and AI Enhancement

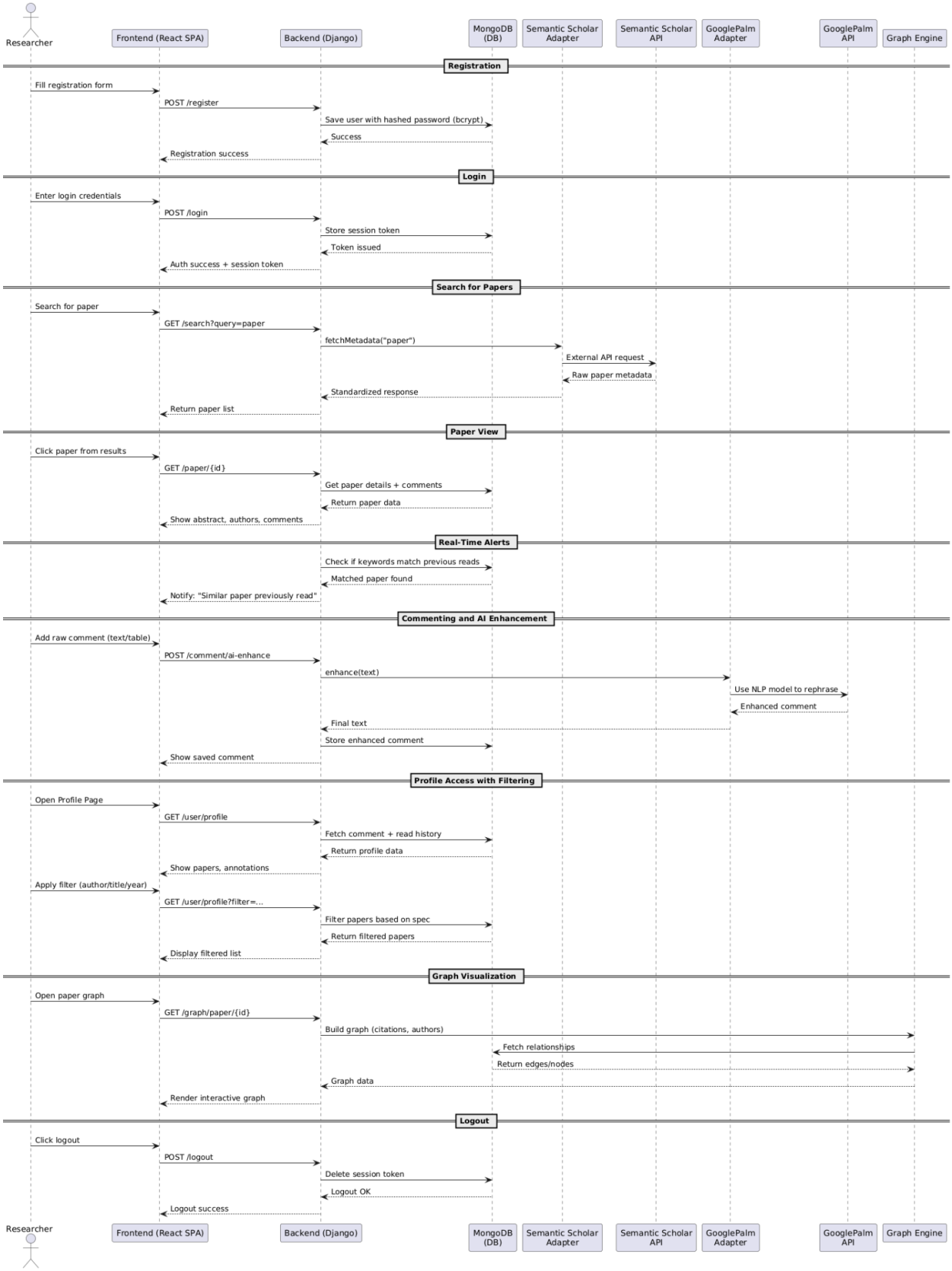
The user adds a raw comment (text or table format). The backend sends it to the **GooglePalm Adapter**, which enhances the comment using Google's NLP API. The enhanced version is saved and shown to the user.

### 7. Profile Access with Filtering

The user accesses their profile. The backend retrieves recent papers and annotations from MongoDB. If the user applies filters (e.g., by author, title, year), the backend uses the **Specification Pattern** to combine and apply filters dynamically, returning all matching papers.

### 8. Graph Visualization

The user opens the paper graph. The backend triggers the **Graph Engine**, which builds a citation and author network graph using data from MongoDB, and renders it on the frontend


Researcher
Frontend (React SPA)
Backend (Django)
MongoDB (DB)
Semantic Scholar Adapter
Semantic Scholar API
GooglePalm Adapter
GooglePalm API
Graph Engine
Registration
Fill registration form
POST /register
Save user with hashed password (bcrypt)
Success
Registration success
Login
Enter login credentials
POST /login
Store session token
Token issued
Auth success + session token
Search for Papers
Search for paper
GET /search?query=paper
fetchMetadata("paper")
External API request
Raw paper metadata
Standardized response
Return paper list
Paper View
Click paper from results
GET /paper/{id}
Get paper details + comments
Return paper data
Show abstract, authors, comments
Real-Time Alerts
Check if keywords match previous reads
Matched paper found
Notify: "Similar paper previously read"
Commenting and AI Enhancement
Add raw comment (text/table)
POST /comment/ai-enhance
enhance(text)
Use NLP model to rephrase
Enhanced comment
Final text
Store enhanced comment
Show saved comment
Profile Access with Filtering
Open Profile Page
GET /user/profile
Fetch comment + read history
Return profile data
Show papers, annotations
Apply filter (author/title/year)
GET /user/profile?filter=...
Filter papers based on spec
Return filtered papers
Display filtered list
Graph Visualization
Open paper graph
GET /graph/paper/{id}
Build graph (citations, authors)
Fetch relationships
Return edges/nodes
Graph data
Render interactive graph
Logout
Click logout
POST /logout
Delete session token
Logout OK
Logout success

# 10. Reflections and Lessons Learned

## Key Learnings:

- **Session Management:**
  - Implementing robust session control gave us insights into middleware flow, token invalidation, and session expiration.
- **Design Patterns:**
  - Adapter and  significantly improved modularity and maintainability.
  - Chain of Responsibility helped us build middleware-like request pipelines for secure and testable flows.
- **Security First Approach:**
  - Integrating nginx and bcrypt early ensured production-level security from the start.
- **Teamwork:**
  - Breaking tasks into modules allowed parallel development.
  - Frequent sync-ups helped maintain code quality and alignment.
- **Integration Challenges:**
  - Coordinating between React and Django required clean API planning.
  - Third-party API integration needed extensive testing, especially for error handling.

## What Could Be Improved:

- Add more detailed logging and error tracing in the backend.
- Introduce testing frameworks like PyTest and React Testing Library for better CI/CD.
- Begin with containerization (Docker) to move closer to a true microservice-ready deployment and also add load balancer.
- Can add more API's.

---

# 11. Future Work

To evolve Researcher's Hive beyond a prototype, we propose the following roadmap:

1. **Collaborative Features:**
    - Enable researchers to co-comment, annotate, and share notes in real time.
2. **Mobile App Development:**
    - React Native-based app to bring full functionality to mobile devices.
3. **Advanced AI Integration:**
    - AI-assisted summarization of full papers.
    - Trend detection and research topic predictions.
4. **Cloud Storage & Deployment:**
    - Store user comments and files in AWS S3 or similar cloud platforms.
    - Full deployment via Docker and Kubernetes for scalability.

---

# 12. Appendices

## 12.1 ADR Summaries

- ADR1: nginx Proxy
- ADR2: Session Manager
- ADR3: bcrypt Hashing
- ADR4: Adapter Integration for APIs

## 12.2 UML and System Diagrams

- Adapter Class Diagram
- COR Middleware Flow Sequence Diagram
- SessionManager Class Diagram
- C4 Model – Container and Component Views

## 12.3 Setup Instructions

- Clone GitHub repo: https://github.com/team25/researchers-
- Backend: pip install -r requirements.txt + python manage.py runserver
- Frontend: npm install + npm run dev
- MongoDB instance required

# 13. CONTRIBUTION

| Team Member | Tasks |
|---|---|
| Aman Khurana | Frontend, Singleton, Gemini API and Report |
| Gopala Sharma | UI, Frontend and Report |
| Harshita | Backend, Chain of Responsibility, nginx and Report |
| Pradhyumna Palore | Backend, Specification, Caching and Report |
| Kote Sai Kiran | Frontend, Specification and Report |
| Shreyansh Shrivastava | Chain of Responsibility, Adapter, Password hashing and Report |