Research Platform: System Design, Implementation, and Evaluation

Project Thesis

November 4, 2025

Contents

List of Figures

List of Tables

Abstract

This document presents a comprehensive description of an academic research platform implemented using Django. The platform consolidates user management, paper ingestion and moderation, search, social grouping, real-time discussion, and a lightweight machine learning recommendation pipeline. The thesis details concrete features as implemented in the repository, including data models, views, templates, and utilities across the accounts, papers, groups, chat, search, and ml_engine apps. We explain the system architecture, database schema, access control, request flows, template-level interactions, and the recommendation engine that records per-user recommendations. Where appropriate, we cite exact application code behavior as the basis for the description.

Keywords: Django, Research Platform, Recommendation, Real-time Chat, Groups, Papers, Moderation, Search.

Chapter 1

Introduction

1.1 Purpose and Scope

The platform enables researchers to register, authenticate, upload and manage scholarly papers, bookmark and rate content, organize into groups, and discuss papers via contextual chat rooms. A recommendation engine computes and stores user-specific recommendations. This thesis documents what is concretely implemented in the codebase, focusing on features observable in the models, views, URLs, and templates delivered in the repository.

1.2 Technology Stack

The backend is built with Django and Django REST Framework (DRF) components where used. Real-time features are prepared via Django Channels configuration, with in-memory layer enabled by default in settings. The ML component uses sentence-transformers (all-MiniLM-L6-v2) for content embeddings. Templates are standard Django templates. The default database configuration in the repository targets SQLite for development; optional MySQL configuration is present in comments and can be activated by environment variables or settings adjustments.

1.3 Structure of This Thesis

This thesis is structured to mirror the code organization: we first present the system architecture, then deep-dive into each app (accounts, papers, groups, chat, search, and ml_engine), followed by cross-cutting concerns (authentication, permissions, logging, background tasks), deployment notes, and evaluation considerations rooted in the implemented functionality.

1.4 Requirements and Constraints

Grounded in the repository's concrete behavior, the following requirements and constraints are evidenced:

- **Functional**: account registration and session login; profile management; paper CRUD with moderation; search and filtering; grouping and membership; contextual chat; generation and display of recommendations.
- **Data**: normalized relations across users, papers, categories, groups, chats, and recommendations; file storage for PDFs and avatars.
- **Access Control**: role-based branching in views using the `user_type` field, enforced for upload, moderation, admin dashboards, and group management.
- **Non-functional**: use of pagination across list views; logging to file/console; in-memory channels layer for dev; SQLite default configuration.
- **Operational**: Celery/Redis knobs present; ML model weights loaded at runtime for embeddings; templates present for end-to-end interactions.

1.5 Assumptions Evident in Code

Only minimal assumptions are implicit: authors stored as text (implying free-form or JSON-encoded list), DOI optional; recommendations computed offline or on demand; chat offensive-content filtering delegated to a utility function.

Chapter 2

System Architecture

2.1 High-Level Design

The platform follows a modular monolith architecture with distinct Django apps:

- **accounts**: Custom user model, user profiles, dashboards, and admin dashboard.
- **papers**: Core scholarly paper entities, categories, bookmarks, ratings, citations, progress, and views. Includes moderation workflow.
- **groups**: Social groups with membership roles and attachment of papers to groups.
- **chat**: Contextual chat rooms bound to papers or groups; includes a rule-based helper bot and basic offensive content filtering hook.
- **search**: Full-page search views, advanced search UI, and persisted per-user search history.
- **ml_engine**: Data models for recommendations, embeddings, and an improved hybrid recommendation engine implementation.

2.2 Runtime Components

- **Web App**: Django WSGI/ASGI application as defined in research_platform/wsgi.py and asgi.py.
- **Templates**: HTML templates render list/detail pages for accounts, papers, groups, and chat.
- **Channels Layer**: Configured to use an in-memory backend by default for development.
- **ML Inference**: The recommendation engine computes sentence embeddings and writes results to relational tables.

2.3 Persistence

The codebase ships with SQLite enabled by default in settings.py. Schema is managed by Django migrations present in each app. The models reflect normalized relational design linking users, papers, categories, groups, and conversational entities.

2.4 Component Interaction Overview

We describe typical flows grounded in the view implementations:

1. **Upload Flow**: authenticated publisher/moderator/admin opens upload form; upon submit, instance is assigned to uploaded_by; is_approved set automatically for moderator/admin; categories M2M saved; redirect to "My Papers".
2. **Moderation Flow**: moderator/admin accesses pending list; approve toggles flag and persists; reject deletes instance with message feedback.
3. **Detail Viewing**: authenticated user hits detail; if no PaperView exists, one is created and the view_count is incremented via F-expression.
4. **Bookmark/Rate**: POST toggles bookmark or creates/updates rating; feedback via messages; redirect to detail.
5. **Search**: GET parameters filtered across fields; when authenticated, query persistence to SearchHistory occurs.
6. **Groups**: creator becomes admin on creation; membership management via dedicated endpoints to join/leave/invite/remove/update roles.
7. **Chat**: room gets created lazily for paper/group; offensive messages blocked; optional bot replies injected when prefixed with @bot.
8. **Recommendations**: background engine populates UserRecommendation; dashboards and pages read from this table.

Chapter 3

Accounts App

3.1 Data Model

3.1.1 User

The custom User model extends AbstractUser with the following concrete fields and behaviors:

- user_type with choices: admin, moderator, publisher, reader (default reader)
- Unique email used as USERNAME_FIELD
- is_verified, created_at, and updated_at
- REQUIRED_FIELDS = [username] and db_table = users

3.1.2 UserProfile

One-to-one extension of User storing personal metadata: first name, last name, institution, research interests, bio, and optional avatar path (upload_to=avatars/).

3.1.3 SearchHistory

Records user queries with timestamps, ordered descending by time.

3.2 Views and Behavior

3.2.1 Authentication and Session

- **LoginView**: Email/password authentication via authenticate using email as username, success/failure messages, redirect to dashboard.
- **RegisterView**: Creates User and associated UserProfile; success redirect to login.
- **LogoutView**: Ends session and redirects home.

3.2.2 User-facing Pages

- **ProfileView**: Shows profile and aggregates counts for uploaded papers, bookmarks, ratings, and group memberships.
- **ProfileEditView**: Edit profile using UpdateView.
- **DashboardView**: Displays recent activity and recommendations sourced from ml_engine.UserRec if present.
- **AdminDashboardView**: Restricted to admin users; shows pending papers, recent submissions, and global counts.

3.3 Permissions Model

User roles (admin, moderator, publisher, reader) drive conditional logic within views (e.g., moderation, upload auto-approval) and are referenced across apps (notably in papers and admin views).

3.4 URLs, Templates, and Navigation

While specific URL patterns are defined per app, the templates under templates/accounts/ align with the views described: login.html, register.html, profile.html, profile_edit.html, dashboard.html, and admin_dashboard.html. Navigation leverages Django messages for feedback and redirects consistent with success URLs in views.

Chapter 4

Papers App

4.1 Data Model

4.1.1 Category

Unique categories with free-text description; verbose_name_plural set to “Categories”.

4.1.2 Paper

Core entity with title, abstract, authors (stored as text; JSON-encoded string by convention), publication date, optional unique DOI, optional PDF path, uploader, categories (through model), created/updated timestamps, moderation flag is_approved, counters for downloads and views, and optional summary.

Derived properties:

- average_rating: arithmetic mean over related ratings
- citation_count: count of incoming citations via cited_by

4.1.3 Associative and Activity Models

- **PaperCategory**: many-to-many bridge (unique per paper/category pair)
- **Bookmark**: per-user bookmarks with folder names (unique per user/paper)
- **Citation**: directed relationship between papers (unique per ordered pair)
- **Rating**: 1–5 integer ratings with optional review text (unique per user/paper)
- **ReadingProgress**: progress tracking per user/paper (percentage, last page, completed)
- **PaperView**: one-time per-user view registration used to increment view_count

4.2 Views and Interactions

4.2.1 Listings and Detail

- **PaperListView**: Filtering by search, category, and sorting by recency, popularity, rating, or citation counts; paginated.
- **PaperDetailView**: Access control by role; increments view count once per authenticated user via PaperView; exposes ratings, citations, and user bookmark/rating in context.

4.2.2 Creation and Moderation

- **PaperUploadView**: Restricted to publisher/moderator/admin; auto-approves when uploader is moderator/admin; saves many-to-many categories; success messaging.
- **PendingApprovalView**: Moderator/admin access to pending papers.
- **Approve/Reject**: Explicit approve action toggles is_approved; reject deletes the paper with appropriate messaging.

4.2.3 Engagement

- **Bookmark**: Toggle bookmark on detail.
- **Rate**: Create/update rating via form.
- **Download**: Serves PDF for approved papers and increments download_count with file existence checks and error handling.
- **AdminPaperListView**: Admin-only listing with search across title, abstract, authors.
- **PaperSummaryView**: Role-conditioned access to a summary template.

4.2.4 API-style Endpoints (DRF Views)

The code includes list-only implementations returning JSON for papers, bookmarks, and ratings to authenticated users (create methods return `501 Not Implemented`). These are building blocks for a REST API but are intentionally scoped to listing in the current code.

4.3 Data Integrity and Constraints

- Uniqueness constraints exist on DOI (optional) and composite keys for M2M bridges and per-user actions (bookmark, rating, paper view).
- Counters (view_count, download_count) are updated atomically using ORM updates.

- Ordering defaults prioritize recency for papers and search history.

Chapter 5

Groups App

5.1 Data Model

- **Group**: name, description, creator, created_at, and privacy flag.
- **GroupMember**: membership with roles (admin, moderator, member); unique per group/user.
- **GroupPaper**: attachment of approved papers to groups with added_by and timestamp; unique per group/paper.

5.2 Views and Interactions

- **GroupListView**: Public groups listing.
- **GroupDetailView**: Shows members and papers; contextual UI for member state and user's approved papers.
- **GroupCreateView**: Creates group and adds creator as admin.
- **GroupEditView**: Creator-only edits.
- **MyGroupsView**: Lists memberships for the requesting user.
- **Join/Leave**: Membership management with error handling for private groups and duplicate membership.
- **Add Paper**: Members can attach approved papers; duplicates prevented.
- **Invite/Remove/Update Role**: Admin/moderator capabilities to manage membership and roles; safeguards around creator removal.
- **GroupMembersView**: Detailed member roster with manage flags for template logic.

Chapter 6

Chat App

6.1 Data Model

- **ChatRoom**: Anchored to a paper or a group; created_by, created_at, active flag.
- **ChatMessage**: Linked to a room, optional user (*None* denotes bot), message text, timestamp, and bot flag.

6.2 Views and Bot Behavior

- **ChatRoomView**: For a paper; creates or retrieves room; returns last messages and supports AJAX posting. Messages are screened via an is_offensive utility before persistence. *Bot replies* are generated for messages starting with @bot, answering about abstract, authors, publication date, and categories based on the bound paper.
- **ChatDetailView**: Full room transcript view.
- **MyChatRoomsView**: Rooms created by the user or where the user has posted.
- **GroupChatRoomView**: Same workflow scoped to a group context with the same safety check.
- **AJAX Endpoint**: send_message_ajax returns structured JSON and optional bot response for paper rooms.
- **Yggdrasil Chatbot View**: A dedicated template hook for a standalone chatbot page.

6.3 Chatbot Utility

An additional class in ml_engine/chatbot.py encapsulates similar rule-based responses for reuse.

6.4 Channels and Real-time Considerations

The project configures an in-memory channels layer in settings for development. With existing consumers.py and routing.py under the chat app, the system can deliver WebSocket-backed features when deployed under ASGI. For production, a Redis channel layer is typical; the codebase includes Redis settings placeholders for task queues and can be extended for channels as needed.

Chapter 7

Search App

7.1 Views

- **SearchView**: Keyword search over title, abstract, and authors with filters for category, author, and publication year ranges. Authenticated queries are persisted into SearchHistory.
- **AdvancedSearchView**: Supplies categories for the advanced UI.
- **SearchHistoryView**: Lists user-specific search history.
- **search_suggestions**: Lightweight suggestions from titles.

Chapter 8

Machine Learning Engine

8.1 Data Model

- **UserRecommendation**: per-user recommended paper with score, reason, and timestamp (unique per user/paper).
- **RecommendationModel**: registry for model metadata and activation flag.
- **PaperEmbedding**: one-to-one per paper storing vector embeddings and model version.

8.2 Improved Hybrid Recommendation Engine

The implementation in ml_engine/recommendation_engine.py computes sentence embeddings using all-MiniLM-L6-v2, builds content embeddings for approved papers, and combines three signals: content similarity, collaborative co-preference (via ratings), and popularity (based on citation and download counts). Scores are normalized, ranked, and written to UserRecommendation. The pipeline methods provided are:

- build_embeddings() — encodes paper text to vectors and stores in PaperEmbedding.
- get_user_profile_vector(user) — averages embeddings of highly rated and bookmarked papers.
- content_based_recommend(user) — returns top-*k* papers by cosine similarity.
- collaborative_filter(user) — mines co-ratings to propose candidate papers.
- hybrid_recommend(user, top_k, ...) — fuses content, collaborative, and popularity with weights.
- save_recommendations(user, ranked) — persists results with a human-readable reason.

8.3 Integration Points

Recommendations are surfaced in the accounts dashboard and a dedicated papers recommendation view. The code reads from UserRecommendation; separate scheduling of build_embeddings and generation is implied and can be triggered via a periodic task runner (Celery scaffolding exists in the project, with Redis configuration hooks in settings).

8.4 Algorithmic and Storage Details

- **Text Representation**: For each approved paper, a concatenation of title, summary (if present), and abstract is encoded with all-MiniLM-L6-v2 into a dense vector.
- **User Profile Vector**: Averaged embedding over a user's highly rated and bookmarked papers; missing vectors yield popularity fallback.
- **Similarity**: Cosine similarity computed between user vector and paper embeddings; candidates ranked descending.
- **Collaborative Signal**: Co-rating frequency (ratings ≥ 4) across users identifies candidate papers; scores reflect count aggregation.
- **Popularity**: Weighted combination of citation counts and download counts consumed when available via properties.
- **Normalization**: Min–max normalization across combined scores to stabilize fusion.
- **Persistence**: Embeddings stored as JSON arrays and recommendations recorded per user with a human-readable reason string.

8.5 Operational Concerns for the ML Pipeline

Model initialization incurs a cold-start cost; the code is structured so embeddings can be built in batch, with subsequent generation running more cheaply. Embedding dimensions and storage size scale linearly with paper count; periodic rebuilds are recommended after content updates.

Chapter 9

Templates and User Experience

The templates under templates/ implement list/detail pages, CRUD forms, and dashboard-like summaries. Notable templates include: accounts pages (login, register, profile, dashboard, admin dashboard), paper views (list, detail, my papers, admin list, moderation queues, bookmarks, categories, summary), group views (list, detail, create, edit, members, my groups), and chat rooms (paper and group contexts). Pagination, flash messages, and form handling are present throughout.

Chapter 10

Cross-Cutting Concerns

10.1 Authentication and Authorization

Authentication uses Django sessions. Authorization leverages the custom user_type field with explicit checks in views. DRF-provided authentication classes are configured in settings for API-style endpoints, including JWT and session authentication.

10.2 Logging

Settings configure file and console logging, writing to logs/django.log with a created logs directory on startup.

10.3 Channels Configuration

An in-memory channel layer is configured for development. WebSocket consumers exist under apps/chat/consumers.py and routing under apps/chat/routing.py to enable real-time messaging when deployed with ASGI.

10.4 Storage and Media

PDFs for papers are stored via FileField under media/papers/pdfs. User avatars are stored under media/avatars.

10.5 Background Processing

Celery and Redis configuration keys are present to enable task offloading. The ml_engine/run_periodi and tasks.py can be used to schedule recommendation updates; the exact schedules are environment-dependent.

10.6 Security and Privacy

Grounded in the codebase, the following concerns are addressed or ready to be addressed:

- **Authentication**: Django session and JWT (via configuration) are enabled in settings for API endpoints; login-required decorators and mixins protect sensitive views.
- **Authorization**: Role checks are explicit in views to gate moderation, administration, and group management actions.
- **Input Handling**: Forms and ORM usage mitigate common injection vectors; file downloads are gated by approval state.
- **Chat Safety**: Offensive content is filtered via a utility before persistence; bot messages are flagged and do not impersonate users.
- **Privacy**: Search history is stored per-user and displayed only to the owner; profiles are editable by their owners.

10.7 Accessibility and Internationalization

Templates are standard Django HTML with form labels and messages. While no i18n translations are included in the repository, the structure is compatible with Django's translation framework and can be extended without invasive changes.

10.8 Scalability and Performance

- **Database**: Unique constraints and selective indexing can be added on frequently filtered fields (e.g., created_at, is_approved, foreign keys).
- **Query Efficiency**: Use of `select_related` and `prefetch_related` appears where appropriate in list/detail views to reduce query counts.
- **Caching**: While not enabled in settings, the code is compatible with per-view caching and template fragment caching for lists and detail sidebars.
- **Pagination**: All large listings are paginated to bound response sizes.
- **ML**: Embedding precomputation avoids runtime encoding; cosine similarity over stored vectors scales with vector count.

Chapter 11

Evaluation of the Implemented System

11.1 Functional Coverage

The repository demonstrates end-to-end flows: user onboarding, profile management, paper upload with moderation, search and discovery, social grouping and curation, contextual chat, and generation plus consumption of recommendations. Counters and metrics (views, downloads, ratings, citations) are explicitly modeled and used in ranking.

11.2 Non-Functional Properties

- **Usability**: Template-driven pages and consistent pagination support straightforward navigation.
- **Security**: Access controls are enforced at view level using role checks and login requirements. Offensive content filtering hooks exist for chat.
- **Performance**: Sentence embedding model is compact and well-suited for moderate datasets; embeddings are persisted to avoid recomputation on every request.
- **Extensibility**: Clear separation of concerns across apps, with DRF-compatible view patterns ready for full API exposure.

11.3 Limitations and Known Trade-offs

This documentation limits itself to what is present in the codebase. API endpoints are read-only for certain resources. Authors are stored as free text; a normalized author entity is not implemented. The chat bot is rule-based; it does not use LLMs. Channels are configured with an in-memory layer by default. Production-ready database and cache settings require environment tuning.

11.4 Testing Strategy and Quality

Grounded in the views and models, suitable tests include: model constraint tests (uniques and M2M bridges), view access tests by role, upload/approval flows, pagination and sorting, bookmark/rating toggles, download gating (approved only), search filtering and history persistence, group membership operations, chat offensive filtering and bot reply insertion, and ML pipeline smoke tests for embedding and recommendation save paths.

Chapter 12

Deployment and Configuration

12.1 Settings Overview

- **Installed Apps**: includes rest_framework, rest_framework_simplejwt, corsheaders, channels, and django_filters.
- **Database**: SQLite by default, with commented MySQL configuration and environment variable hooks.
- **Media/Static**: Media under media/; static served from /static/ with STATIC_ROOT configured.
- **CORS**: Local development origins pre-configured.
- **Logging**: File and console handlers with INFO level.

12.2 Operational Notes

To surface recommendations, ensure the recommendation engine is executed periodically to refresh PaperEmbedding and UserRecommendation. Channels require an ASGI deployment (e.g., Daphne or Uvicorn) for WebSocket functionality at scale.

12.3 CI/CD, Backup, and Monitoring

While not included in the repository, the configuration lends itself to:

- **CI**: linting and unit tests on push; migrations check; static analysis.
- **CD**: environment-variable driven settings enabling database and cache backends.
- **Backup/DR**: periodic database dumps and media storage snapshots; embedding tables can be regenerated but still warrant backup.
- **Monitoring**: log shipping of logs/django.log; request metrics and background job durations.

Chapter 13

Conclusion

The implemented platform provides a cohesive environment for managing scholarly content and researcher interactions, complete with moderation, engagement metrics, social organization, contextual chat, and a content-aware recommendation subsystem. The modular code structure and the explicit, typed models position the system well for iterative extension, including fuller REST APIs, more advanced ranking, and production-grade deployments.

Appendix A

Primary Data Models

Model	Key Fields and Relations
User	email (unique), username, user_type, timestamps
UserProfile	one-to-one User; personal info and avatar
SearchHistory	user, query, timestamp
Category	name (unique), description
Paper	title, abstract, authors, publication_date, DOI (unique, optional), pdf_path, uploaded_by, categories M2M (through), counters, summary, moderation
PaperCategory	paper, category (unique together)
Bookmark	user, paper, folder (unique together user/paper)
Citation	citing_paper, cited_paper (unique together)
Rating	user, paper, rating 1–5, review_text (unique together user/paper)
ReadingProgress	user, paper, progress_percentage, last_page, completed
PaperView	user, paper (unique together), viewed_at
Group	name, description, created_by, created_at, is_private
GroupMember	group, user, role (unique together)
GroupPaper	group, paper, added_by, added_at (unique together)
ChatRoom	paper or group, created_by, created_at, is_active
ChatMessage	room, optional user (None=bot), message, timestamp, is_bot_message
UserRecommendation	user, paper, score, reason, created_at (unique together user/paper)
RecommendationModel	name, version, model_path, is_active, created_at
PaperEmbedding	one-to-one paper, embedding JSON array, model_version, created_at

Appendix B

Key Views and Flows

Appendix C

Data Dictionary

This appendix tabulates key fields based on the concrete model definitions.

Model	Field	Type	Notes
User	email	Email	unique, USERNAME_FIELD
User	user_type	Char(20)	choices: admin/moderator/publisher/reader
UserProfile	avatar	Image	uploads to avatars/
Paper	doi	Char(100)	optional, unique
Paper	pdf_path	File	uploads to papers/pdfs/
Paper	is_approved	Boolean	moderation flag
Rating	rating	Integer	1..5
ReadingProgress	progress_percentage	Float	0..100
ChatMessage	is_bot_message	Boolean	distinguishes bot entries
PaperEmbedding	embedding	JSON	numeric array
UserRecommendation	score	Float	normalized hybrid score

Appendix D

Permissions Matrix

Action	Reader	Publisher	Moderator	Admin
View approved papers	Yes	Yes	Yes	Yes
Upload paper	No	Yes	Yes	Yes
Auto-approve own uploads	No	No	Yes	Yes
Moderation queue	No	No	Yes	Yes
Admin paper list	No	No	No	Yes
Create group	Yes	Yes	Yes	Yes
Invite/remove/update roles in group	No	No	Yes (if group role)	Yes (if group role)
Access dashboard	Yes	Yes	Yes	Yes
Access admin dashboard	No	No	No	Yes

- **Paper Upload**: form assigns uploaded_by; auto-approves for moderator/admin; saves categories; success message; redirect.
- **Paper Detail**: role-conditional queryset; view count increment via PaperView; related ratings and citations provided in context.
- **Group Management**: create (creator becomes admin), join/leave, invite/remove/update role with permissions.
- **Chat**: block offensive messages; bot answers metadata queries; AJAX endpoint returns structured JSON.
- **Search**: persists search queries for authenticated users; filters by category/author/year.
- **Recommendations**: engine builds embeddings and writes to UserRecommendation; dashboard reads and displays them.