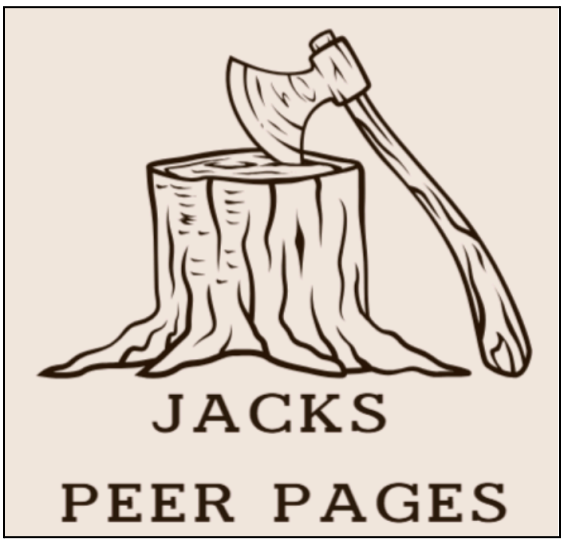
Software Design Document

Version 2.0 [FINAL]



Jack's Peer Pages

Feb. 23rd, 2026

Sponsor: Dr. Keith Nowicki
Team Mentor: Md Nazmul Hossain
Faculty Mentor: Dr. Ana Chaves

Team: Haley Berger (Team lead), Haley Kloss (member), Jeremiah Lopez (member), Tyler Austin (member)

Table of Contents

Table of Contents 2
Introduction 3
Implementation Overview 4
Architectural Overview 6
Component-Level Design 10
Implementation Plan 16
Conclusion 19

Introduction

Scientific research depends on the ability to share knowledge and evaluate its quality. While peer review plays a critical role in this process, the systems that support it have not kept pace with the scale and needs of modern academia. Peer reviews are often difficult to find, scattered across journals, or hidden behind paywalls, which limits their usefulness to researchers and students. As the volume of published research continues to grow, and as AI-assisted writing becomes more common, issues of transparency, credibility, and trust in academic work have become increasingly important.

This project, the Continuous Community Review Compendium, aims to improve access to and understanding of post-publication peer reviews by providing a centralized, web-based platform for articles and their associated reviews. The system allows verified users to upload article metadata, submit peer reviews, and search for existing content in an organized and transparent way. A key focus of the platform is education: professors are able to sponsor and oversee student accounts, review submitted peer reviews, and provide guidance before reviews are published. This approach allows students to learn how to write peer reviews in a structured, low-risk environment while maintaining academic standards.

From a technical perspective, the system must support several core requirements that guide its design. These include fast and reliable searching of articles and peer reviews, clearly defined user roles (such as students and advisors), and workflows that support review approval and moderation. The platform is implemented as a web application using Django and a MySQL database, and is deployed on managed cloud infrastructure. Performance, scalability, and ease of maintenance are important constraints, as the system is intended to grow over time and be maintained by future developers.

This document describes the architectural design and component-level structure of the system. It is intended to provide a clear understanding of how the system is organized, how major components interact, and why key design decisions were made.

Implementation Overview

The section of the document details what is being implemented, as well as rehashing project constraints mentioned in prior documents.

The Continuous Community Review Compendium is being implemented as a web-based platform designed to support post-publication peer review in an accessible, transparent, and educational way. The system provides a centralized location for article metadata and peer reviews, allowing users to search for articles, view and submit reviews, and participate in a moderated academic discussion. A central goal of the implementation is to support new researchers/students in learning how to properly write a peer review while maintaining clear ownership of content and review quality.

The system is implemented as a traditional website (specifically, a web app), using a server-side framework and relational database. This approach was chosen to prioritize reliability, maintainability, and clarity over experimental or highly distributed designs. Given the academic context of the project, the expected scale is moderate during initial deployment, with growth over time as more students and faculty adopt the platform. As a result, the design emphasizes clean separation of concerns, ease of extension, and the ability for future developers to understand and modify the system without requiring deep domain knowledge.

The backend is built using the Django web framework, which provides a structured Model-View-Template (MVT) architecture and strong, built-in integration with relational databases. Django is responsible for handling business logic, enforcing role-based permissions, managing peer-review workflows, and coordinating interactions between the database and user interface. A MySQL database is used to store persistent data such as user accounts, article metadata, peer reviews, and advisor-student relationships. This database-driven design supports structured querying, data integrity, and future scalability.

The frontend is implemented using Django's templating system, with standard web technologies (HTML and CSS) to render pages for article searching, viewing, submitting, and account management. This server-rendered approach reduces architectural overhead while maintaining responsiveness and ease of development. Communication between the frontend and backend occurs through synchronous HTTP requests, with Django handling request routing and response generation.

The system is deployed using DigitalOcean's App Platform, with the MySQL database hosted separately using DigitalOcean Managed Databases. The application is connected directly to the project's GitHub repository, allowing automated deployment whenever changes are pushed to the designated deployment branch. Hosting the application and database as separate managed services allows each to be maintained independently while remaining tightly integrated, supporting system stability and ease of recovery.

It should also be noted that design decisions were made from project constraints. For example, although we agree that MySQL serves this project well, our client did make the initial request for MySQL to be used. Additionally, financially, the cost of hosting and use of all associated features needs to be manageable. We don’t have an unlimited, or even a large, budget, so we needed to find a hosting and database service that someone with little to no development experience can manage on their own (after the completion of the project) while still having the power we need. Because this project is expected to be used long-term with hopeful plans for large expansion, we also needed to set the project up in a way such that new developers can easily jump on. The website needs to be easily maintained. So, when considering the framework, we had to consider maintainability and scalability. Django was selected in part for these reasons, as it is widely used, well-documented, and supports scalable development practices.

Overall, this implementation approach provides a clear and maintainable foundation for both initial development and continued evolution of the platform.

Architectural Overview

This section of the document provides a high-level architectural overview as well as written details regarding primary components and how they communicate, as well as deployment details and a summary of non-trivial decisions.

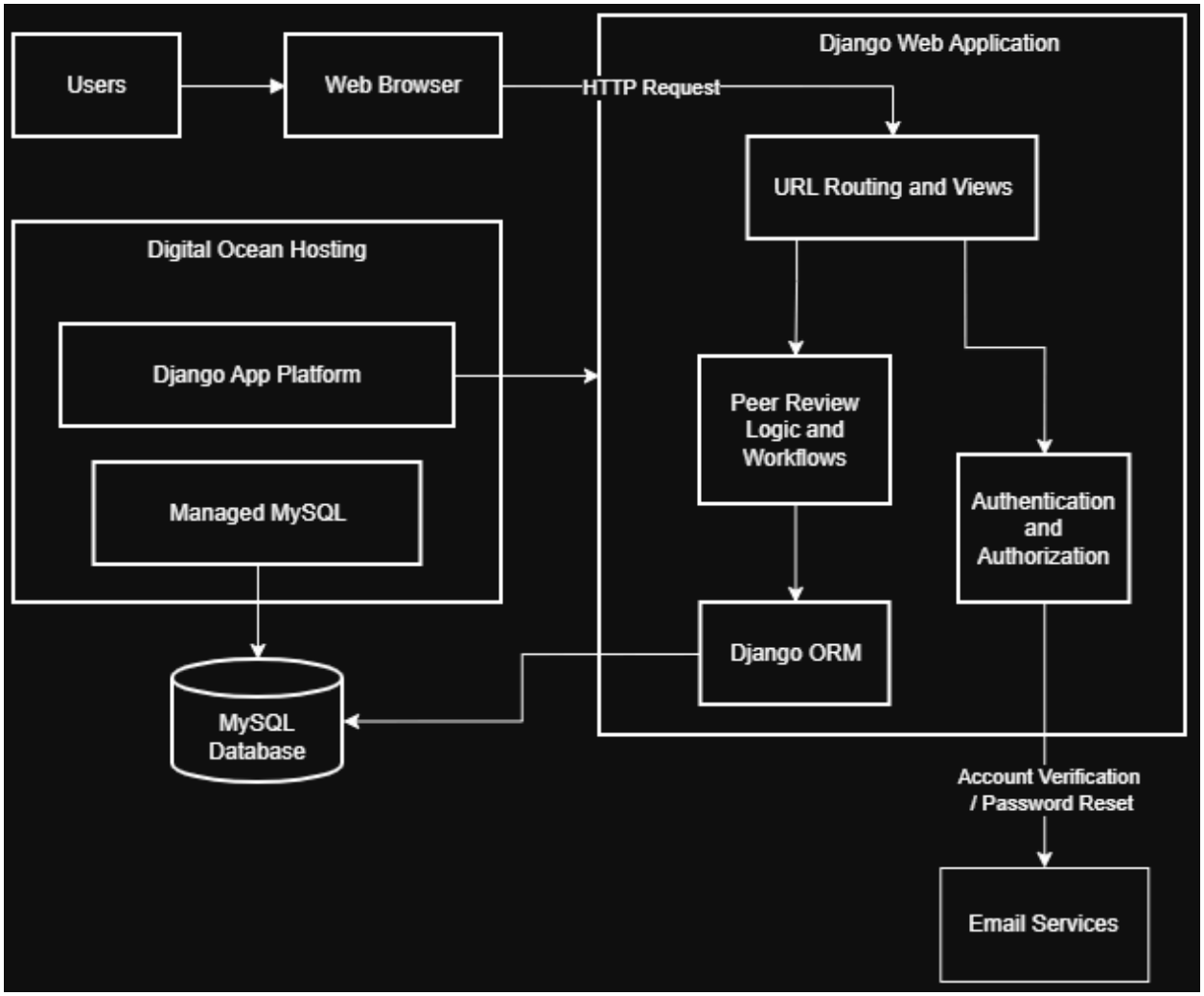


Diagram 1: Architectural diagram displaying the overall system architecture.

Overall Architecture

The system is implemented as a client-server web application built using the Django web framework and deployed on the DigitalOcean App Platform. At a high level, the architecture follows a monolithic Django application design with clearly separated internal components responsible for request handling, feature logic, authentication, persistence, and external service integration. This structure is fairly simple, but modular enough to be easily maintained and support future expansion. This architecture is depicted in diagram 1.

Overall, the system has a layered monolithic architecture rather than a microservice-based one. All application logic is contained within a single Django project, with functionality being divided across multiple Django apps. This design was chosen over a distributed services architecture due to the project's scope and other constraints mentioned prior (such as future development, simplicity, and consistency). A monolithic approach reduces complexity and simplifies development, allowing us to focus on getting our project to the client with all of the core features. But, being able to split parts of the project into separate apps has allowed us to still have a degree of separation for independent feature development and limit the cross-feature impact of bugs.

Our web application operates with a client-server boundary, where users interact with the system exclusively through a browser. All logic, authentication, and other functions are handled server-side within the application.

Primary Components

Web Browser / Client	Users access the system via a standard web browser, which serves as a client. The browser is responsible only for rendering HTML pages, collecting user input, and issuing HTTP requests to the server. No software logic is executed on the client, which helps us limit security concerns and centralizes validation enforcement on the server.
URL Routing (views.py)	The URL routing and Django views serve as the entry point to the Django application. Incoming HTTP requests are synchronously received and mapped to view functions or class-based views using Django's URL dispatcher. This component is responsible for request validation, access control enforcement (via authentication and authorization checks), and delegation to the appropriate logic. Views coordinate interactions between the authentication system, software workflows, and data persistence layer rather than just implementing domain logic right away. They then construct HTTP responses. We do want to quickly note that each app has its own views.py file where this happens.
Logic and Workflows	Logic and workflow is where the system's primary behavior is contained. It implements all the rules and logic for the site's components, including article submission, peer review creation, Advisor/Student account connections, and other core features. With this being separate from the view layer, we are able to enforce a degree of separation that prevents excess rules and logic getting mixed up at various endpoints. This then interacts with Django ORM for and may trigger things such as email notifications (from views and routing).
Authentication	User authentication is handled using Django's built-in authentication system along with custom account logic and the addition of Google's

	re-CAPTCHA. This manages user identity, sessions, role-based privileges, and et cetera. By centralizing authentication, we can ensure consistent behavior and security.
Django ORM	Django's ORM (Object-Relational Mapper) gives us a better abstraction over a database, in this case, the managed MySQL database. All data access is performed through the ORM models for data integrity. Using ORM rather than straight SQL queries makes the whole project more modular and, if the decision to swap databases to something besides MySQL comes along, easily portable.
Email Services	Email services, for this project, are treated as an external dependency. It is used for account verification and password resets. The Django application communicates with the email service through a third-party provider (a Google developer feature) using DJango's email utilities. The service is managed through Google and accessible through the jackspeerpages@gmail.com project email. This is a synchronous interaction from the perspective of the application, with the independent operation of the external email service. Because the service is isolated, it is easy to swap to a different email service provider, or, if desired, even create a new one.

Component Communication

All primary communication within the system is synchronous and request-based. User actions (in their browser, on the website) create HTTP requests that go through URL routing and the view layer, into the corresponding appropriate logic components (that being authentication based, or other logics). Depending on the flow, an email may then be sent out through the external provider or will continue to the ORM for database interactions. New user requests will then start again with the web browser and their interactions.

In the case of an email being sent, the user is provided (in the email) with a new link by which they can continue the process they were going through. In other words, if a user requests a password reset, then they would send an HTTP request, enter our URL routing and views logic, be passed along to authentication and authorization, and then be sent an email. The email would contain a link, allowing the user to complete their wanted task (resetting their password) with the appropriate permissions by taking them back to the website with the new URL (creating another HTTP request) where the routing would now, instead, take them to our other logic and workflows. The ORM is then used, and the new password is accounted for.

Deployment and Infrastructure

As mentioned prior, the website is deployed through the DigitalOcean App Platform, which provides managed runtime environments for the Django application and an option of a managed MySQL database, which we use. As a reminder, this was used for ease of development and for the sake of our client to be able to easily maintain the project after capstone's completion.

Hosting both the application and database on the same platform helps with latency, but also simplifies the management process.

Non-Trivial Design Choice Rational

Several non-trivial architectural decisions were behind the design mentioned above:

1. A monolithic-style Django application was chosen over microservices to reduce project complexity and better fit with our project constraints and scale. We still modularized features to make later development easy and to better allow future developers to swap from a monolithic-style later on, if they so please.
2. Our server-side logic choices were made to help keep the project centralized and maintain security and future maintainability.
3. The use of Django's ORM was picked over straight SQL queries to help with bug catching, but also to make it easy for future developers to swap to a different database if they want.
4. The synchronous communication helps to simplify project flow and debugging while keeping performance up. It was also easy to implement.
5. Our decision to use a managed hosting and database service was made with future expansion and maintenance in mind. Although we as a team have the skills to set everything up ourselves, it would be very difficult for our client to maintain once we are no longer working on the project.

Component-Level Design

This section of the document details the internal structures and responsibilities of major components identified in the system architecture. Each component is related to a specific area of system behavior, has a public interface, and has defined boundaries.

URL Routing (views.py)

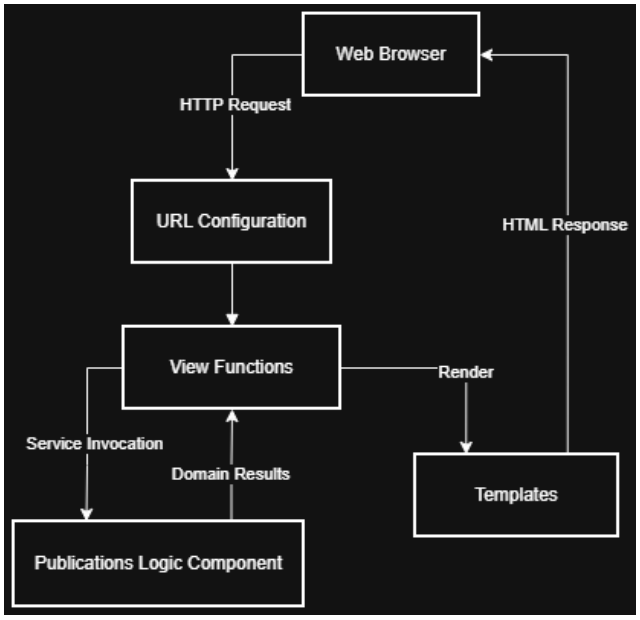


Diagram 2: Diagram depicting the internal routing system. This is how the website decides what to do/display to the user.

Responsibility: URL routing is responsible for handling all incoming HTTP requests and returning appropriate responses. This is depicted in diagram 2. This is the system's main point of entry and handles all coordination between the client, authentication system, and system logic. It essentially handles the delegation of behaviors between logic and authentication.

Boundary: Only owns request handling, validation, and response decisions (what part of the website the user should end up on). It does not handle any other logic (feature logic) and is essentially just the delegator.

Internal Structure

APP (ex: publications)

- urls.py: Defines URL patterns and maps them to views
- views.py: Implements function-based or class-based views

Interface

- **Service:** Routes user to the correct part of the website
- **Inputs:** HTTP requests, URL parameters, data from forms
- **Outputs:** HTTP responses (HTML pages, redirects, error responses)
- **Interactions:**
  - Calls logic services (peer review, posting articles, etc)
  - Invokes authentication checks

Logic and Workflows

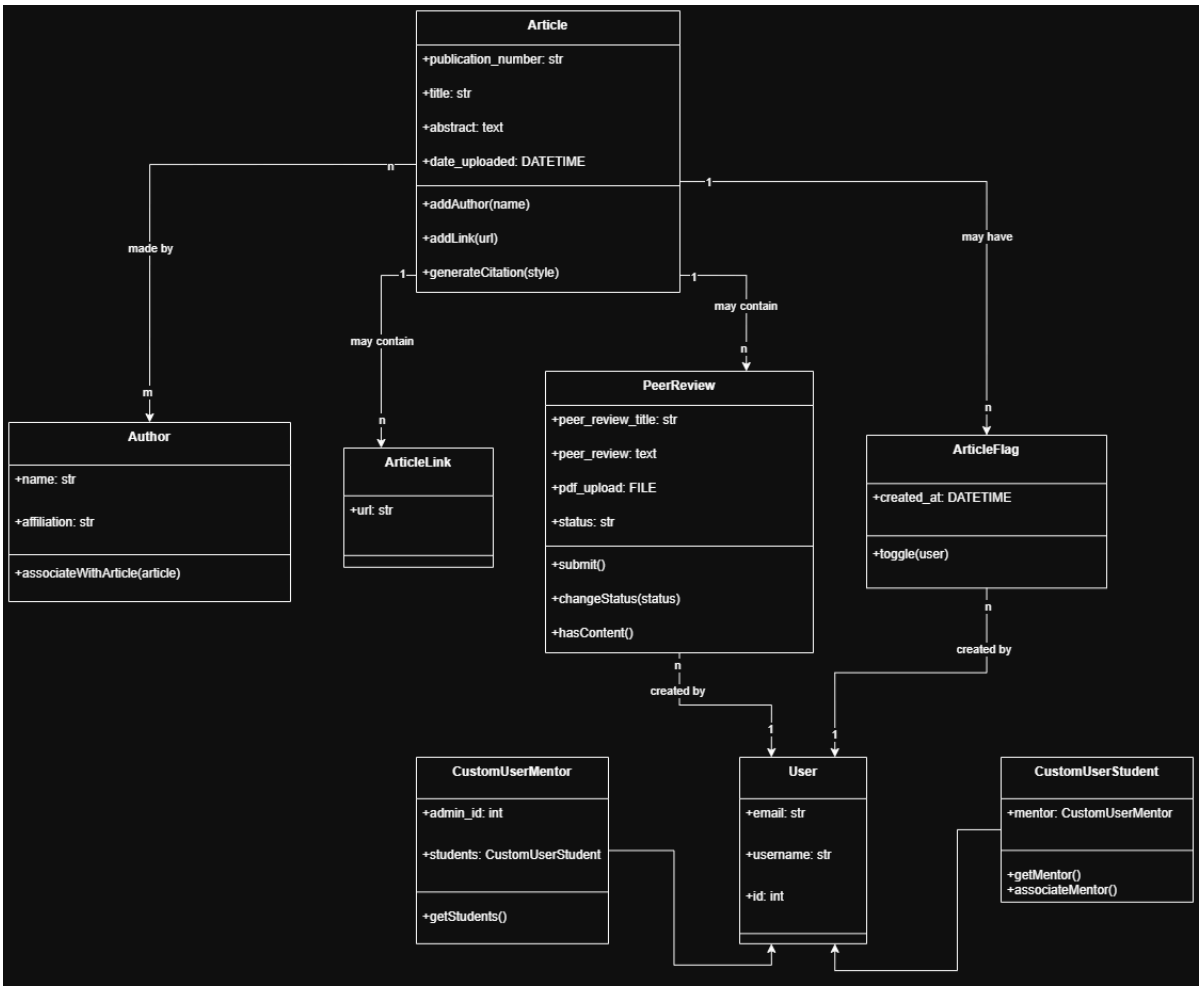


Diagram 3: This is a class diagram depicting the general system structure.

Responsibility: This component is responsible for all primary system behavior (see: diagram 3). It manages everything from article submission to peer review creation. The rules are managed here, making sure they don't get mixed up between components.

Boundary: Does not handle HTTP requests or authentication; if the system is in this component, then it is just acting out the logic and workflows for the features themselves. Delegation between this and authentication has already passed.

Internal Structure	**Interface**
Internally, this block is composed of many sub components. It is split up	As each domain contains its own service and interactions, they will be similarly split up to demonstrate all. ● Article

into 4 primary domains such that:

- Article
  - Responsible for article metadata, authorship, and references
- Peer Review
  - Responsible for creation and lifecycle of peer reviews
- Flagging
  - Responsible for AI-generated article flag tracking
- Citations
  - Determines citation formatting and allows users to get article citations

  - **Service:** Create new articles, update existing articles, associate authors, attach external reference links
  - **Inputs:** Publication metadata, author names, reference urls, user identity
  - **Outputs:** Persisted article, possible validation errors
  - **Interactions:** Reads and writes Article, Author, and ArticleLink
- Peer Review
  - **Service:** Create peer reviews, assign review status, attach abstract
  - **Inputs:** Title, text, status, user identify
  - **Outputs:** Persisted peer review, possible validation error
  - **Interactions:** Writes to PeerReview
- Flagging
  - **Service:** Add and remove AI flag, query for AI
  - **Inputs:** Article reference, user reference
  - **Outputs:** Flag creation or removal confirmation
  - **Interactions:** Reads or writes to ArticleFlag and enforces one flag per user.
- Citations
  - **Service:** Generates a citation string
  - **Inputs:** Article reference and citation style (like APA, MLA, ETC).
  - **Outputs:** Generated citation text
  - **Interactions:** Reads Article and associated Author

Authentication

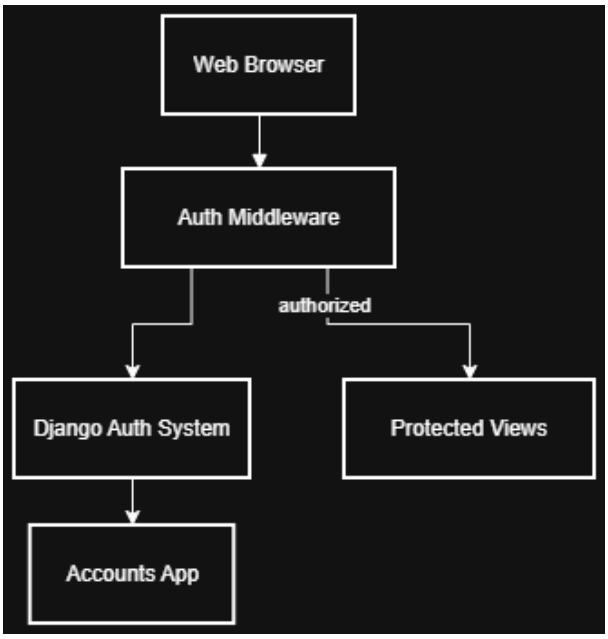


Diagram 4: This diagram depicts how the system runs through authentication.

Responsibility: Authentication manages the user's identity and their ability to change their password/make an account. It used both built-in Django security frameworks and Google's re-CAPTCHA for security checks. It works with the email services component for external email validation for the sake of user security. Please see diagram 4.

Boundary: Does not have anything to do with application-specific workflows. This only focused on verifying a user or letting someone make an account.

Internal Structure

The authentication structure (in terms of project software) is contained in the Allauth Manager app and uses the forms:

- Login
- Signup
- ReCaptcha
- CustomSignupForm

Interface

- **Service:** Login, logout, registration, email verification, and password reset
- **Inputs:** Credentials, tokens, and authentication request
- **Outputs:** Authenticated sessions, authorization decisions
- **Interactions:**
  - Invoked by views
  - Sends an email service trigger for verification

Django ORM

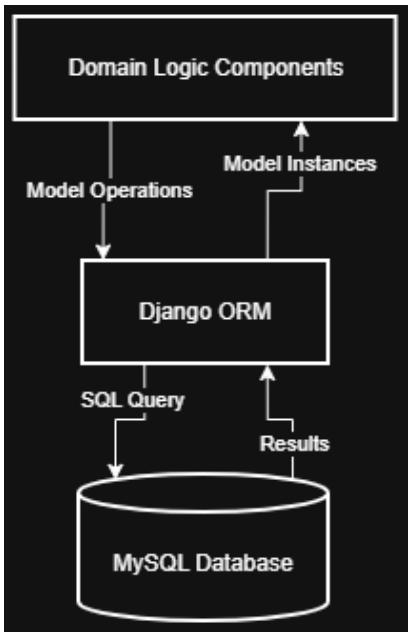


Diagram 5: This diagram depicts the general interaction between the system logic, ORM, and database.

Responsibility: The ORM helps maintain data access integrity. It communicates with our database and makes sure data access is reliable and safe. Please see diagram 5 for a visual representation of the interaction.

Boundary: Logic is not handled here. This is just a “mediator” between the database and requests from the user/system.

Internal Structure

- All model.py definitions in the apps.
- Generated ORM queries from the model.py relationships.

Interface

- **Service:** Query operations based on model managers, interactions with the database
- **Inputs:** Model instances and query parameters
- **Outputs:** Query results
- **Interactions:**
  - Invoked for a database interaction request

Email Services

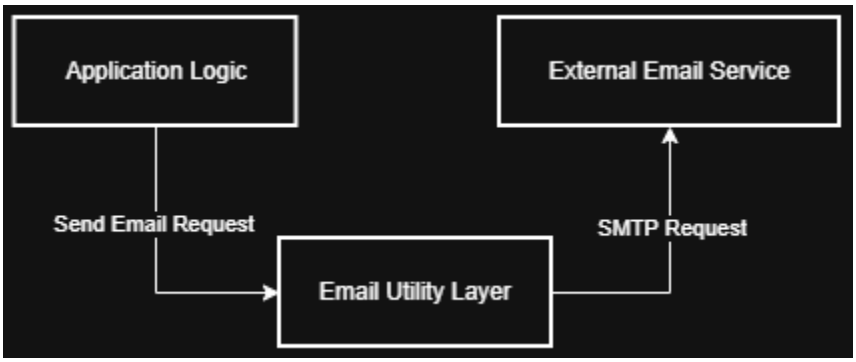


Diagram 6: This diagram depicts the system interaction with an external email service.

Responsibility: Email services is a component that has external dependency. It is a service we integrate. It is responsible for sending emails to validate account creating, and allows for users to reset their passwords. For a visual on how this interacts with the system itself, please see diagram 6.

Boundary: This is only used when an email is needed. It is external to the Django project (and its internal apps).

Internal Structure

- Django email utils configured with an external email provider through Google, linked to the jackspeerpages@gmail.com account.

Interface

- **Service:** Sends email for password reset or account creation.
- **Inputs:** Recipient address, subject, message context
- **Outputs:** Successful email delivery or failure message
- **Interactions:**
  - Called by Authentication and Accounts logic

Implementation Plan

This implementation plan describes how the system architecture is realized across the remaining development cycles. It is split up into three phases that correlate with our demos. These phases are Alpha Demo I, followed by Alpha Demo II, and then the Acceptance Demo. At this stage of the project (from when this is being written), the majority of core features defined for Alpha Demo I have been implemented (that being, most of the total MVP features for this project). The remaining work focuses on the final addition of features, usability refinement, adjustments based on user testing, and preparation for project demonstrations.

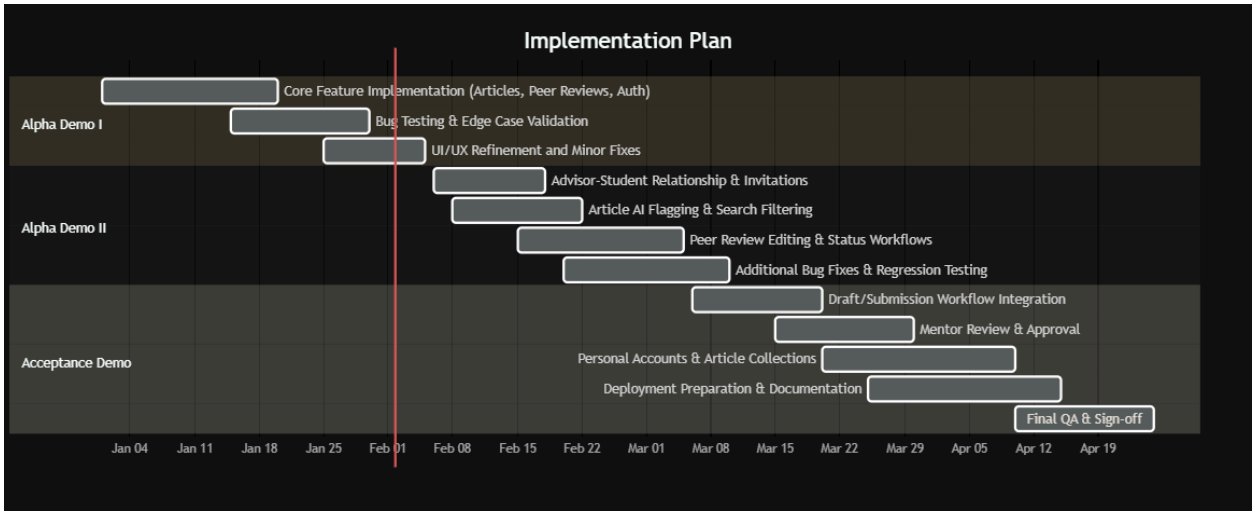


Diagram 7: A diagram displaying the timeline of our implementation plan.

Phases and Milestones

Phase 1: Alpha Demo I Status: Done

This phase focuses on developing the MVP features and improving implementation from last semester. It also includes getting the project hosted on the live production site, making it available for users.

Features to be added, tweaked, and bug-tested during this phase include:

- ☐ Account creation and verification
- ☐ Authentication and password reset
- ☐ Article upload, editing, viewing, and searching
- ☐ Peer review creation and viewing
- ☐ Citation generation
- ☐ Home page navigation and routing

Primary activities during this phase include:

- ☐ Bug identification and resolution
- ☐ Validation of acceptance criteria for all Alpha Demo I use cases
- ☐ Edge-case handling (invalid input, missing data, permissions)
- ☐ Deployment configuration verification

Phase 2: Alpha Demo II Status: In Progress

Once Alpha Demo I functionality is stabilized, development proceeds to Alpha Demo II features, which build directly on existing components:

- ☐ Advisor-student account linkage via keys
- ☐ Article AI flagging
- ☐ Search filtering using article flags
- ☐ Flag assignment during article upload
- ☐ Peer review editing by authors and mentors

These features are implemented incrementally, reusing the existing publications, accounts, and authentication components. Development is meant to allow for feature ownership, by which each member of the team will get development tasks.

Additional development will be done in the areas of:

- ☐ Account relationship features
- ☐ Publication metadata and flagging logic

Phase 3: Acceptance Demo Status: Planned

The final phase will involve adjusting things and fixing any bugs that the user testers have found, while also potentially adding in last minute stretch goal features, including:

- ☐ Draft and submission states for peer reviews
- ☐ Mentor review and approval workflows
- ☐ Personal account information pages
- ☐ Personal article collections (saved bibliography)

Integration Strategy

The features will be implemented according to the above mentioned phases, with intermittent testing and validation sprinkled through each phase. Features will be added to the production site intermittently, depending on the feature. Smaller features will be added on completion, while larger features (such as account connectivity between student and mentor) will be added at the end of each development cycle during a planned website downtime to prevent our client and users from being affected.

Team members will all be assigned coding tasks. Once completed, the expectation is that the new code be fully tested locally before a pull request into main is added. At which point, code review will be done by another team member before the code is officially pushed into main. The code will only be added to the deployment branch for production, depending on the feature

addition timeline mentioned above. For the sake of clean integration, everyone is also expected to stay up to date with main at any given time.

Technical Risks, Trade-offs, and Mitigation Strategies

Risk: Undiscovered Defects/Bugs in Implemented Features

- Impact: Users may encounter problems that impede their ability to properly use the website.
- Mitigation: Focused bug testing and validation, testing everything as it is implemented. Also, respond quickly to bug reports and issues.

Risk: Increasing Complexity

- Impact: As the project progresses and more features are added, the complexity of the overall project increases greatly. This could lead to features unintentionally impacting one another and causing bugs.
- Mitigation: Test everything! Make sure to test every new feature and to modularize code as much as possible.

Risk: Email Service Dependency

- Impact: Delayed verification or password reset failures.
- Mitigation: Unfortunately, there is nothing we can do about this, as it is an external process. We can warn users and make them aware that there could be a delay in receiving account validation or password reset emails.

Overall, this implementation plan reflects the current state of the system and provides a realistic path from Alpha Demo I through final delivery. By prioritizing testing and incremental feature addition, we have confidence in the completion of the project.

Conclusion

Having a platform on which researchers and students can share their thoughts and opinions on scientific articles is essential to the spread of accurate information in scientific communities. The continuous community review compendium fulfills the peer review gap in the scientific publication system, allowing students to learn how to peer review, professionals to share their peer reviews, and readers to flag AI usage. All of these functions ensure that people are reading high-quality articles or have a way to spread the word about inaccurate information.

Our platform is designed with these goals in mind, with our key design elements being Django architecture, our server-side framework, and managed hosting. The benefits of using Django architecture are its consistency, reliability, and ease of understanding for future programmers. It also employs server-side logic that emphasizes security, so that user accounts retain their integrity and are not susceptible to AI. Using managed hosting allows for easy maintenance of the website and for future expansion and scaling. These are all vital to ensuring that our website works as intended, not only now but in the future, when it has more users and different programmers working on it. All of our design choices center on the goal of having a well-functioning, future-proof website for researchers to use for many years to come.