Design and Development of OpenGalaxy: A Full-Stack Web Application for Browsing Open-Licence Media

UNIVERSITY OF
LINCOLN

Alfie Atkinson
25715017

25715017@students.lincoln.ac.uk

School of Engineering and Physical Sciences

College of Science

University of Lincoln

Submitted in partial fulfilment of the requirements for the
Degree of Master of Science in Computer Science

May 2025

Abstract

This report documents the development of OpenGalaxy, a full-stack web application designed to provide a unified and intuitive interface for discovering and managing open-licence media. The platform was built in response to the fragmented user experience found across existing repositories such as Openverse. Leveraging a modern tech stack — including a React/Next.js frontend and a Django REST Framework backend — OpenGalaxy adopts an API-first, containerised architecture with emphasis on performance, SEO, and developer productivity. The frontend incorporates hybrid rendering, while the backend integrates PostgreSQL and Redis for persistence and caching, respectively. Continuous Integration and Deployment (CI/CD) workflows were implemented using GitHub Actions, Vercel, and Heroku, with pre-commit quality checks and end-to-end testing via Cypress. While the project follows a solo waterfall methodology due to its fixed scope, it incorporates industry practices in testing, system design, and deployment. This report outlines the rationale behind key technical decisions, architectural design, testing strategy, and project execution.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Project Context

OpenGalaxy (Figure 1.1) is a full-stack web application developed in response to the growing need for a unified platform to search, browse, and manage open-licence media. As the volume of openly licensed images and audio continues to grow across disparate repositories, users face fragmented and often unintuitive interfaces. OpenGalaxy aims to streamline this experience through a modern, performant interface powered by robust backend services.

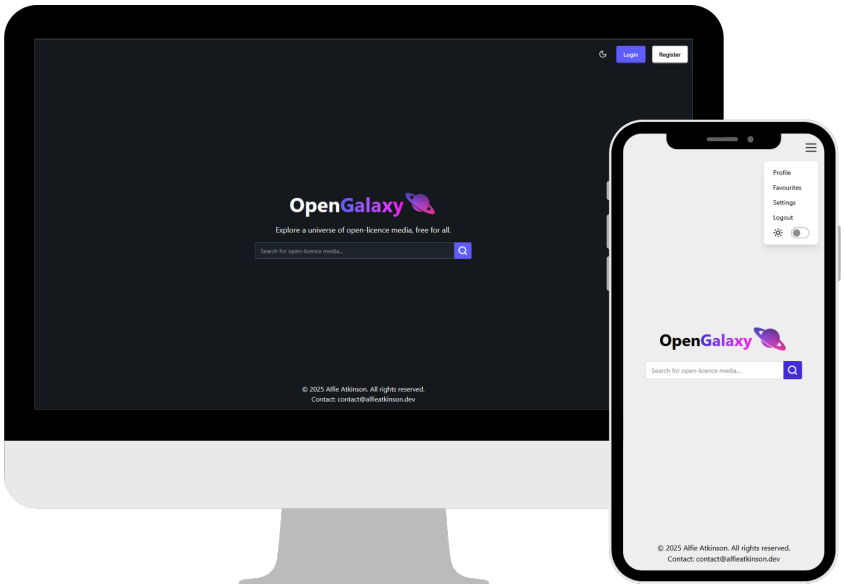

Figure 1.1: OpenGalaxy homepage interface showcasing search functionality and responsive layout.

The design of OpenGalaxy is rooted in current industry trends, particularly the shift towards API-first architecture (Gamez-Diaz, Fernandez and Ruiz-Cortes, 2017),

which decouples frontend and backend concerns and enhances scalability. The application adopts a Single Page Application (SPA) approach, leveraging React and Next.js to enable both client-side and server-side rendering (CSR/SSR). This hybrid rendering strategy improves both performance and SEO — critical for a media-heavy content discovery platform.

From a systems architecture perspective, OpenGalaxy embraces microservice principles without the full overhead of a distributed system. Each component — frontend, backend, and supporting services — without the complexity of a fully distributed system. Containerisation using docker and orchestration via Docker Compose enabled modular, reproducible environments during development and testing.

1.2 Chosen Technologies and Methods

1.2.1 Frontend Technologies

TypeScript was chosen as the primary frontend language for its strong static typing, which reduces runtime errors and enhances code reliability. It improves developer productivity through early error detection and better refactoring support, particularly in large-scale component-based architectures like React and Next.js (Emmanni, 2021). TypeScript's features such as interfaces, generics, and method overloading provide added structure and maintainability without sacrificing JavaScript's flexibility, in fact, empirical studies have shown that TypeScript has a substantial positive impact on API usability and developer productivity compared to JavaScript (Fischer and Hanenberg, 2015).

Next.js and React were selected as the framework and library due to their developer tooling and support for both CSR and SSR. React creates reusable, component-based UIs, while Next.js enhances performance and Search Engine Optimisation (SEO) via hybrid rendering strategies like Static Site Generation (SSG) and Incremental Static Regeneration (ISR) (Jartarghar et al., 2022; Ekpobimi, 2024). Next.js also includes features such as automatic code splitting, built-in image optimisation, and first-class

support for Progressive Web Apps (PWAs) (Ekpobimi, 2024), which is useful in the context of OpenGalaxy.

Combined with TailwindCSS and DaisyUI, this stack allowed for rapid styling and responsive design, while reducing DOM size by 80% (*DaisyUI* 2025) and keeping the UI consistent. Client-side interactions were validated using Zod, providing type-safe form handling.

1.2.2 Backend Technologies

The backend is built with Django and Django REST Framework (DRF), chosen for their maturity, built-in security features, admin interface, and rapid development capability. Its Model-View-Template (MVT) architecture and Object-Relational Mapping (ORM) simplify database interaction, while the Django REST Framework allows for efficient development of RESTful APIs (M. Kumar and Nandal, 2024). This allowed focus on delivering core features rather than boilerplate infrastructure. PostgreSQL was used as the primary data store, aligning with the relational structure needed for user accounts, saved media, and search history. The database was normalised to Third Normal Form (3NF).

Redis was used to cache the Openverse API token, reducing redundant network calls and improving response times for frequent queries. This in-memory caching strategy ensured low-latency access to the token while keeping the implementation simple. Although minimal, the use of Redis demonstrates how even lightweight caching can improve performance and scalability of backend services without adding unnecessary complexity.

1.2.3 Continuous Integration and Deployment Methods

For CI/CD, pipelines were implemented to test and deploy pushes to main using Vercel (frontend) and Heroku (backend). Both platforms provide automated deployments from GitHub branches, with environment variables and rollback features

integrated out of the box. Development environments were unified through Docker, with Docker Compose orchestrating service dependencies locally.

To enforce code quality and consistency, pre-commit hooks running Prettier, ESLint, Black, and Flake8 were integrated, ensuring both the TypeScript and Python codebases remained clean and standards-compliant. For end-to-end testing, Cypress was used, enabling thorough browser-based tests simulating real user flows, replacing the need for time-consuming manual testing.

1.2.4 Development Methodology

Although iterative Agile methodologies dominate in team environments (A. Kumar and Goel, 2012; Tarwani and Chug, 2016), a solo waterfall-style approach was chosen as it suits the fixed scope and timeline of this individual project. Requirements were defined up front, followed by phased implementation and testing, with documentation generated as the final deliverable. However, the model's rigidity can be a limitation when dealing with evolving requirements or short-term projects (Haniva, Ramadhan and Suharso, 2023). For simplicity, waterfall was still selected for this project.

Chapter 2

Project Planning

2.1 System Architecture Design

The System Architecture Design document (Atkinson, 2025c) for OpenGalaxy details a modular full-stack web application composed of a Next.js frontend, a Django REST Framework (DRF) backend, a PostgreSQL database, and external API integrations primarily with the Openverse API. The frontend handles UI interactions and state management via React Contexts. The backend focuses on data management, authentication, and acting as an intermediary API gateway between the frontend and Openverse.

2.1.1 High-Level Architecture Diagram

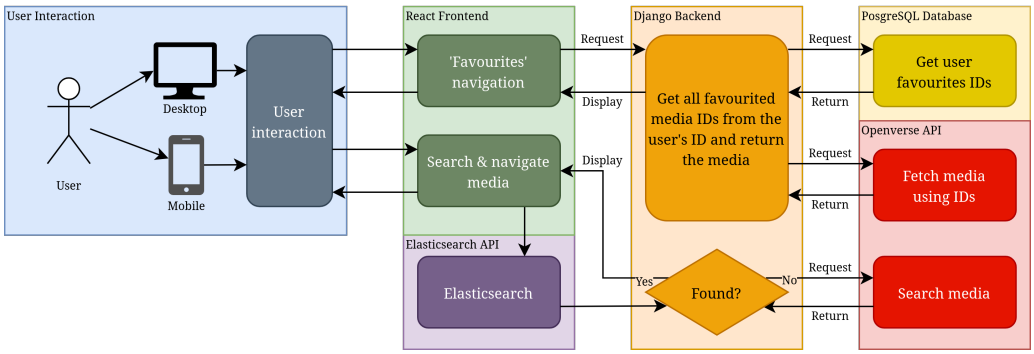

Figure 2.1: High-Level System Architecture of OpenGalaxy.

The High-Level Architecture Diagram (Figure 2.1) illustrates the clear separation between the core services:

- **Frontend:** Next.js React application hosted on Vercel with SSR and SSG for

performance, TypeScript for type-safety, and TailwindCSS with DaisyUI for concise UI.

- **Backend:** Python Django REST Framework app deployed on Heroku, responsible for user authentication, caching, and serving as the authoritative source for user data and media metadata.
- **Database:** PostgreSQL database hosted on Heroku for persistent storage of user accounts, search history, favourites, and preferences.
- **External API:** Openverse API integration for sourcing open-licence media content, supplemented by Redis caching for rate limit mitigation and performance.

2.1.2 Deployment Architecture

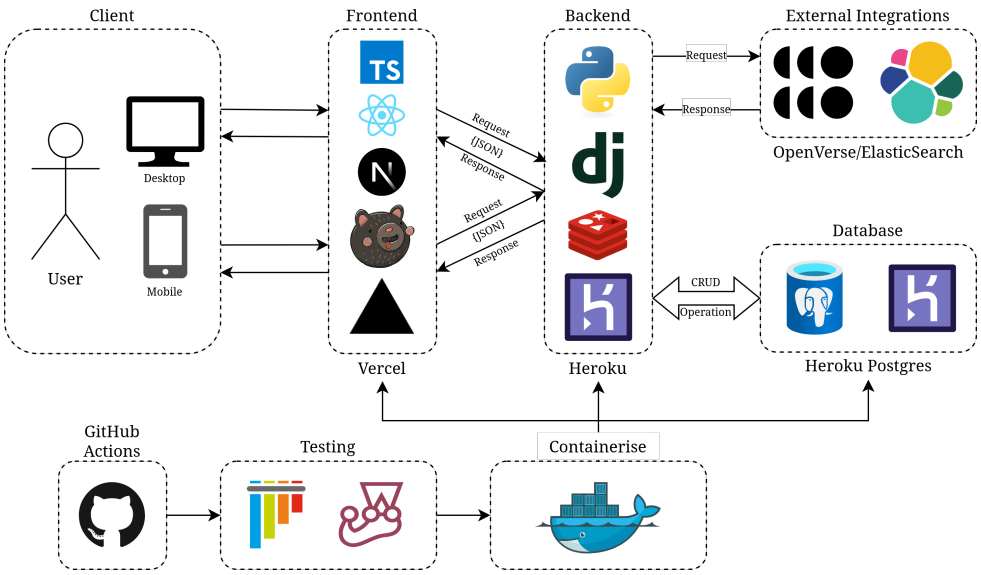

Figure 2.2: Deployment Diagram for OpenGalaxy

The platform leverages modern cloud services and containerisation to facilitate scalability, reliability, and consistency between environments. A deployment diagram (Figure 2.2) outlines the infrastructure:

- Frontend deployed on Vercel, benefitting from its global CDN and seamless Next.js optimisation.
- Backend services, including Django, PostgreSQL, and Redis, hosted on Heroku, utilising Heroku's managed add-ons for PostgreSQL and Redis caching.
- Docker ensures dev/staging parity; production uses native Heroku/Vercel services for simplicity

2.1.3 Data Flow and Interaction

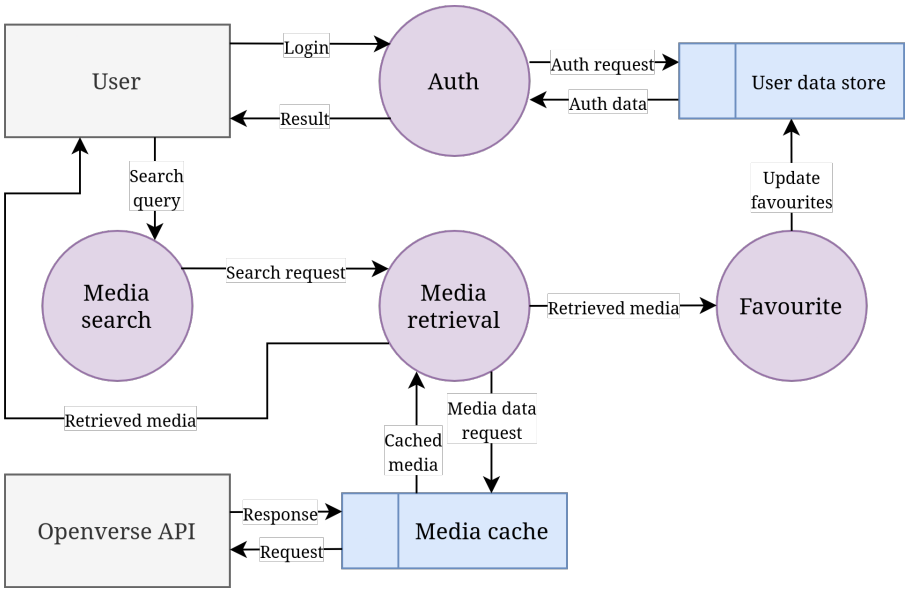

Figure 2.3: Level 1 Data Flow Diagram of OpenGalaxy.

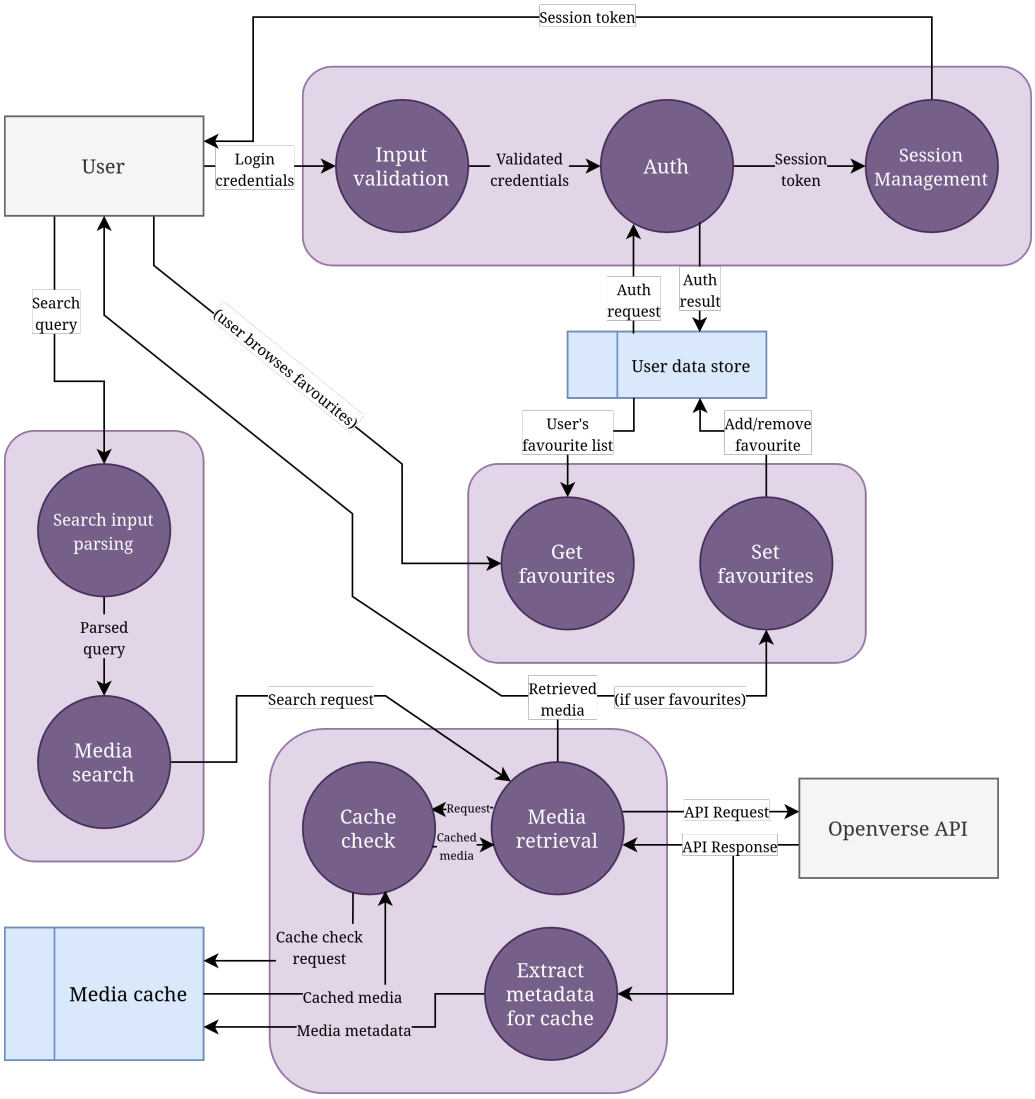

Figure 2.4: Level 2 Data Flow Diagram of OpenGalaxy.

The application's data flow is carefully mapped in a multi-level data flow diagram. The Level 1 and Level 2 Data Flow Diagrams (Figures 2.3, 2.4) detail user interactions with the system. For example, search queries: React frontend ⇒ DRF backend ⇒ Openverse API ⇒ Redis cache ⇒ frontend render. Authentication uses JWT in HttpOnly cookies, refreshed via backend endpoints. Protecting against cross-site scripting and providing stateless session management.

2.1.4 Database Design

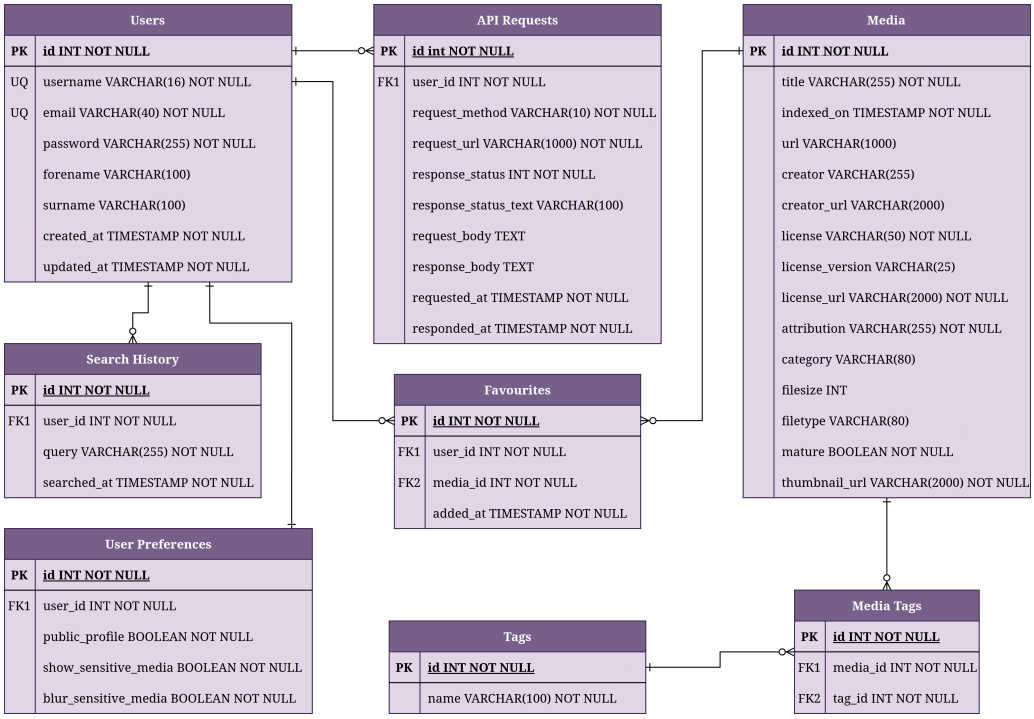

Figure 2.5: Entity Relationship Diagram for OpenGalaxy Database.

The domain-driven design extends to the database schema as well. The Entity-Relationship Diagram (ERD), shown in Figure 2.5 visualises the key entities and their relationships — including users, media, tags, and search history — and is normalised to Third Normal Form (3NF). This eliminates redundancy, enforces data integrity, and avoids anomolies during inserts, updates, and deletes so all non-key attributes depend solely on the primary key. As a result, all relationships are one-to-many or one-to-one.

2.1.5 File Structure and Object-Oriented Design

Backend Organisation

The backend follows Django's recommended app structure, broken down into domain-specific modules under `core`, including `accounts`, `media`, `search`, and `analytics`. Each module encapsulates models, views, URLs, and serializers relevant to its do-

main, promoting clean separation of concerns. For example, the `accounts` app splits user logic across multiple model files (`user.py`, `preferences.py`), while `media` cleanly isolates concerns across `media.py`, `favourite.py`, and `tag.py` models.

Frontend Organisation and Routing

The frontend uses Next.js with the App Router, structured around domain-specific routes under `src/app`. Routing is file-based and mirrors backend boundaries where applicable. For example:

- `app/media/[uuid]/page.tsx` handles rendering individual media assets.
- `app/profile/[username]/favourites/page.tsx` shows a user's favourites.
- `app/search/page.tsx` is responsible for the search results page.

API routes are colocated under `app/api`, for a cohesive directory layout.

Component Design and Layout

Components are grouped under `src/components` by domain (e.g., `media`, `search`, `profile`), improving cohesion and discoverability. Shared UI elements (buttons, headers, layouts) live under `components/shared` and `components/layout` to support DRY principles. This structure reinforces separation between business logic and presentational logic.

The design generally follows the container/presenter pattern:

- **Container components** manage state, side effects, and data fetching — e.g., the search bar component handles debouncing and query state.
- **Presenter components** focus on rendering props — e.g., `MediaCard` renders metadata without knowing where it came from.

This split facilitates testability, reuse, and code clarity.

State Management and Caching

Global application state is handled via the Context API (e.g., `AuthContext`) to avoid introducing unnecessary third-party dependencies like Zustand. This minimal setup is appropriate given the app's complexity and helps keep bundle size lean.

A bespoke caching utility based on localStorage provides TTL-based storage for search results. Implemented in `cache.ts`, this utility supports simple client-side persistence with expiry logic and stale entry eviction — an effective but lightweight solution for reducing redundant API calls.

2.1.6 Sequence and Interaction Diagrams

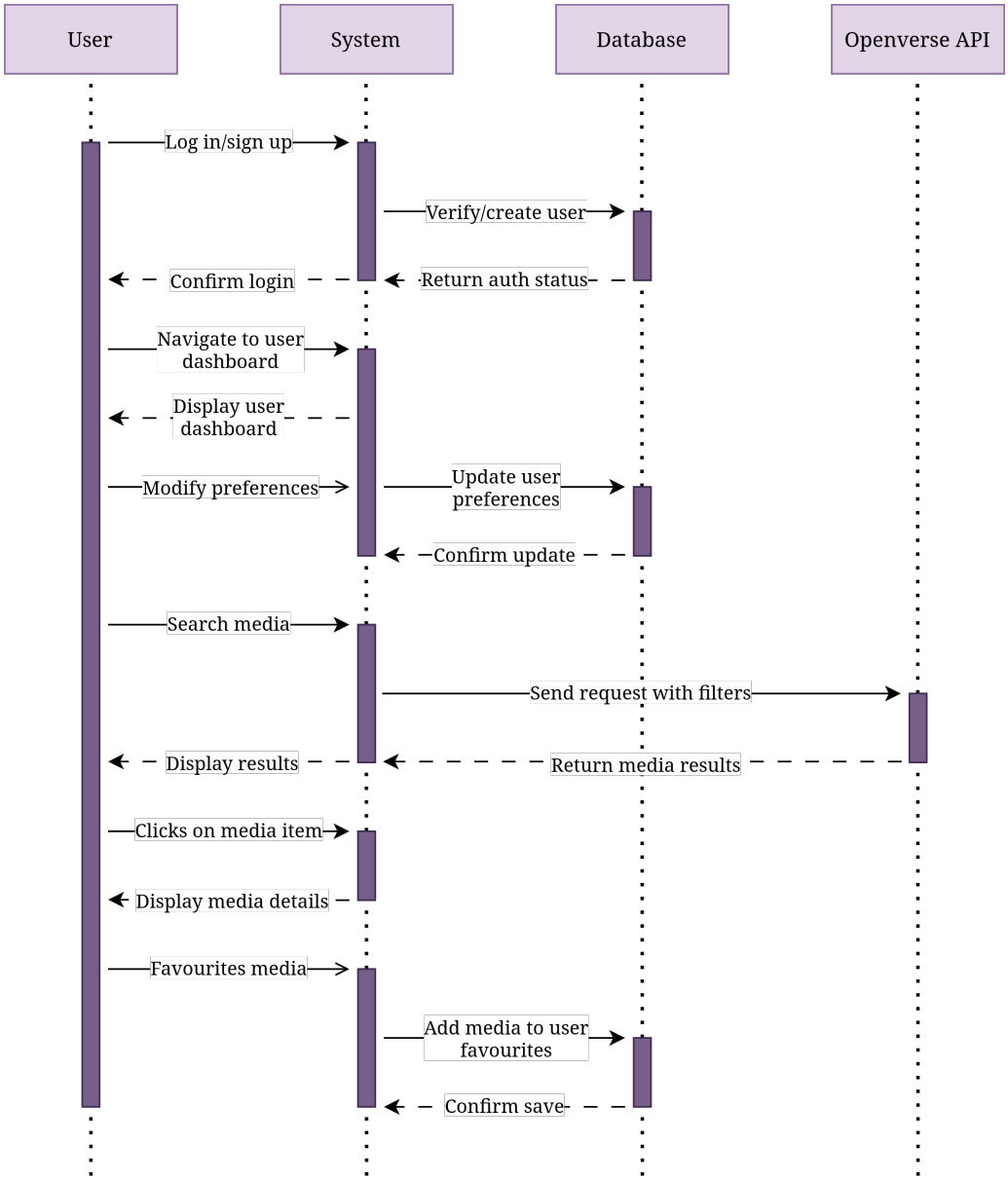

Figure 2.6: Sequence Diagram of a Typical User Journey in OpenGalaxy.

Two key user journeys are highlighted in Figure 2.6: authentication and search.

2.2 Software Requirements Specification

The Software Requirements Specification (SRS) document (Atkinson, 2025b) was developed to clearly outline the functional and non-functional requirements of the OpenGalaxy application. It specifies core user needs, system features, and constraints that guide the design and development process. This foundational document served as a reference keep alignment between requirements and implementation.

2.3 Timeline and Milestones

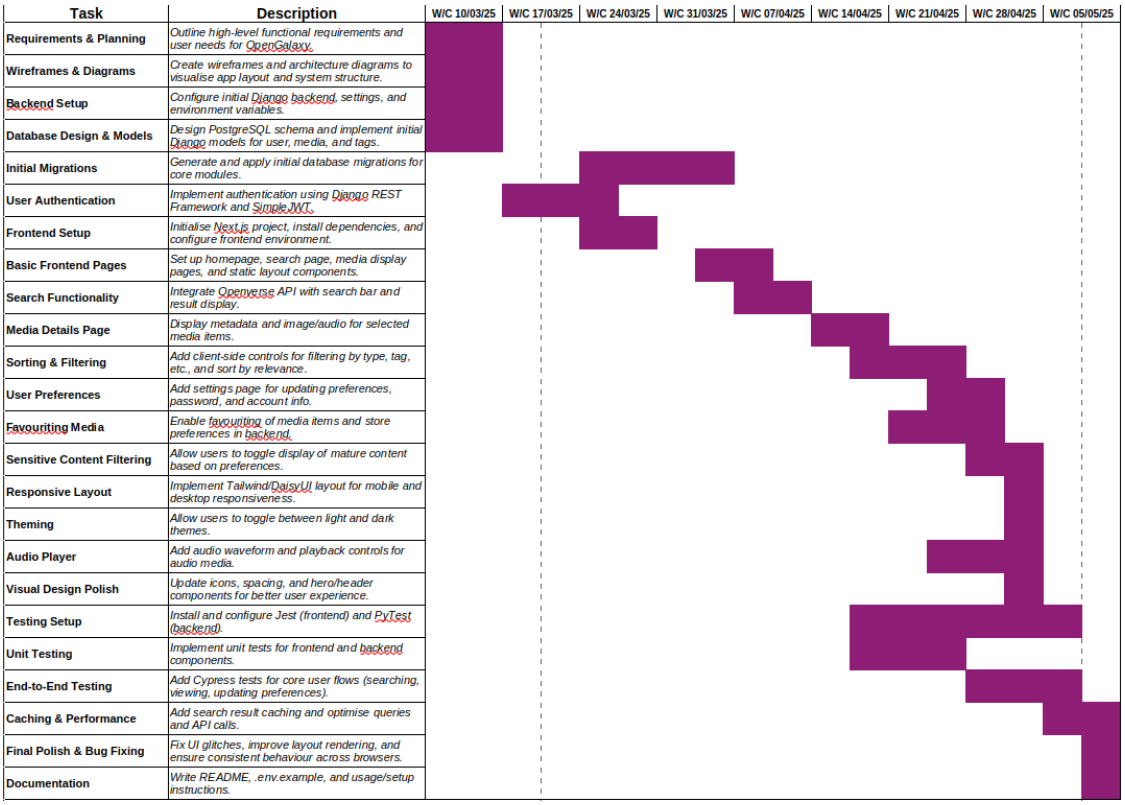

Figure 2.7: Gantt chart for OpenGalaxy development timeline and milestones.

The Gantt chart in Figure 2.7 illustrates the project timeline, highlighting key milestones and deliverables. The timeline was structured around iterative development cycles, with each phase building on the previous one.

2.4 Risk Assessment and Mitigation

- **Openverse API Rate Limits:** To prevent service disruptions, a caching layer using Redis stores frequently and recently accessed media metadata. Retry logic with exponential backoff in `cache.ts` ensures graceful handling of transient failures.
- **Cross-Origin Authentication Challenges:** Implemented CORS configuration in Django's `settings.py` alongside secure JWT handling using HttpOnly cookies to prevent token theft and cross-site attacks.
- **Scope Creep in Solo Development:** Strict feature branch workflows with pre-commit hooks and defined goals maintained manageable progress, avoiding over-extension.
- **Recovery and Resource Constraints:** Early adoption of Cypress tests mitigated reduced manual testing capacity during surgery recovery, ensuring automated regression coverage.

Chapter 3

Project Management and Development Process

3.1 GitHub Repository

The entire codebase for this project can be found on the GitHub repository (Atkinson, 2025a), and the live website can be found at opengalaxy.alfieatkinson.dev. The repository serves as the primary documentation of the development process, structured around best practices in modern software engineering, such as feature-driven development, automated testing, and traceable version control.

3.1.1 Workflow and Branching Strategy

Feature branch workflows are widely used in software development to manage collaborative coding and ensure code quality. They involve creating separate branches for new features, which are then reviewed before integration into the main codebase (Krusche, Berisha and Bruegge, 2016). This improves code quality, allows parallel progress, and facilitates easy review (Krusche, Berisha and Bruegge, 2016). Branches follow a clear naming convention aligned with their function, e.g., `feature/search-filter`, `feature/authentication`, `fix/search-issue`. This clarity helped maintain organisation and speed during development.

GitHub Actions workflows were configured to automate critical processes. A deployment pipeline ensures that only changes within the `backend/` directory trigger redeployment to Heroku, preserving frontend independence and improving deployment clarity. Another workflow automatically runs Cypress end-to-end tests on every pull request and push to `main`, preventing regressions and maintaining confidence in user-critical flows.

Pull Requests (PRs) are raised for every feature or fix and undergo self-review against a checklist before merging. The checklist includes:

- Code styling consistency (using Prettier and linting hooks).
- Functional testing (end-to-end with Cypress, live preview on Vercel).
- No unresolved comments or warnings.
- Proper PR description and linked issues (`Closes #X`) for traceability.

Merges are squash merges to keep history clean.

3.1.2 GitHub Issues and Project Board

GitHub Issues were used for task tracking, labelling issues with 'enhancement' or 'bug' tags. Each issue corresponds to a PR, and PRs explicitly close related issues. This issue-based workflow helped keep clear scope boundaries and allow for granular progress tracking.

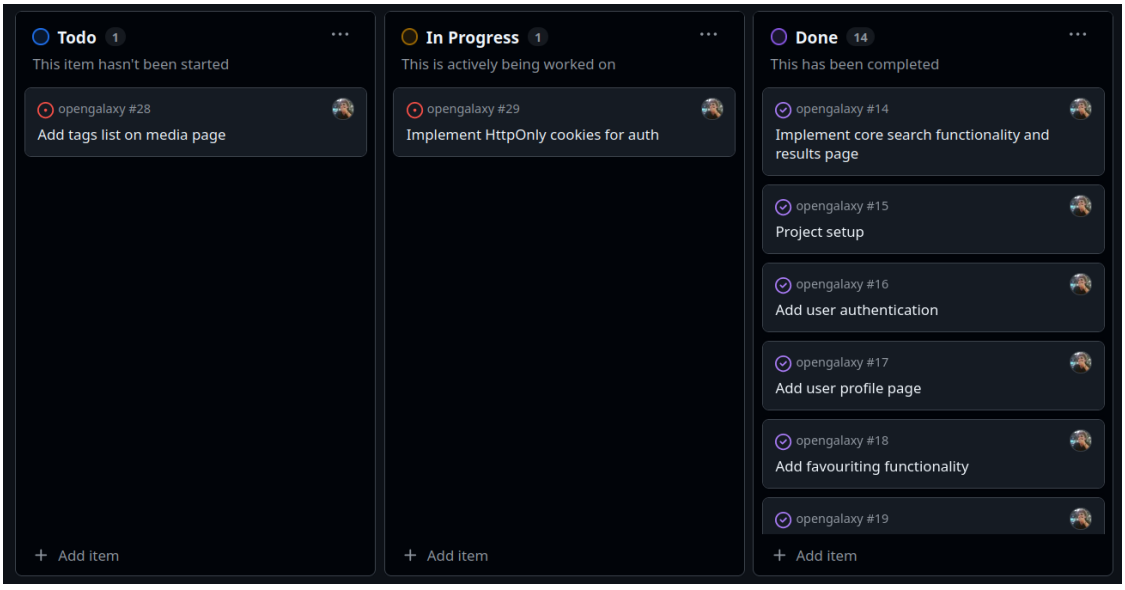

Figure 3.1: GitHub Projects Kanban board to track GitHub Issues for OpenGalaxy.

The issues were organised using GitHub Projects and a Kanban board (3.1) structured into 'To Do', 'In Progress', and 'Done' columns. Cards are linked to the

corresponding issues and PRs, providing visual progress and backlog prioritisation. Custom issue and pull request templates were created to standardise contributions. The templates enforce a clear structure for bug reports and feature suggestions.

3.1.3 Commit and PR History

Commits and PRs were granular and descriptive, reflecting incremental development. Key highlights include:

- `#40 Switch from localStorage auth to JWT HttpOnly cookies`: Replaced localStorage-based authentication with secure cookie-based JWT token management to enhance security and streamline token handling. Implemented custom backend authentication class, cookie-aware token views, and updated the frontend to remove reliance on localStorage. Also improved proxy handling and standardised code style in the settings.
- `#13 Add Cypress e2e for core flows`: Implemented end-to-end testing prioritising real user journeys over unit tests, due to time constraints. Critical for stable core functionality like login, search, and profile management.
- `#10 Add settings page`: Delivered a comprehensive user settings page supporting profile updates, password changes, and account deletion, emphasising user experience and security.

Pre-commit hooks for linting and formatting were configured via Husky and a `pyproject.toml` file to maintain consistent code quality across the codebase.

3.2 Trade-Offs and Iterations

- **State management:** Initially considered Zustand but switched to React Contexts for simplicity and reduced dependencies, which streamlined development and debugging.
- **Search engine:** Planned to use Elasticsearch for advanced search capabil-

ities but dropped this due to time constraints and infrastructure complexity. Instead, built a custom filtering and caching mechanism sufficient for MVP needs.

- **Testing strategy:** Pivoted from unit testing to end-to-end testing using Cypress to focus on validating complete user flows, which better reflects real usage and prioritises critical functionality within limited time.
- **Caching:** Opted for an in-process cache for search results rather than Redis to reduce operational complexity, as caching needs were moderate.
- **Authentication:** Replaced localStorage-based JWT authentication with HttpOnly cookies following a security review, significantly reducing XSS risk and simplifying token management across backend and frontend.
- **Incremental business domain addition:** Developed features iteratively by focusing on one domain at a time (authentication, search, media management, user profile, settings, etc), which helped isolate issues and maintain code clarity.
- **UX prioritisation:** Kept smooth user flow as the primary goal, often sacrificing feature breadth to guarantee core flows (login, search, media browsing) worked flawlessly. Ensured website is fully responsive and accessible, with mobile-first design principles applied throughout.

Mid-project disruptions, such as surgery recovery, necessitated adjustment of scope and priorities. To mitigate regression risks during this period, Cypress was introduced early for end-to-end testing instead of Jest unit testing. This choice reflects a pragmatic focus on core user journeys, especially given Django's role as primarily a data and API interface rather than complex business logic.

Chapter 4

Testing, Building, and Containerisation

4.1 Testing, Building, and Containerisation

4.1.1 Testing Methodology

This project employed a layered and pragmatic testing strategy, prioritising realistic end-to-end (E2E) coverage over exhaustive unit testing. The chosen approach reflects both the system's architecture and the time constraints of the development timeline.

Unit Testing

Unit testing with PyTest was initially considered for the Django backend; however, given the backend's limited business logic and its role as a thin layer over the database and external API, unit testing was deprioritised in favour of integration and E2E coverage, where issues are more likely to surface

On the frontend, Jest was planned to provide unit and integration tests for reusable components and local logic. However, given limited time, focus shifted toward end-to-end coverage that more directly benefits user experience validation.

Integration and End-to-End Testing

End-to-end tests were implemented extensively using Cypress, simulating real user interactions across the full stack. These covered key user workflows such as account registration and login, search with filtering and sorting, profile updates, settings management, and user search history. To streamline test execution, setup steps like

user registration are performed via direct API calls, after which the UI is tested to ensure consistency between frontend presentation and backend state.

Cypress tests are further enhanced through the use of network interception and response stubbing. This technique avoids reliance on live backend data, eliminating flakiness caused by asynchronous delays or unpredictable external behaviour. As a result, tests remain deterministic, fast, and reliable—critical traits for automated regression testing.

Test Coverage and Code Quality Tools

While unit test coverage is minimal, test effort was concentrated on the application's critical user-facing paths. Given the nature of the system, this prioritisation of E2E coverage over fine-grained unit tests supports a better return on time invested in testing.

StaStatic analysis and code formatting are enforced locally via pre-commit hooks, implemented with Husky and configured in the project's `pyproject.toml`. These include Black and Flake8 for Python, and ESLint and Prettier for TypeScript. While these checks are not enforced in CI, they help maintain code consistency during local development and lay the groundwork for future server-side enforcement via CI.

4.1.2 CI/CD Pipelines

Continuous integration and deployment processes were designed to minimise friction and ensure rapid iteration during development.

The frontend is deployed automatically via Vercel, with pushes to the `main` branch triggering new deployments. This integration supports efficient feedback cycles and ensures production remains up to date without manual intervention.

Similarly, the backend is deployed on Heroku using a GitHub-connected pipeline and a declarative `Procfile`. Pushes to the `main` branch automatically trigger backend updates, including database migrations and static file collection.

GitHub Actions are configured to run end-to-end Cypress tests on all pull requests to the `main` branch. This ensures that any changes to the codebase do not introduce regressions in critical user flows. Another GitHub Action workflow is set up to run on pushes to the `backend/` directory, making sure only relevant changes trigger backend redeployments.

4.1.3 Building and Environment Configuration

Frontend and Backend Build Processes

The frontend is built using the standard Next.js production build command (`npm run build`), which performs optimisations such as TypeScript minification, dead-code elimination (tree shaking), and bundling of static assets. These optimisations are applied without requiring custom configuration, reducing load times and improving user experience.

On the backend, Django is containerised and configured to run with Gunicorn as the WSGI server. On container startup, database migrations are automatically applied to keep the schema synchronised, and static files are collected for serving via the web server.

Environment Management

Sensitive configuration such as API keys and database credentials are managed through environment variables injected at container runtime via `.env` files. This separation of configuration from code allows seamless transitions between development, staging, and production environments without altering application logic.

4.1.4 Containerisation Strategy

Local Development with Docker Compose

The project uses Docker and Docker Compose to orchestrate a reproducible development environment, comprising the frontend, backend API, PostgreSQL database,

and Redis cache. Each service runs in its own isolated container with a shared virtual network, simulating production-like conditions during development.

Persistent volumes are used to cache database and dependency layers, reducing rebuild times. Startup order and inter-service dependencies are managed via custom scripts such as `entrypoint.sh` and `wait-for-postgres.sh`, which ensure services only start once their dependencies are healthy.

Production Deployment Decisions

In production, containerisation is not used directly. Instead, the project leverages platform-managed deployment services—Vercel for the frontend and Heroku for the backend. These services abstract away the underlying infrastructure and offer integrated scaling, HTTPS, and zero-downtime deployments. This trade-off limits runtime control but significantly reduces operational complexity, which is appropriate for a project of this scale.

Despite not using containers in production, the Docker setup remains critical for local development and testing, ensuring parity between environments and supporting future scalability.

Scalability and Portability

Docker Compose enables rapid onboarding, predictable environments, and the potential for scalable production deployments via container orchestration platforms like Kubernetes or Heroku's container stack. Scripts exist to support horizontal scaling of backend dynos, illustrating the project's readiness to grow without fundamental architectural changes.

Chapter 5

Social, Ethical, and Entrepreneurial Implications

5.1 Social Implications

OpenGalaxy's primary societal value lies in its role as a public-good platform. By aggregating open-licence media in one accessible interface, it promotes digital inclusivity and lowers the barrier of entry for creators, educators, journalists, and researchers. Users from low-income backgrounds, small organisations, or regions with limited access to commercial assets benefit from freely accessible, high-quality media.

5.1.1 Accessibility

Accessibility was an important consideration during development, particularly in structuring the HTML semantically, elements such as `<main>`, `<section>`, and `footer` were used to help screen readers interpret the page structure, and headings followed a logical hierarchy. Form elements, including search bars and login inputs, use proper labels and input types.

Basic support for keyboard and touch-based navigation as well as screen readers is in place, though certain accessibility improvements, such as theme toggles and full mobile responsiveness, are scheduled for future implementation due to time constraints.

5.1.2 Digital Licence Literacy

Open-licence media is commonly used across creative and digital projects, but its adoption is often marred by misunderstanding and misuse. Studies have found that a significant portion of Creative Commons (CC) licensed content — such as images on

Flickr, a main source of Openverse — is embedded without proper attribution, with violation rates as high as 70-90% (Seneviratne, Kagal and Berners-Lee, 2009). This is partly due to the perceived simplicity of CC licensing, which can mask its legal nuance (Koščík and Šavelka, 2013). These misunderstandings not only pose legal risks for users, but have also created opportunities for bad-faith actors to exploit attribution failures for financial gain (Stewart, 2021).

To address this, OpenGalaxy ensures that every item of media clearly displays its licence, along with a direct link to accessible information explaining what that licence entails. This aims to reduce accidental misuse and limit opportunities for exploitation.

5.2 Ethical Implications

The ethical responsibility of any aggregation platform includes respecting original authorship, ensuring correct attribution, and avoiding exploitation of creators. Although OpenGalaxy sources only openly licensed content, not all repositories have equally rigorous curation or metadata standards. Relying on external APIs like Openverse means some content may be incorrectly tagged or incompletely attributed, as shown by Seneviratne, Kagal and Berners-Lee (2009). This creates a risk of propagating attribution errors or surfacing inappropriate or offensive content.

From a data ethics perspective, OpenGalaxy collects minimal user data (e.g., account info, search history, saved media). This data is stored securely and never shared with third parties. JWT-based authentication and HttpOnly cookies reduce the risk of XSS and session hijacking, though no system is fully immune. As per GDPR principles, user data must be handled with transparency, minimisation, and the right to deletion — all of which are respected in the system's design.

5.3 Entrepreneurial Implications

5.3.1 Potential and Challenges

Open source software and open content models represent a fundamental shift in business paradigms, transcending conventional economic assumptions and offering novel eBusiness opportunities (Clarke, 2004). Unlike traditional creative industries that rely on tight content protection, open content platforms embrace collaboration and shared value, focusing on the needs of multiple stakeholders — creators, consumers, and distributors — rather than pure profit maximisation.

Though OpenGalaxy is a not-for-profit academic project, its foundation reflects this emerging entrepreneurial potential. Platforms utilising open content have found sustainable models through donations, freemium API access, or institutional partnerships. A commercial iteration of OpenGalaxy could similarly add value by offering premium services such as batch downloads, integration with content management systems, or enhanced filtering tools tailored for journalists and creators.

Technically, OpenGalaxy's use of scalable, open technologies like React, Django, and PostgreSQL, coupled with containerised, cloud-deployable architecture, positions it well for rapid expansion with low infrastructure costs.

The key entrepreneurial challenge lies in achieving sustainability without compromising open culture principles. Ethical monetisation requires creating genuine added value — such as curation, enhanced user experience, or educational resources — rather than merely repackaging publicly available content. Failure to do so risks alienating the open content community and potentially breaching upstream API terms.

5.3.2 Future Work

Having established a stable and functional MVP, the next phase of development will focus on deepening interactivity, enhancing user engagement, and expanding backend

capabilities. These planned enhancements will continue to align with OpenGalaxy's ethos of openness, attribution clarity, and user-centric design.

Several frontend improvements could increase visual appeal and responsiveness. Including the addition of smooth, accessible animations for state changes (e.g., loading indicators and feedback on actions) and a dynamic ambient background — such as a twinkling starfield — to reinforce the thematic identity of the platform.

A major long-term goal is the implementation of social and curation functionality. Users will be able to create and share curated collections of media, follow other users, and access personalised media feeds based on social connections. This will transform OpenGalaxy into a community-centric discovery platform, loosely inspired by the mechanics of Pinterest but focused on openly licensed content.

From an infrastructure perspective, plans include:

- Adding OAuth-based authentication (e.g., Google, GitHub) to reduce friction in user sign-up and login flows.
- Enhancing backend scalability and performance through query optimisation and additional caching strategies, particularly to support the social features and personalised feeds.
- Potential development of administrative tools or moderation workflows to support future growth and community safety.

As the project matures, a plugin architecture remains a long-term ambition. This would enable external contributors to introduce new search sources, filters, or visualisation tools, keeping the ecosystem extensible and open to innovation.

These future additions will be driven by both user feedback and technical feasibility, with a focus on maintaining performance and accessibility across devices.

References

Atkinson, Alfie (2025a). *OpenGalaxy GitHub repository*. URL: https://github.com/alfieatkinson/opengalaxy (cit. on p. 15).

– (2025b). *OpenGalaxy Software Requirements Specification*. URL: https://raw.githubusercontent.com/alfieatkinson/opengalaxy/main/docs/Software-Requiements-Specification.pdf (cit. on p. 13).

– (2025c). *OpenGalaxy System Architecture Design*. URL: https://raw.githubusercontent.com/alfieatkinson/opengalaxy/main/docs/System-Architecture-Design.pdf (cit. on p. 5).

Clarke, Roger (2004). 'Open source software and open content as models for eBusiness'. In: (cit. on p. 25).

DaisyUI (2025). URL: https://daisyui.com/ (cit. on p. 3).

Ekpobimi, HO (2024). 'Building high-performance web applications with NextJS'. In: *Computer Science & IT Research Journal* 5.8, pp. 1963–1977 (cit. on pp. 2, 3).

Emmanni, P Sekhar (2021). 'The role of typescript in enhancing development with modern javascript frameworks'. In: *Int. J. Sci. Res* 10.2, pp. 1738–1741 (cit. on p. 2).

Fischer, Lars and Stefan Hanenberg (2015). 'An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio'. In: *ACM SIGPLAN Notices* 51.2, pp. 154–167 (cit. on p. 2).

Gamez-Diaz, Antonio, Pablo Fernandez and Antonio Ruiz-Cortes (2017). 'An analysis of RESTful APIs offerings in the industry'. In: *International Conference on Service-Oriented Computing*. Springer, pp. 589–604 (cit. on p. 1).

Haniva, Diandara Tresya, Jadid Alif Ramadhan and Aries Suharso (2023). 'Systematic Literature Review Penggunaan Metodologi Pengembangan Sistem Informasi Waterfall, Agile, dan Hybrid'. In: *JIEET (Journal of Information Engineering and Educational Technology)* 7.1, pp. 36–42 (cit. on p. 4).

Jartarghar, Harish A et al. (2022). 'React apps with Server-Side rendering: Next. js'. In: *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 14.4, pp. 25–29 (cit. on p. 2).

Koščík, Michal and Jaromír Šavelka (2013). 'Dangers of over-enthusiasm in licensing under creative commons'. In: *Masaryk University Journal of Law and Technology* 7.2, pp. 201–227 (cit. on p. 24).

Krusche, Stephan, Mjellma Berisha and Bernd Bruegge (2016). 'Teaching code review management using branch based workflows'. In: *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 384–393 (cit. on p. 15).

Kumar, Akhil and Bindu Goel (2012). 'Factors influencing agile practices: A survey'. In: *International Journal of Engineering Research and Applications* 2.4, pp. 1347–1352 (cit. on p. 4).

Kumar, Manoj and Rainu Nandal (2024). 'Role of Python in rapid web application development using Django'. In: *Available at SSRN 4751833* (cit. on p. 3).

Seneviratne, Oshani, Lalana Kagal and Tim Berners-Lee (2009). 'Policy-aware content reuse on the web'. In: *The Semantic Web-ISWC 2009: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings 8*. Springer, pp. 553–568 (cit. on p. 24).

Stewart, Daxton R (2021). 'Rise of the copyleft trolls: When photographers sue after creative commons licenses go awry'. In: *Ohio St. Tech. LJ* 18, p. 333 (cit. on p. 24).

Tarwani, Sandhya and Anuradha Chug (2016). 'Agile methodologies in software maintenance: A systematic review'. In: *Informatica* 40.4 (cit. on p. 4).