Project - High Level Design

On

Task Atlas
(CI/CD for Flask Technology Application)

Course Name: DevOps Foundation

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1.	Aryan Gangrade	EN22IT301023
2.	Suzal Hasam Ali Devani	EN22IT301111
3.	Ayushi Jat	EN22IT301028
4.	Naina Sahu	EN24CA5030106
5.	Aditi Singh	EN24CA5030008
6.	Preeti Prajapat	EN24CA5030131

Group Name: Group11 D12

Project Number: DO-49

Industry Mentor Name: Mr.Vaibhav Tiwari

University Mentor Name: Prof.Akshay Saxena

Academic Year: 2025-26

Table of Contents

1. Introduction.
   1.1. Scope of the document.
   1.2. Intended Audience
   1.3. System overview.

2. System Design.
   2.1. Application Design
   2.2. Process Flow.
   2.3. Information Flow.
   2.4. Components Design
   2.5. Key Design Considerations
   2.6. API Catalogue.

3. Data Design.
   3.1. Data Model
   3.2. Data Access Mechanism
   3.3. Data Retention Policies
   3.4. Data Migration

4. Interfaces

5. State and Session Management

6. Caching

7. Non-Functional Requirements
   7.1. Security Aspects
   7.2. Performance Aspects

8. References

1. Introduction

1.1 Scope of the Document :-

This High Level Design (HLD) document provides a comprehensive overview of the Task Atlas application. The purpose of this document is to describe the overall architecture and design of the system at a high level. It covers the main functional components, system workflows, data flow, integration points, and non-functional requirements such as security and performance.

This document focuses only on high-level design aspects and does not include low-level implementation details such as source code, database queries, or internal algorithms. The scope is limited to components that are directly relevant to the Task Atlas system and its intended functionality.

1.2 Intended Audience :-

The intended audience for this document includes faculty members, project reviewers, system architects, developers, and other stakeholders who are involved in understanding or evaluating the Task Atlas application. This document is especially useful for readers who need a clear understanding of how the system is structured, how different components interact with each other, and what technologies are used, without going into technical coding details.

1.3 System Overview :-

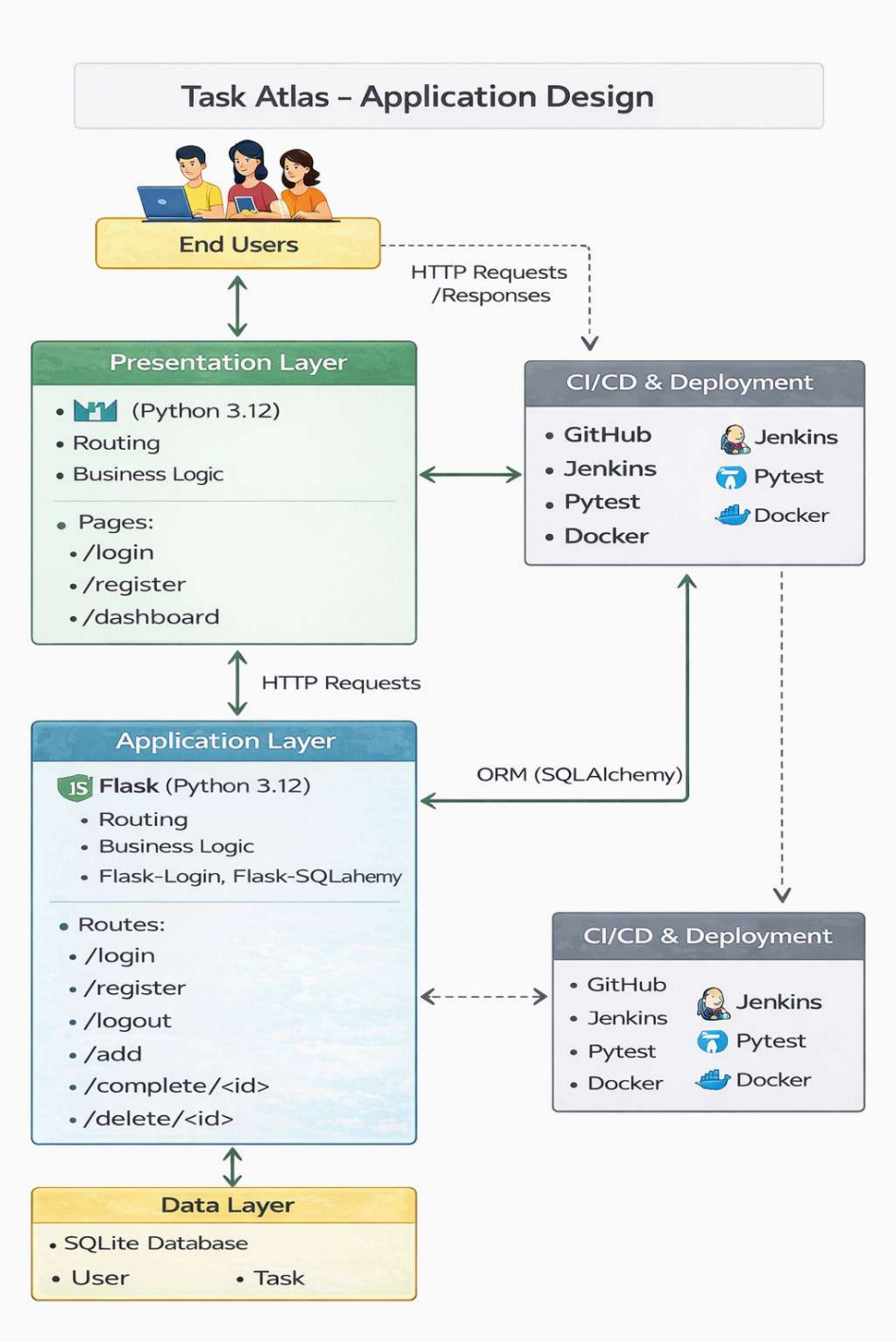
Task Atlas is a lightweight, web-based task management application built using Flask. It is designed to provide a clean and focused environment for managing personal to-do lists in a multi-user system. Each registered user has a private workspace where tasks can be added, marked as completed, or deleted. The system ensures user-specific data isolation through secure authentication and session management using Flask-Login, while persistent storage is handled using a local SQLite database.

The application follows a client–server architecture, where the backend manages routing, authentication, and business logic, and the frontend delivers an interactive user interface using Jinja2 templates and CSS. The user interface is designed with a modern glassy look to enhance usability and visual appeal. Flask-SQLAlchemy is used to manage database interactions, ensuring structured data access and easy maintainability of the system.

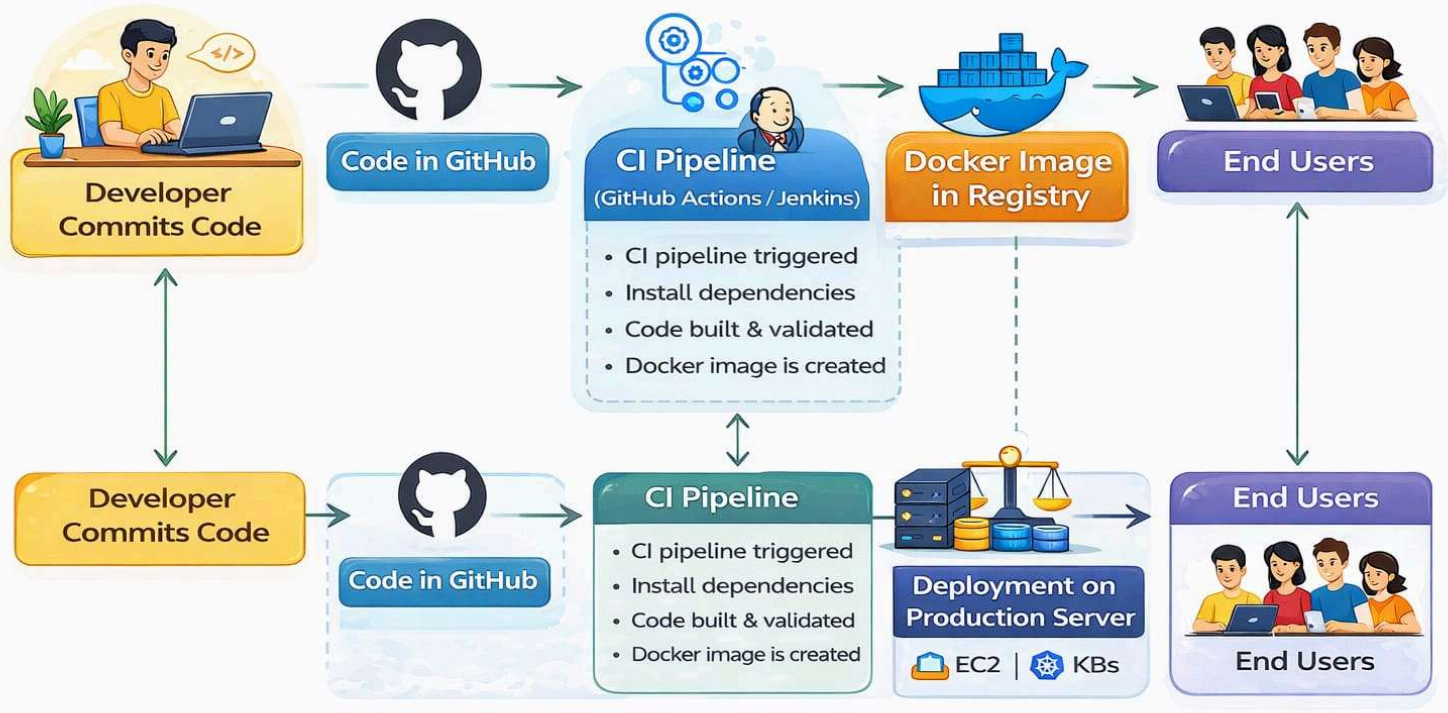
Task Atlas is developed with minimal dependencies to enable fast setup and smooth execution. The system supports automated testing, containerization, and deployment through an integrated CI/CD pipeline using Jenkins, Docker, and Pytest. This setup helps maintain code quality and enables reliable deployment workflows. Overall, Task Atlas is a secure, efficient, and maintainable application suitable for academic projects and small-scale personal task management.

2. System Design

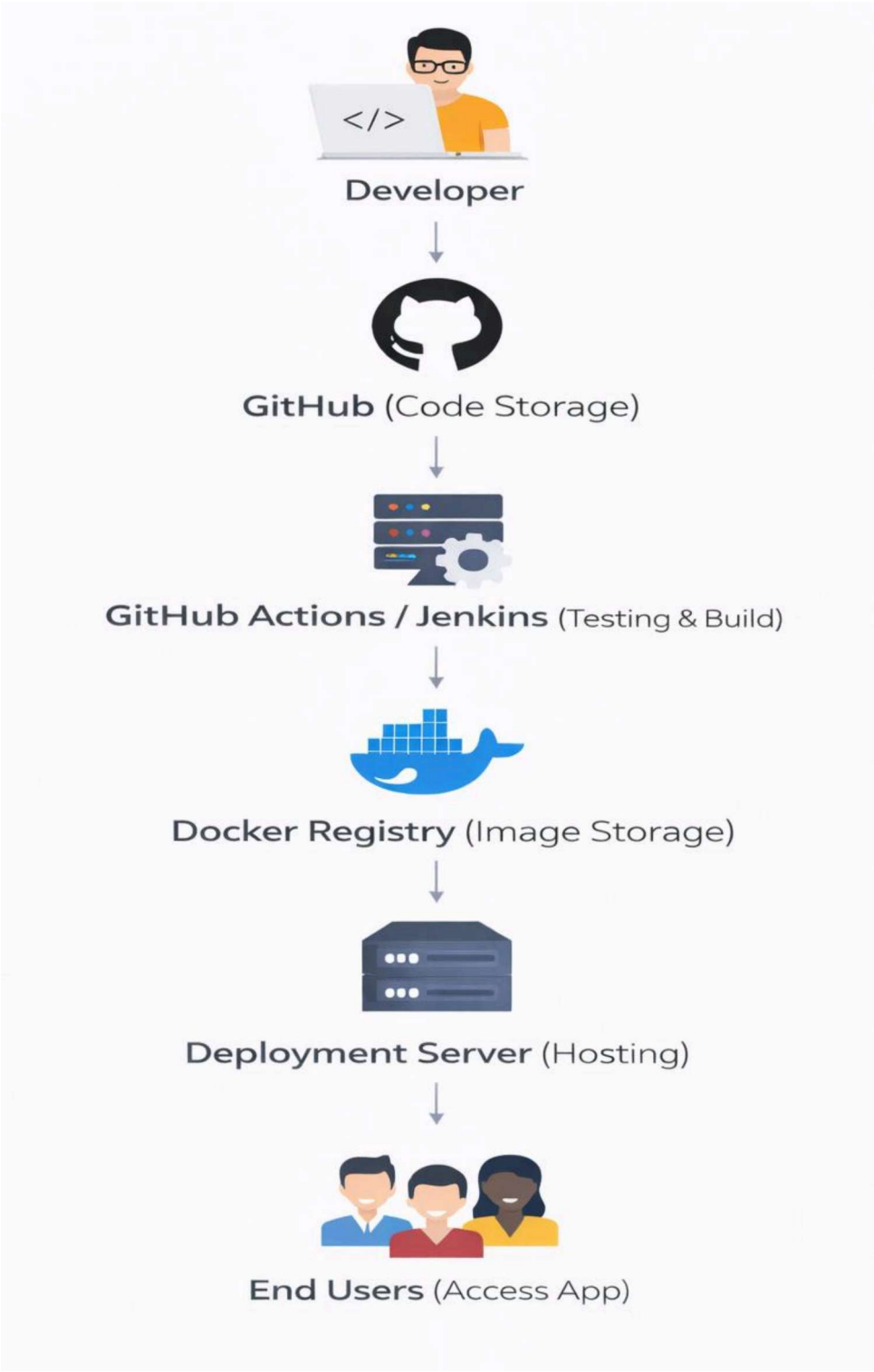
2.1 Application Design



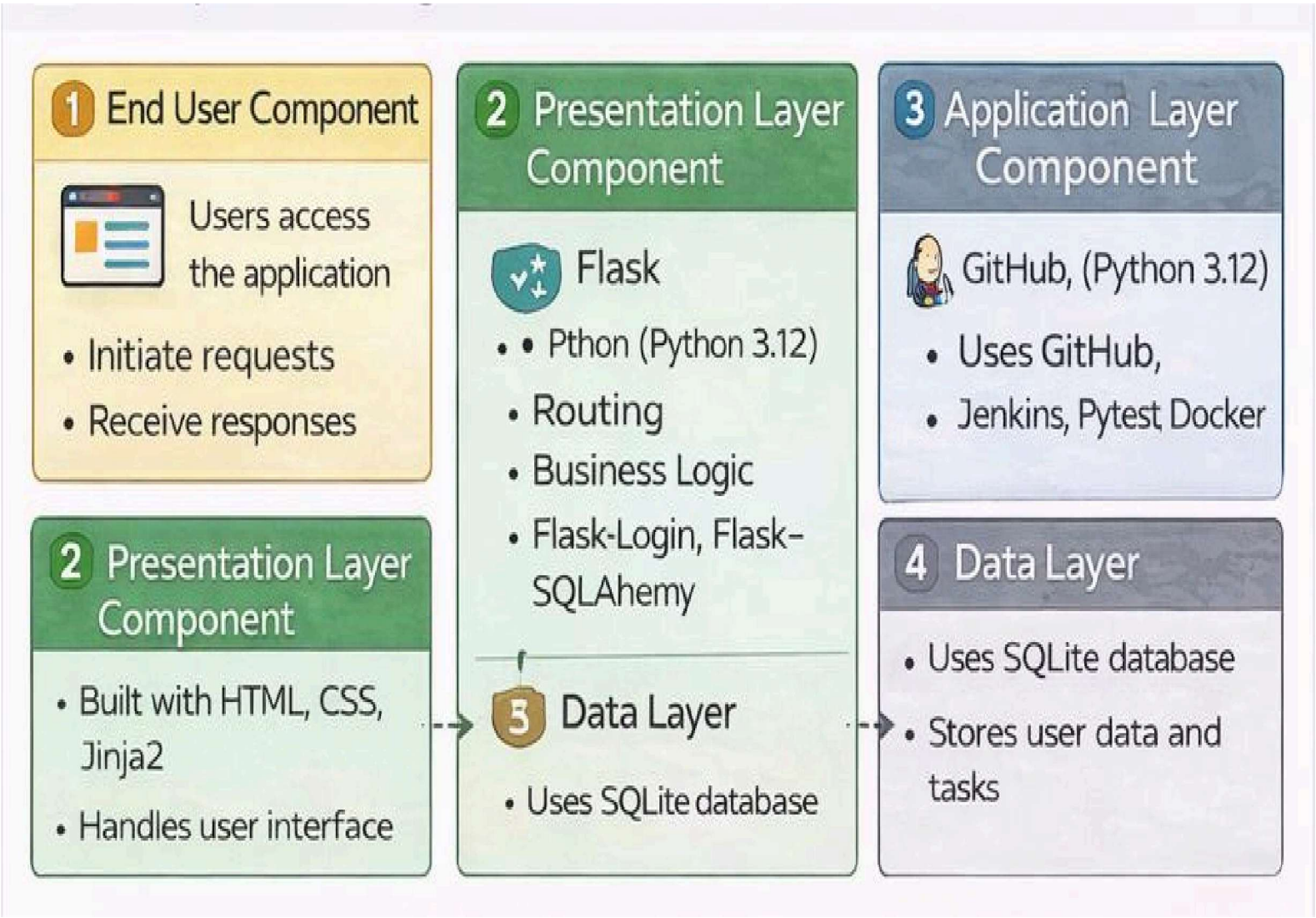
2.2 Process Flow



2.3 Information Flow



2.4 Component Design



2.5 Key Design Considerations

1. Simplicity

- Use of Flask to keep the application lightweight
- Minimal dependencies for faster setup
- Easy to understand and maintain codebase
- Suitable for academic and small-scale use

2. Security

- User authentication using Flask-Login
- Encrypted password storage
- Secure session handling using SECRET_KEY
- Restricted access to user-specific data

3. Data Integrity

- One-to-many relationship between User and Task
- Use of SQLAlchemy ORM for safe database operations
- Each task mapped to a specific user
- Prevention of unauthorized data access

4. Scalability

- Architecture allows database upgrade in future
- Modular design supports feature expansion
- Docker enables environment portability
- CI/CD supports future deployment scaling

5. Maintainability

- Clear separation of presentation and business logic
- ORM simplifies database changes
- Modular route and component structure
- Easy debugging and testing

6. Performance

- Lightweight framework ensures fast response
- Direct database access avoids overhead
- Minimal server resource usage

- Browser caching for static files

7. Reliability

- Automated testing using Pytest
- CI pipeline with Jenkins
- Consistent builds via Docker
- Reduced deployment errors

8. Portability

- Docker containerization
- Runs consistently across environments
- Easy deployment on different servers
- Supports future cloud deployment

2.6 API Catalogue

The Task Atlas application exposes the following internal APIs (routes) to support its functionality:

API Endpoint	HTTP Method	Description
/login	GET / POST	Displays login page and authenticate user
/register	GET / POST	Registers a new user
/logout	GET	Logs out the current user
/	GET	Displays user dashboard
/add	POST	Adds a new task
/complete/<id>	GET	Toggles task completion status
/delete/<id>	GET	Deletes a task

3. Data Design

3.1 Data Model

Task Atlas uses a simple and relational data model designed to support multi-user task management. The system consists of two main entities: User and Task. Each task is associated with a specific user, ensuring data isolation and security.

User Entity:-

- id (Primary Key)
- username (Unique)
- password (Encrypted)

Task Entity:-

- id (Primary Key)
- content (Task description)
- completed (Boolean status)
- user_id (Foreign Key referencing User.id)

The relationship between User and Task is one-to-many, where one user can have multiple tasks.

3.2 Data Access Mechanism

Data access in Task Atlas is handled through Flask-SQLAlchemy, which acts as an Object Relational Mapper (ORM). The ORM abstracts direct SQL queries and allows the application to interact with the SQLite database using Python objects and methods.

All Create, Read, Update, and Delete (CRUD) operations on user and task data are performed through backend routes. Database access is restricted to authenticated users, and all task operations are executed in the context of the currently logged-in user to ensure secure and authorized data access.

3.3 Data Retention Policies

Task Atlas stores user and task data persistently in a local SQLite database. User account information and task records are retained as long as the user account exists. Tasks remain stored until the user explicitly deletes them.

No automatic data expiration or archival mechanism is implemented. Since the application is designed for personal and academic use, data retention is managed directly by the user. Passwords are stored in an encrypted format to ensure data security.

3.4 Data Migration

Task Atlas does not require complex data migration due to its lightweight architecture and local database usage. The SQLite database and required tables are automatically created when the application runs for the first time.

In case of schema changes or database errors, migration can be handled by deleting the existing database file and restarting the application. For future enhancements or production deployment, database migration tools can be integrated if required.

4. Interfaces

Task Atlas provides a simple and user-friendly web-based interface that allows users to interact with the system through a standard web browser. The interface is designed using Jinja2 templates and styled with CSS to provide a modern and clean glassy appearance. The system supports user interaction through clearly defined web routes and forms.

4.1 User Interface (UI)

The application provides the following main interfaces:

Login Interface

- Allows existing users to enter username and password.
- Authenticates users using Flask-Login.
- Redirects successfully authenticated users to the dashboard.

Registration Interface

- Allows new users to create an account.
- Stores user credentials securely in the database.
- Ensures unique username validation.

Dashboard Interface

- Displays the logged-in user's personal task list.
- Allows users to:
    - Add new tasks
    - Mark tasks as completed
    - Delete tasks
- Shows task completion status dynamically.

4.2 Application Programming Interfaces (Routes)

Task Atlas exposes internal application routes for handling user actions:

- `/` – Dashboard (Login Required)
- `/login` – User Login
- `/register` – User Registration
- `/logout` – End User Session
- `/add` – Add Task (POST)
- `/complete/<id>` – Toggle Task Completion
- `/delete/<id>` – Delete Task

These routes handle HTTP requests and return appropriate responses or rendered templates.

4.3 External Interfaces

Currently, Task Atlas does not integrate with any external APIs or third-party services. All functionality is self-contained within the Flask application and local SQLite database.

4.4 Browser Compatibility

The application is accessible through modern web browsers such as Chrome, Edge, and Firefox. Since it uses standard HTML, CSS, and Flask rendering, it does not require additional plugins.

5. State and Session Management

Task Atlas manages user state and session information to ensure secure and personalized access to the application. Since the application is multi-user, session management is essential to maintain user identity and restrict access to personal task data.

User authentication and session handling are implemented using Flask-Login. When a user successfully logs in, a session is created and maintained using a secure session cookie. This session stores the user's authentication state, allowing the system to identify the logged-in user across multiple requests without requiring repeated authentication.

The application follows a server-side session management approach. Protected routes such as the dashboard and task management operations can only be accessed by authenticated users. If an unauthenticated user attempts to access a restricted route, they are automatically redirected to the login page.

User sessions are terminated explicitly when the user logs out or implicitly when the browser session expires. The application uses a secure `SECRET_KEY` to protect session data from tampering. This ensures confidentiality, integrity, and proper session handling throughout the user interaction lifecycle.

6.Caching

Task Atlas does not implement a dedicated server-side caching mechanism in its current design. The application is lightweight and designed for small-scale usage, where direct database access through SQLite provides sufficient performance. Each request made by the user retrieves fresh data from the database to ensure accuracy and consistency of task information.

The system relies on Flask-SQLAlchemy to interact directly with the SQLite database for all Create, Read, Update, and Delete (CRUD) operations. Since the number of concurrent users is expected to be low, database access latency remains minimal, eliminating the immediate need for additional caching layers.

Static resources such as CSS files are automatically cached at the browser level, which improves page loading speed and reduces repeated static file transfers. However, dynamic task data is not cached to ensure that users always view the most up-to-date information.

For future scalability, caching mechanisms such as in-memory caching or external cache stores (e.g., Redis) can be integrated to improve performance under higher traffic conditions. The current architecture prioritizes simplicity, data consistency, and maintainability over complex caching strategies.

7. Non-Functional Requirements

Non-functional requirements define the quality attributes of the Task Atlas system. These requirements ensure that the application is secure, reliable, performant, and easy to maintain.

7.1 Security Aspects

1. Authentication & Authorization

- User authentication implemented using Flask-Login
- Only registered users can access protected routes
- Unauthorized users are redirected to login page

2. Data Protection

- Passwords stored in encrypted format
- Secure session management using SECRET_KEY
- User-specific task isolation to prevent data leakage

3. Session Security

- Server-side session handling
- Sessions terminated on logout
- Protection against unauthorized access to dashboard

4. Deployment Security

- Docker containerization for environment isolation
- CI/CD pipeline ensures secure build process
- Sensitive configuration managed securely

7.2 Performance Aspects

1. Response Time

- Lightweight Flask framework ensures fast request handling
- Minimal dependencies reduce overhead

- Efficient routing structure

2. Database Performance

- SQLite optimized for small-scale applications
- ORM ensures structured and efficient queries
- Quick CRUD operations for tasks

3. Resource Utilization

- Low memory and CPU usage
- Suitable for development and small deployment environments
- Docker ensures consistent resource allocation

4. Scalability Consideration

- Can migrate to production-grade database if needed
- CI/CD supports continuous performance improvements
- Architecture supports future optimization

8. References

- Flask Official Documentation.

  Used for: Developing backend web application and routing logic.

  https://flask.palletsprojects.com/

- Flask-Login Documentation.

  Used for: User authentication and session management implementation.

  https://flask-login.readthedocs.io/

- Flask-SQLAlchemy Documentation.

  Used for: Database modeling and ORM-based data handling.

  https://flask-sqlalchemy.palletsprojects.com/

- Docker Documentation.

  Used for: Containerizing the application and ensuring consistent deployment.

  https://docs.docker.com/

- Git Documentation.

  Used for: Version control and tracking code changes.

  https://git-scm.com/docs