# PawfectMatch

# DESIGN REPORT ON SOFTWARE MAINTAINABILITY

Version 1.2

28/10/2025

# VERSION HISTORY

| Version # | Implemented By | Revision Date | Approved By | Approval Date | Reason |
|---|---|---|---|---|---|
| 1.0 | Chloie | 15/10/2025 | Zi Jian | 25/10/2025 | Design Report on Software Maintainability Template |
| 1.1 | Chong Yao | 20/10/2025 | Zi Jian | 25/10/2025 | Update architectural design pattern |
| 1.2 | Chloie | 28/10/2025 | Zi Jian | 28/10/2025 | Finalized design report on software maintainability |

# TABLE OF CONTENTS

# 1 DESIGN STRATEGIES

The design strategy for *PawfectMatch* is centred on achieving high, long term software maintainability. This goal was established during the planning phase, recognizing that a maintainable system is more robust, scalable, and cost-effective over its lifecycle. Our core strategic decision was to adopt a modular, scalable architecture. This design, which enforces a clean separation of concerns and low coupling, directly supports all forms of software evolution, from corrective bug fixes to adaptive and perfective enhancements. This architectural choice is the foundation upon which all our specific development processes and quality control practices are built.

## 1.1 PLANNING PHASE

During the planning phase, we prioritized designing *PawfectMatch* architecture and code with **scalability** in mind as a core non-functional requirement. This makes our system capable of supporting future growth and functional changes without requiring disruptive overhauls. We anticipated the need to scale the application to handle significantly higher traffic via mechanisms such as caching and horizontal scaling.

To facilitate this plan and ensure high maintainability, we structured the system around a **3 layer architecture** (Presentation, Application, Data), enforcing a clean separation of concerns between the frontend (React) and backend (FastAPI), while enforcing coding standards.

Within the backend, we adopted a **Modular Monolith** pattern. This pattern allows modules to be developed and changed independently. By enforcing strict boundaries and encapsulating layers (Controller, Service, Model, Repository) within each module, we ensure that granular changes can be made to one business function without impacting or necessitating changes in unrelated functions, thereby significantly reducing the risk and cost of maintenance.

## 1.2 DEVELOPMENT PROCESS

Our project follows the **Agile Incremental Development Model**. This approach allows for iterative progress and continuous feedback. A key component of our process is integrated testing, where developers write and execute unit and end-to-end tests as part of our CI/CD pipeline.

Due to the constraints of an academic project, a large-scale external beta test with the target user base was not feasible. To mitigate this, we implemented a formal internal User Acceptance Testing (UAT) process.

This UAT was performed continuously by all project team members, leveraging our cross-functional team structure. Both developers and non-developer roles acted as testers, providing constant feedback on the application's design, functionality, and usability. This approach allowed us to identify and correct defects rapidly.

## 1.3 CORRECTION BY NATURE

Our maintenance strategy is built on a proactive, two-pronged approach that addresses both existing and future faults.

- **Corrective Maintainability**

  Fault detection is a structured activity performed by our QA team and developers. All identified bugs are logged and tracked as formal tickets in our Issue Tracking System (JIRA). This ensures that every fault is documented, prioritized, and resolved in a traceable manner.

- **Preventive Maintainability**

  Preventing future faults is prioritized by improving code quality during development. This is achieved through two primary mechanisms:

  1. **Architectural Design:** Modular monolith pattern enforces high cohesion and low coupling. This atomicity ensures features are isolated, making them easier to test, and preventing errors in one module from cascading into others.
  2. **Automated Testing:** CI/CD pipeline automatically runs unit tests on every pull request to the main branch, allowing the team to detect and fix errors immediately and easily, long before they are integrated into a production build.

## 1.4 ENHANCEMENT BY NATURE

Our design strategies were chosen to facilitate future enhancements, ensuring the application can evolve with new user demands and technologies.

- **Adaptive Maintainability:** Adapt the application to new operational environments

  This was implemented by designing a decoupled 3 tier architecture to separate the React frontend, FastAPI backend, and PostgreSQL database, making it easily adaptable from its current servers to any modern cloud hosting environment with minimal changes.

- **Perfective Maintainability:** Easily add new features and enhancements

  This was a primary driver for our choice of a modular monolith architecture that allows new features to be developed and integrated as independent modules without destabilizing the core. Our phased release plan is built on this capability, allowing us to "perfect" the system over time by adding new, validated features.

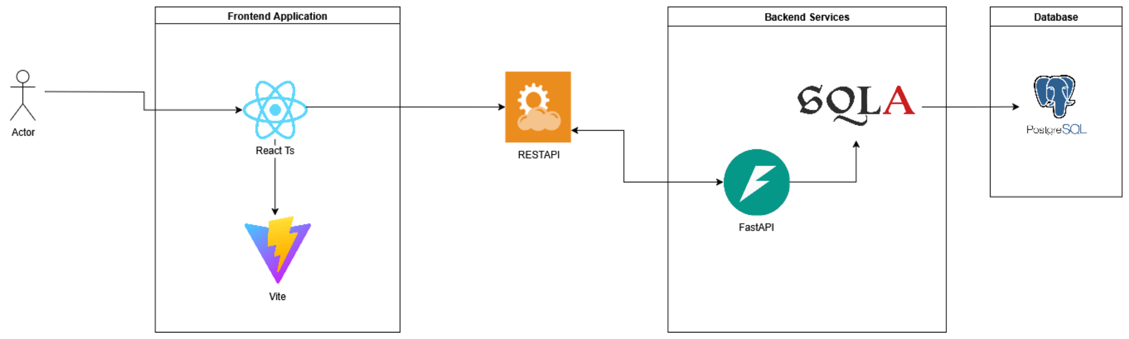
## 1.5 MAINTAINABILITY PRACTICES

To uphold quality in both process and product, we have implemented the following maintainability practices over the course of our project:

- **Readable Code:** Mandatory, automated CI/CD pipeline that enforces the following:
    - **Code Formatting:** Black (backend) and Prettier (frontend)
    - **Static Analysis and Linting:** Ruff and ESLint to detect errors and enforce consistent style
    - **Type Checking:** MyPy for the backend to ensure type safety
- **Version Control:** GitHub as the central software configuration repository, with our branching strategy protecting the integrity of our product.
    - Main branch is protected and always production-ready
    - All new work are done on isolated feature branches
    - Changes are only merged through Pull Requests which require peer review and a passing CI pipeline
- **Standardized Documentation:** All formal project artifacts are treated as configuration items. All documents are stored in the /Documentation repository folder and follow a pre-defined naming convention. Each document contains a Version History table to track all revisions, approvals, and authors, ensuring full traceability.
- **Modularity:** Core of our design, and realized through:
    - **Architectural Separation:** 3 layer architecture that fully decouples the Frontend (React) from the Backend (FastAPI)
    - **Code Isolation:** Modular monolith pattern on the backend, which isolates business logic into distinct modules
    - **Pipeline Independence:** Separate CI/CD pipelines that test and validate the frontend and backend independently, reinforcing their low coupling

## 2 ARCHITECTURAL DESIGN PATTERNS

PawfectMatch follows a client-server model utilizing a 3-tier architecture: Presentation (Frontend), Application (Backend), and Data (Database). The diagram below illustrates the technology stack used to implement the architectural model.



### 2.1 SYSTEM ARCHITECTURE

*PawfectMatch* follows a client-server model using a 3-tier architecture to enforce a complete separation of concerns.

1. **Presentation (Frontend):** Client-side React + TypeScript application responsible for all UI rendering and user interaction
2. **Application (Backend):** Server-side FastAPI + Python application responsible for all business logic, validation and API services
3. **Data (Database):** PostgreSQL database responsible for data persistence and storage

This decoupling ensures that the frontend and backend can be developed, tested, and deployed independently, which is a cornerstone of our maintainability strategy.

### 2.2 FRONTEND ARCHITECTURE

The presentation tier is built using React and TypeScript. Its maintainability is achieved through a component-based architecture. The UI is broken down into small, reusable, and isolated components. This allows developers to update or fix a single component with no risk of side effects, and it ensures a consistent UI and codebase.

### 2.3 BACKEND ARCHITECTURE

Within each module, we enforce a strict layered structure:

- **Controller Layer:** API router (FastAPI APIRouter) defines the routes and uses Pydantic for strict input validation and global exception handlers to ensure this layer remains lean
- **Service Layer:** Contains all business logic, enforcing rules and coordinating function calls to complete a task
- **Repository Layer:** Manages all database queries, primarily using SQLAlchemy to isolate business logic from the data access, allowing queries to be optimized without affecting the service layer
- **Model Layer:** Represents the data models and their table mappings (SQLAlchemy) and database schema versioning (Alembic)

To maintain low coupling between these modules, we enforce dependency injection using FastAPI's "Depends" and Python protocols to define explicit internal and external boundaries. This pattern makes the application highly maintainable and directly supports future evolution into a distributed system if required.

# 3 SOFTWARE CONFIGURATION MANAGEMENT TOOLS

Software Configuration Management (SCM) is the core discipline we use to manage, control, and protect the integrity of all project artifacts, from source code to formal documentation. It is the practical and technical enforcement of the design strategies.

Our SCM strategy is centralized on a single, integrated platform, GitHub. GitHub is not just used as a simple code repository, but instead has its entire toolset leveraged to become our single source of truth, and the engine for our development workflow.

The following sections detail our specific implementation of this SCM system. This integrated system is the mechanism that ensures our project remains traceable, auditable, and highly maintainable.

## 3.1 GITHUB

Our project utilizes GitHub as the central software configuration management platform. It serves as far more than a source code hosting platform, it is the single source of truth for all project artifacts, including code, documentation, and automated workflows.

GitHub was chosen for its robust support for the Git distributed version control system, its seamless integration with our development tools, and its powerful integrated features for issue tracking, peer review, and CI/CD automation. The following sections detail the specific policies and workflows that we have built on to ensure a maintainable, high-quality, and traceable development process.

### 3.1.1 Issue Tracking and Project Management

A hybrid approach to project management was employed, combining JIRA for task tracking and GitHub's integrated tools for code-centric task execution.

JIRA is our primary tool for:

- Tracking tasks, user stories, and bugs
- Monitoring overall sprint progress and resource allocation
- Serving as the central dashboard for the Project Manager

GitHub's integrated Issue Tracking System is then used to manage the developer-level workflow, which allows for:

- Structured reporting and tracking of software bugs directly linked to the codebase
- Assignment of technical tasks and feature enhancements to developers
- Linking Pull Requests directly to issues for automated traceability and task closure

This two-tool strategy allows the Project Manager to maintain a high-level view in JIRA, while developers manage their work directly within the code repository, ensuring all changes are traceable.

### 3.1.2 Repository Structure and Branching Strategy

The project follows a feature-branch workflow to ensure clear organization and strict isolation of all development tasks.

**Repository Structure**

The project repository is structured to separate application code from project documentation, which is crucial for configuration management.

```
3040-TEL2-KK/
├── app/ : Contains all source code (frontend and backend)
├── Backlog/ : Contains project management artifacts
├── Documentation/ : Contains all formal technical documentation
└── Meeting Minutes/ : Contains all team collaboration records
```

**Branching Strategy and Rules**

Our strategy is designed to protect the stability of the main branch.

- **Main branch:** Protected and maintains the production-ready, stable, and fully tested release of the application, where no direct commits are allowed
- **Feature branch:** All new work is done on task-specific branches created from main
- **Pull Request:** Code is only merged into main through a Pull Request which requires:
    1. Mandatory peer review from another developer
    2. Passing CI pipeline to ensure no tests are broken
- **Branch Naming:** To ensure traceability, branch names link directly to our Issue Tracking System using conventions like `ASE-[number]`, `feature/[description]`, or `bugfix/[description]`

### 3.1.3 Automated Quality Assurance and CI/CD

The project employs GitHub Actions for continuous integration and delivery (CI/CD), with pipelines for both backend and frontend systems.

Backend Workflow (backend-check.yml):

- Black for code formatting
- Ruff for linting and static analysis
- MyPy for type checking
- Unit testing for functional validation

Frontend Workflow (frontend-test.yml):

- ESLint for linting
- Prettier for consistent formatting
- End-to-End tests for browser-based validation

This automated CI/CD pipeline is the enforcement mechanism of our branching strategy. No Pull Requests can be merged into the main branch until all checks in these workflows have passed. This automation prevents human error, enforces code consistency, and guarantees that every new feature is verifiably stable before being integrated.

This process is also fundamental to our corrective and preventive maintainability strategies, as it catches faults and style violations at the earliest possible moment.

### 3.1.4 Team Collaboration and Change Tracking

While our task planning and weekly stand-ups are maintained via JIRA and Telegram, GitHub serves as the central hub for all technical collaboration and structured change tracking. GitHub facilitates transparent collaboration and structured change tracking.

**Development Metrics**

- Over 300 commits distributed across 11 branches
- 7 contributors with defined roles and responsibilities
- Feature-driven development integrated through pull requests

**Change Management Practices**

- Standardized commit messages (feat:, fix:, docs:)

- Automated workflows triggered by path-specific changes (app/backend/**, app/frontend/**)
- Protected main branch requiring successful CI/CD and peer reviews
- Complete audit trail with timestamps and author attribution

### 3.1.5 Quality Gates and Code Standards

To ensure all code entering the main branch is stable, consistent, and secure, we enforce automated quality gates that check dependency integrity and code standards. The project applies strict dependency and quality management rules.

**Dependency Management**

- **Backend:** requirements-runtime.txt and requirements-dev.txt
- **Frontend:** package.json with version locks in package-lock.json
- **Database:** Alembic migrations for schema version control

**Quality Enforcement (Makefile excerpt)**

```
format: black app/ && ruff check --fix app/

lint: ruff check app/ && mypy app/

check: black --check app/ && ruff check app/ && mypy app/
```

These dependency and quality checks are not optional, they are executed automatically as our primary quality gate within the CI/CD pipeline on every Pull Request, ensuring no change can be merged until it passes 100% of these standards.

### 3.1.6 Integration and Traceability

The project process is designed to ensure that every change is fully traceable and integrated in a controlled, auditable manner. This is embedded into our daily workflow:

- **Issue-to-Code Traceability:** Direct links between tasks and codes is enforced through requiring branch names and commit messages to reference their corresponding task identifiers from the Issue Tracking System
- **Complete Change History:** GitHub captures an immutable, complete commit history for all files which is essential for maintainability, allowing us to understand why a change was made and enabling safe rollbacks or recovery operations

- **Centralized Artifact Repository:** All project artifacts, including formal documentation, meeting minutes, and technical references, are stored and versioned in the same GitHub repository as the source code which creates a single source of truth for the entire project
- **Controlled Integration:** Our use of protected branches and role-based access controls ensures that code is only integrated after passing a formal Pull Request and peer review

## 3.2 DEVELOPMENT WORKFLOW INTEGRATION

Our GitHub-based Software Configuration Management is a tightly integrated, automated workflow that ensures efficiency and reliability. This process is best illustrated by the lifecycle of a single change:

- **Feature Development:** Developers first creates an isolated feature branch from main to begin work
- **Review and Automated Validation:** To merge the change, the developer must open a Pull Request which triggers two parallel, mandatory quality gates
  - **Peer Review:** Human review is initiated, allowing other team members to check the logic and quality of the change
  - **Automated Validation:** CI/CD pipeline is automatically triggered, running all formatting, linting and automated tests to provide automated validation
- **Integration:** Change can only be integrated into main after both quality gates are passed
- **Deployment and Post-release:** Merge to main creates a new, verified baseline which is then tagged with a semantic version, and its artifacts are deployed with the process being completed by publishing formal Release Notes

This implementation, which enforces both human and automated checks on every change, ensures robust version control, full traceability, consistent code quality, and effective team collaboration throughout the *PawfectMatch* project.