Tab 1

**CS673 Software Engineering**

**Team 1 - Alpha**

**Project Proposal and Planning**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Boya Zhao | Security Configuration SCRUM | *B.Z* | 2025-09-12 |
| Ming Leong Tsui | Team Leader  Admin  Domain knowledge | *ML.T* | 2025-09-12 |
| Sujan Gowda | Backend | *S.G* | 2025-09-12 |
| Xinyu Wang | QA  CI/CD | *X.W* | 2025-09-12 |
| Xiwen Fang | UI | *X.F* | 2025-09-12 |
| Zihan Wang | ML | *Z.W* | 2025-09-12 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Revision history

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| 1 | Lucas | 2025-09-12 | Initial draft |
| 2 | Lucas | 2025-09-19 | Add details |
| 3 | Xinyu | 2025-09-23 | Modify 4&5 |
|  |  |  |  |

[Overview](#_g6igqliy7rm)

[Related Work](#_bf21eadgjj29)

[Proposed High level Requirements](#_rgyo4hi9stmq)

[Management Plan](#_ts358bsdtbcv)

[Objectives and Priorities](#_nxeeppkjxgn4)

[Risk Management (need to be updated constantly)](#_tk7yixobah8p)

[Timeline (need to be updated at the end of each iteration)](#_iksrndohvx29)

[Configuration Management Plan](#_j5uvivmxqcsp)

[Tools](#_dzly5b9kz982)

[Deployment Plan if applicable](#_sd8zu6r3jisd)

[Quality Assurance Plan](#_vra5ptwu59qx)

[Metrics](#_vwjduhc9wuah)

[Code Review Process](#_hx3eaiwb8v3m)

[Testing](#_l9xnpmd6hh0y)

[Defect Management](#_5amsh8h9f0c7)

[References](#_pd9euov6m4du)

[Glossary](#_ty3i2nqffhtc)

# Overview

The Motivation of the project is driven by one of the most persistent challenges is the overwhelming heterogeneity of raw financial statements. Daily property and portfolio transactions arrive in a myriad of formats, each structured differently depending on the source, making it difficult for owners and managers to consolidate information into a reliable view of performance. This fragmentation delays decision-making, increases the risk of error, and raises operational costs. The motivation for this project arises from the recognition that without a standardized, automated system for data organization, asset managers and portfolio owners remain dependent on tedious manual reconciliation processes that do not scale.

The purpose of the proposed system is to create a robust end-to-end data pipeline that transforms raw, inconsistent statements into a unified schema suitable for analysis. By automating ingestion, cleaning, normalization, and enrichment, the pipeline ensures that downstream tasks—statistical modeling, anomaly detection, forecasting, and visualization—are built on a consistent foundation. The system’s ultimate goal is to empower business stakeholders with timely, accurate, and actionable insights into their portfolio performance, cost structures, and cash flows.

The potential users of this software include small-to-medium asset owners managing multiple properties, institutional portfolio managers overseeing large real-estate holdings, financial analysts within asset-management firms, and even regulatory auditors requiring consistent transaction histories. For these stakeholders, the software reduces overhead, minimizes errors, and provides clarity where opacity has long been the norm.

The basic functionality of the proposed system includes:

* Automated ingestion of statements in PDFs with heterogenous layouts.
* Parsing and transformation of transactions into a standardized schema (property, date, category, amount, metadata).
* Data cleaning to handle missing values, inconsistent labels, and duplicate entries.
* Storage in a relational database that supports both structured queries and large-scale reporting.
* Analytical modules for anomaly detection (e.g., unusually high management fees), trend analysis, and reserve adequacy checks.  
  Dashboards and reporting interfaces that communicate portfolio performance in real time.

To deliver these capabilities, the possible technology stack includes the following:

Backend:  
 Flask (Python), Flask-CORS, Flask-SQLAlchemy  
 SQLAlchemy ORM  
 Gunicorn (production)  
 Environment configuration via python-dotenv

Database:  
 SQLite (development, file at AssetManagementAnomalyDetection/instance/asset\_management.db)  
 Azure SQL (production option via pyodbc / DATABASE\_URL fallback logic)

ML/Analytics:  
 scikit-learn (Isolation Forest), pandas, numpy, joblib  
 Matplotlib / Seaborn (visuals)  
 Optional: deterministic KPIs in analysis/business\_analyzer.py

Frontend:  
 React with TypeScript  
 Material UI (@mui/material, icons) and Emotion  
 Chart.js with react-chartjs-2  
 axios  
 Create React App (react-scripts)

OCR utilities:  
 Python: pdf2image, pytesseract, Pillow  
 System: Poppler (pdftotext, pdftoppm), Tesseract OCR

In summary, the project addresses a pressing operational pain point in asset management by transforming unstructured, multi-format statements into structured, actionable intelligence. It combines automation, statistical rigor, and intuitive presentation to serve property owners, portfolio managers, and analysts, with a technical foundation designed for scalability and long-term adaptability.

Related Work

* AWS, Google Cloud: expensive and too complicated to deploy for niche needs. Our project targets a niche demand too small to be addressed by large technological companies, yet ubiquitous in small-to-mid size organizations.
* N8N: AI orchestration tools have human-like context understanding, making them excellent at reading PDFs, but it incurs a per document reading cost, which is negligible for large organizations but significant for small-and-mid sized companies with high financial austerity.

# Proposed High level Requirements

* + Functional Requirements
    - Essential features:
      * Consolidate fragmented performance data (basic ingestion & schema).
      * Visualize portfolio performance (summary statistics, monthly income, PV/NPV).
      * Compare overhead and costs as ratios.
      * Detect anomalous management fees
      * Provide secure access (login, auth)
    - Desirable features
      * Analysis of cost efficiency of each each asset manager
    - Optional Features: locally installed AI that compares portfolio performance against web scrawl market data to make buy/sell recommendations.
  + Nonfunctional Requirements
    - Account log in.
    - The system must be able to handle multiple PDF layouts.
    - summaries and dashboards should update in near real-time.
    - CRUD and modular pipeline structure.
    - CI/DI implemented throughout the development process.
    - Web API p95 latency ≤ 1000 ms for reads and ≤ 100 ms for PDF ingestion endpoints at ≤ 100 RPS.
    - dashboard initial load ≤ 2 s on a 20 Mbps connection; batch import of a 10-page PDF ≤ 20 s end-to-end.
    - API error rate < 1% per iteration.
    - Must handle: (a) 30 properties, (b) 100 statements without configuration changes.

# Management Plan

## Objectives and Priorities

* + - Priority I (highest priority):
      * Callable function: parse fixed-layout PDFs into JSON rows; store in DB with partial CRUD.
      * Web app: show # assets and average monthly income
      * Summary stats from DB
      * Anomaly detection module (basic outputs from DB)
      * Partial CRUD (iteration 1 DB interface)
      * Public web app with summary
    - Priority II:
      * Web app: show # assets, monthly revenue, PV estimates using assumed market data.
      * Extended summary stats including extrapolation.
      * Refined anomaly detection integrated with extrapolated data.
      * Implement at least one security feature (login)
      * Full CRUD across modules
      * Public web app with performance stats & PV
    - Priority III:
      * Handle complex, multi-layout PDFs.
      * Web app: show # assets, monthly revenue, PV estimates refined with live market data.
      * Full analysis including web-crawled market data.
      * Advanced anomaly detection with live market feeds.
      * Account login with full security integration
      * Optimized CRUD with extended schema and market-data integration
      * Public web app with advanced comparisons and buy/sell/hold recommendations.

The following table maps high level requirements into deliverables at each iteration:

## Risk Management

Our group identified several key risks that could impact the success of the project and developed strategies to mitigate them. To address the risk of data leaks, we plan to implement strong security measures such as account logins . In the event of unexpected resignation, we will adopt a flat hierarchy and maintain flexible role boundaries so that each teammate can contribute across functions as needed. To manage the risk of work overload, we will develop the project in piecewise increments and ensure that every contribution delivers value to at least one stakeholder. Finally, to prevent project creep, we will prioritize implementing simpler technologies first to achieve functional milestones, while treating earlier iterations as stable rollback points if needed.

**Risk Management Sheet Link:** [**Risk\_Management**](https://docs.google.com/spreadsheets/d/1-VMJx4FRmm6cEVtfBUus_R8Pt58zwy6xU99j-oEkGh8/edit?gid=0#gid=0)

## Timeline

| Iteration | Functional Requirements(Essential/Disable/Option) | Tasks (Cross requirements tasks) | Estimated/real total person hours |
| --- | --- | --- | --- |
| 1 | **Essential:** Parse fixed-layout PDFs into unified rows; Store parsed data into database; basic API; dashboard shows asset count & monthly income; Pass unit test and CI/CD.  **Desirable:** real-time parsing and dashboard update.  **Optional:** Dockerize dev stack. | Create a PDF reader that reads rental statement with one layout into a table with fixed schema;  Put database online with usable end points;  Create minimal React/Vue dashboard;  Develop and deploy unit & integration tests. | 60 |
| 2 | **Essential:**  Parse PDFs with more than one layouts;  Implement full CRUD API for core entities;  Implement basic analytics (monthly revenue, PV with fixed rates); Implement basic security (account login).  **Desirable:** Role-based access; data validation on upload.  **Optional:** Feature flags for risky changes. | Extend the PDF reader to include more than one layouts with layout detection.  Database endpoints include CRUD operations for analytics.  Implement basic analytics such as monthly revenue per asset, and present value analysis using assumed discounted value.  Implement basic security (account login).  Expand and improve the unit test checklist. | 70 |
| 3 | **Essential:**  External market data ingestion;  anomaly detection v1;  comparative views per manager;  public staging demo; E2E tests.  **Desirable:** Recommendation (buy/sell/hold).  **Optional:** Observability (metrics/traces) dashboards. | Expand the PDF reader to include more layouts.  Implement cost or revenue anomaly detection.  Crawl publicly available market information, e.g., interest rate, and average asset price tomake present value comparison. | 70 |

# Configuration Management Plan

**Source of Truth.** GitHub repository; default branch main.

**Branching.** Short-lived feature branches: feat/<scope>, fix/<scope>, chore/<scope>. Issue/PR must reference a ticket or checklist item.

**Protections.** main is protected: require PR, 1 reviewer, CI green, no direct pushes, linear history (squash-merge).

**Versioning & Tags.** Semantic versioning; every merged release to staging tags vMAJOR.MINOR.PATCH (e.g., v0.3.0). The GitHub Actions build embeds GIT\_SHA and VERSION into the app.

**Release Artifacts.** Docker images named statement-reader:<VERSION> pushed to GitHub Container Registry; build manifests and changelog attached to the GitHub Release.

**Changelog.** CHANGELOG.md updated in each PR using “Keep a Changelog” style with Added/Changed/Fixed.

**Configuration & Secrets.** App reads config from environment variables (.env for local only). Secrets live in GitHub Actions Secrets and staging environment variables; never committed.

**Database Changes.** SQL migration scripts in /migrations with idempotent up/down. PRs that change schema must include a migration and a rollback note.

**Code Quality Gates.** Pre-commit hooks (black/ruff for Python; eslint/prettier for TS). CI runs lint + unit tests + build on every PR.

**Documentation.** README.md covers setup, run, test, deploy, rollback. /docs/ folder holds API schema and data dictionary.

**Backups.** Nightly cron (GitHub Actions scheduled or server cron) dumps DB to a private storage location; filename includes VERSION and date.

**Change Control.** For anything user-visible or schema-changing: open an issue, link it to the PR, include a test plan and rollback plan. Instructor/team lead approves by review.

**Ownership.** Tech lead owns main protection and releases; each module has a rotating “steward” per iteration for quick code reviews.

## Tools

**Version Control:** Git, GitHub, GitHub Desktop

**Project Coordination:** Jira, Discord

**IDEs:** VS Code, PyCharm, SQL Server Management Studio (SSMS)

**AI Tools:** Cursor, ChatGPT, DeepSeek, Codeium

**Integration & API Tools:** Postman, Docker

**CI/CD Tools:** GitHub Actions

**UI / Frontend:** [Vue.js](http://vue.js), Vite

**Backend Framework:** Flask, SQLAlchemy, Alembic(schema migration)

**Code Quality:** pylint+ flake8+ mypy(Python), ESLint(JS/TS)

**Testing:** pytest

**Database:** SQL server, SQLite, Azure

**Knowledge & Notes:** Obsidian

**Monitor:** Prometheus, Grafana

* + Code Commit Guideline and Git Branching Strategy  
    (Please briefly describe criteria for the code commitment and the branching strategy used, e.g. what are the branches to be used, how the pull request will be used etc. Here is an article to give you some basic knowledge about different git branching strategies: <https://www.flagship.io/git-branching-strategies/>
* Main branches contain presentable results and work in progress that does not interfere with presentation.
* Each teammate decides their branching and naming convection. Clarification of the meaning of each branch is done through direct communication.
* Merging conflict is to be resolved by the person sending the pull request and the person whose code is conflicted. This rule implicitly encourages small pull requests so as to avoid conflicts against multiple persons.
* If the merge breaks the code, rollback to the most recent stable release.

## CI/CD & Deployment Plan

Our CI/CD plan protects the main branch and requires all changes to enter via pull requests that pass automated checks before merge. Every PR triggers a pipeline that runs linters, type checks, unit and integration tests, and any schema migrations against a test database, and it blocks on failures.

* Step 1: Static analysis & Code Quality Checks. Runs linters (e.g. pylint, flake8 for Python; ESLint for Vue) and type checks (e.g. mypy) to enforce coding standards and catch basic errors early.
* Step 2: Run Unit Tests. Executes the full unit test suite using pytest to verify individual components function correctly.
* Step 3: Run Integration Tests. Runs integration tests against a test database in an environment that mimics production, validating interactions between components.
* Step 4: Build & Deploy. Upon successful completion of all previous steps, automatic builds deployable artifacts (e.g., Docker images) and deploys them to a staging environment for further validation.

As development progresses, we expand the pipeline by turning every discovered defect into a permanent automated check, so each past mistake becomes a test that prevents recurrence. The QA/CI owner monitors pipeline health, curates the growing suite of checks, and enforces branch protection rules, while developers are responsible for adding or updating tests with their changes and linking PRs to issues. Successful builds automatically produce artifacts and reports (coverage, complexity, and quality metrics) and prepare deployable images; only green builds can be promoted, ensuring that what ships have already met our standards.

# Quality Assurance Plan

## Metrics

(Describe the metrics to be used in the project to measure the quality of your software. Each metric should be measurable and quantifiable. Examples of metrics include product complexity (LOC, # of files, # of classes, # methods, cyclomatic complexity, etc.) , defect rate (# of defect per KLOC), # of test cases, test case pass rate, cost (# of person hours used), # of user stories completed, etc. **The result of these metrics should be reported in the progress report/ iteration summary sheet.**)

| Metric Name | Description | Target | Report cadence |
| --- | --- | --- | --- |
| LOC | Lines of code.watch the growth trend | Sustainable growth, growth ≤ 10%/iteration unless justified in PR | Each iteration |
| No. of classes | Number of classes. proxies for design size/complexity. | Sustainable growth, justify >10%/iteration growth | Each iteration |
| No. of methods | Number of methods. Exponential growth indicates haphazard coding. | Sustainable growth, justify >10%/iteration growth | Each iteration |
| No. of user stories in progress | Number of functional features valuable to a specified end user in progress. | 3 | Each iteration |
| No. of issues opened | Number of technical flaws, e.g., bugs, foreseeable risk, identified. | 3 | Each iteration |
| No. of issues closed | Number of technical flaws, e.g., bugs, foreseeable risk, identified and fixed. | 3 | Each iteration |
| No. of issues remain | Number of technical flaws, e.g., bugs, foreseeable risk, identified and remains. | 3 | Each iteration |
| Total Individual Work Time (h) | Total amount of work hours. | < 50 | Each iteration |

* + Coding Standard

Project scope and rationale:

Adopt a single, enforceable standard to keep code readable, testable, and secure across ingestion, storage, analytics, and presentation. All rules below are checked automatically via linters, formatters, and pre-commit hooks; exceptions must be justified in code review.

Languages and formatting:

Python: Follow PEP 8 with 99-column limit, isort for imports (sections: stdlib, third-party, local), Black for formatting, flake8/ruff for lint. Use type hints everywhere; mypy in strict mode for core modules (ingestion, schema, business logic). One class/file unless there is a clear cohesion reason.

JavaScript/TypeScript: Prefer TypeScript. Prettier for formatting, ESLint with typescript-eslint for lint. No implicit any; strict true. Avoid default exports except React components.

SQL (Postgres): Uppercase keywords, lowercase identifiers, snake\_case for tables/columns, singular table names for entities, plural for collections only when natural. No SELECT \* in committed code. All migrations via a tool (e.g., Alembic). Views/materialized views documented with owning modules.  
  
Naming:  
 Modules and packages: snake\_case. Classes: PascalCase. Functions, methods, variables: snake\_case. Constants: UPPER\_SNAKE\_CASE. Test files mirror module names with \_test suffix. Migration files include sequential id and short purpose tag.  
 Domain entities use stable nouns (property, statement, transaction). Avoid abbreviations except widely recognized (id, url).  
  
Structure and dependencies:  
 Backend packages: ingestion/, parsing/, normalization/, db/, business\_logic/, api/, jobs/, utils/. No cross-layer imports that violate layering (api → business\_logic → db; never db → api). Utilities must be pure or clearly side-effecting and documented.  
 Pin dependencies with exact versions; use a lockfile. Never commit unused packages. Security-patch dependency updates within one iteration.  
  
Error handling and reliability:

No bare except. Catch the narrowest exception; log with context and structured fields (module, function, entity\_id). Business-rule violations raise custom exceptions (e.g., InvalidStatementLayoutError). Retries use exponential backoff with jitter for network/IO. All ingestion steps are idempotent; re-running a batch cannot duplicate rows.  
  
Logging and observability:

Use structured logging (json if possible). Levels: debug for developer detail, info for pipeline milestones, warning for recoverable anomalies, error for failed operations, critical for system-wide impact. Emit metrics for ingestion counts, parse failures, dedupe events, DB latencies, and API response times. Traces propagate request/batch ids.  
  
Database and schema:

Every table has a bigint primary key, created\_at, updated\_at, and source metadata (agent, statement\_date, file\_hash). Foreign keys enforced; cascades explicit. Non-null constraints preferred over nullable + app-level checks. Unique constraints were logical (file\_hash, agent, statement\_date). Indexes justified in migration description and explained in PR. Use transactions for multi-step writes; no autocommit logic.

APIs and contracts:

Backend HTTP API follows REST with nouns and plural resources. GET /properties/{id}/transactions supports pagination; POST/PUT uses request schemas validated by pydantic. Responses include explicit error shapes. Breaking API changes require a deprecation notice and version bump (v1 → v2).

Tests and coverage:

Test pyramid: unit > integration > e2e. Unit tests for every public function with non-trivial logic; aim for ≥90% line coverage in business\_logic and parsing, ≥80% project-wide. Integration tests cover DB migrations, ingestion→normalization→write path, and API endpoints. E2E smoke test for a minimal batch from PDF to dashboard. Randomized tests seed-locked. Flaky tests are forbidden; mark xfail only with a ticket link.

Full Process: Upload a sample PDF -> Trigger backend parsing -> Store data in the database -> Frontend requests data -> Display results on the dashboard. Verify whether the final displayed data matches the original data in the PDF.  
  
Security and secrets:  
No secrets in code or git history. Use .env.example template; real secrets come from the environment or vault. Validate and sanitize all external inputs (PDF metadata, crawled data). Parameterize SQL; forbid string-built queries. Enforce CORS and rate limits at the API edge. Scan dependencies (e.g., safety/pip-audit, npm audit) each iteration; block merge on high-severity issues.  
  
Performance and scalability:

Baseline SLA: API p95 < 300 ms for standard reads; ingestion throughput target stated per iteration. Parse stage must stream large PDFs; avoid loading entire files into memory. Use COPY for bulk inserts. Any O(n²) logic on batch paths must be justified and ticketed for follow-up optimization.

Documentation:

Each module has a top-level README with purpose, public interfaces, invariants, and failure modes. Public functions carry docstrings with args, returns, raises, and examples. Migrations are documented with rationale and rollback strategy. An ADR (architecture decision record) is created for significant choices (storage model, dedupe strategy, market-data source).

Git hygiene, branching, and commits:

Default branch is main; feature branches use feature/short-tag. Commit messages follow:

<type>(scope): short imperative summary

Body explains what and why, links to issue, lists notable side effects or migrations.

Types: feat, fix, refactor, chore, docs, test, perf. One logical change per PR; limit PRs to ~400 lines net diff unless discussed. No direct commits to main; fast-forward merges only after review.

Tooling and automation:

Pre-commit hooks run black, isort, ruff/flake8, mypy, prettier, eslint, and sqlfluff (or equivalent) on staged files. CI pipeline enforces linters, type checks, unit tests, integration tests with a real Postgres service, and coverage thresholds. CI artifacts publish coverage and complexity reports. Main branch is protected; require green CI and up-to-date with main.

Cyclomatic complexity and maintainability:

Keep per-function cyclomatic complexity ≤10 (warn at 11–20; refactor >20). Track with radon/ruff-cyclomatic or Sonar metrics in CI. Functions exceeding thresholds must include a refactor ticket linked in the PR.

Definition of Done (for any feature):

Code merged with passing CI; tests at required coverage; docs updated (module README + API docs); metrics and logs added; feature behind a toggle if relevant; migration applied and rolled back in staging; no P1/P2 open defects; dashboard reflects new data where applicable.

Iteration-level quality metrics you will report:

KLOC, cyclomatic complexity (avg/max by module), defects per KLOC (by severity), # test cases and pass rate, code coverage, build success rate, mean PR review time, # user stories completed vs planned, ingestion throughput, parse failure rate, and p95 API latency.

Exception process:

If a rule must be broken (e.g., hotfix), the PR must include a short “Standards Exception” note describing why, scope, and a follow-up ticket to realign within one iteration.

## Code Review Process

Two AI reviewers using two different models; two for risky or schema-changing PRs. Blockers: failing CI, missing tests, undocumented migrations, new public APIs without docs, secrets exposure. Reviewer checks for readability, test sufficiency, error handling, and adherence to layering.

* Goals: Ensure correctness, security, readability, and consistency before merge.
* Standard PRs: ≥1 human reviewer + 1 AI reviewer
* Risky PRs (schema, auth, parsing core, migrations, security, public API): 2 human reviewers + 2 AI reviewers (require codeowner approval).
* Size limits: Target ≤ 400 net LOC; > 800 LOC must include rationale and review plan.
* Gates (merge blockers)
* CI green: lint, type check, unit+integration tests, coverage thresholds, security scans, migration dry-run.
* Required reviewers approved; no “request changes” unresolved.
* OpenAPI/spec updated when API changes.
* Migrations present + reversible; runbook updated for new jobs.
* No secrets in diff (git-secrets/trufflehog).
* Reviewer checklist
* Design & layering: API → business\_logic → db only; no forbidden imports.
* Correctness: Clear invariants, input validation, error handling, idempotency where needed.
* Tests: Fail-then-pass unit tests; integration/E2E added; coverage impact acceptable.
* Security & privacy: Parametrized SQL; authz checks; PII masked in logs; dependency diffs reviewed.
* Performance: Obvious hot paths; complexity thresholds; queries indexed or justified.
* Observability: Structured logs, metrics, traces; alerts/dashboards updated if needed.
* Docs: Code comments/docstrings; README/ADR/OpenAPI/migration notes updated.
* Rollout: Feature flag or safe-by-default; rollback path described.
* Author responsibilities
* Keep PR small & focused; write clear description (Why/What/How tested/Risks/Rollback).
* Link issues; include screenshots/logs for UI/perf; attach query plans for new indexes.
* Provide migration forward/backward plan; sample data/backfill notes.
* Review SLAs
* First response within 8h (workdays).
* Risky PRs: schedule synchronous review if no decision in 72h.
* Stale PRs (>7d) require rebase or closure.
* Merge strategy
* Protected main; squash-merge by default; rebase-merge allowed for history clarity.
* Auto-close linked issues on merge; delete merged branches

## Testing

* + - * Goals: Ensure software quality, reliability, and performance by implementing a multi-layered testing approach. This strategy integrates both manual and automated testing across unit, integration, and end-to-end (E2E) levels to identify defects early, reduce risks, and deliver a stable product.
      * Key objectives:
      * Early Bug Detection: Find and fix defects as early as possible in the development lifecycle.
      * Code Quality: Maintain a high standard of code quality and prevent regressions.
      * Requirements Validation: Ensure the software meets all specified functional and non-functional requirements.
      * Automation Efficiency: Increase testing efficiency and coverage through strategic automation, allowing manual testers to focus on complex, exploratory testing.
      * Using the testing pyramid model,emphasizing a solid foundation of fast, inexpensive tests with a smaller set of slower, more complex tests.
      * High Volume (Base): Unit Tests - Fast, isolated tests for individual functions/classes.
      * Medium Volume (Middle): Integration & API Tests - Verify interactions between modules and services.
      * Low Volume (Top): UI / End-to-End (E2E) Tests - Validate complete user workflows and system behavior.

## Defect Management

Our team will use GitHub Issues as the central tool for defect management. All defects will be logged, categorized, assigned, and tracked through this platform. Each defect will be labeled with its severity and type, and linked to the corresponding code branch or pull request for traceability.

​​Types of Defects Tracked

* Functional defects: features not working as specified in requirements (e.g., PDF parsing failure, incorrect CRUD behavior).
* Performance defects: slow response times, excessive memory or CPU usage, inefficient queries.
* Usability defects: unclear messages, misleading UI elements, poor user flow in the web app.
* Security defects: unsafe SQL queries, unhandled exceptions, exposed secrets.
* Regression defects: features that worked in previous iterations but break after new code is merged.

Actions and Responsibilities

* Logging: Any team member discovering a defect must immediately create a GitHub Issue with a reproducible description, expected vs. actual behavior, severity, and screenshots/logs if relevant.
* Classification: The QA Lead (Iris) will review each issue, assign severity (critical, major, minor), and ensure labeling consistency.
* Assignment: The Team Lead (Lucas) will assign the defect to the appropriate developer, typically the one most familiar with the affected module.
* Resolution: The assignee fixes the defect in a feature branch, links the pull request to the GitHub Issue, and ensures all relevant test cases are updated.
* Verification: The QA Lead validates the fix by rerunning tests and confirming the defect is closed only after successful reproduction of the expected behavior.
* Reporting: At the end of each iteration, defect statistics (number of open/closed defects, defect rate per KLOC, average resolution time) will be reported in the progress report.
* This plan ensures defects are systematically tracked, accountability is clear, and resolution is integrated into the team’s workflow without being ad hoc.

# AI usage Log

You are allowed and even encouraged to use AI tools to help you generate the project idea, plan it and build it, but you need to clearly describe 1) What tools were used? 2) for what specific tasks and 3) Is it helpful? 4) how did you evaluate or modify AI-generated content? Additionally, you should submit the exported AI chat history as an appendix or share that with the instructor and facilitators.

| Tools | Who | Tasks | helpful | Evaluation/modification | links |
| --- | --- | --- | --- | --- | --- |
| Chat-GPT | Lucas | List all reporting requirements and where to report | Yes | AI output still needs human verification. To improve, the AI criticizes its own work. | [Link](https://chatgpt.com/c/68c44266-3f04-832f-a682-7e31aaf66ca1) |
| Chat-GPT | Lucas | Create TODO list from complex and scattered requirements | Yes | Could use 8n8 to dynamically keep a checklist for the team if funding allows. | [Link](https://chatgpt.com/g/g-p-68a86574cd94819185479aad4b8f70f5-statement-reader/c/68c46748-70dc-8323-99e1-638273360b8c) |
| Chat-GPT | Lucas | Evaluate entire project and codebase against grading criteria | Yes | Massively saves time in checkbox ticking but over reliance might lead to loss of overall intuition over the flow of the project. | [Link](https://chatgpt.com/g/g-p-68a86574cd94819185479aad4b8f70f5/c/68c49887-0078-8331-bbd2-b8fc979d8720%20https://chatgpt.com/c/68cd5cc1-add0-8323-b736-08ddc9a9c60d) |
| Chat-GPT | Lucas | Draft documents | Yes | It uses concepts beyond my knowledge level. I will experiment with its advice incrementally to see if difficulties arise. | [Link](https://chatgpt.com/c/68d5fb09-01a0-8330-b09a-60c7b9284015) |
| Chat-GPT | Lucas | Check feasibility of SPPP | Yes | It correctly points out the risk of scope creep, and recommends a more conservative scope. | [Link](https://chatgpt.com/c/68d55db1-c18c-832d-a5bd-32fe2f273d14) |
| Cursor | Lucas | AI Coding | Mix | A simple “fix it for me” command yields unpredictable results. It achieves runnable codes faster if I first ask it to diagnose the cause of error first and the mix AI recommendation and human judgement to instruct further prompts | Not Available |

# References

Anjali. (n.d.). *Boss-less management: A peek into the flat hierarchy of Valve.* Eaton Business School. [https://ebsedu.org/blog/boss-less-management-a-peek-into-the-flat-hierarchy-of-valve](https://ebsedu.org/blog/boss-less-management-a-peek-into-the-flat-hierarchy-of-valve?utm_source=chatgpt.com) [Eaton Business School](https://ebsedu.org/blog/boss-less-management-a-peek-into-the-flat-hierarchy-of-valve)

Organimi Inc. (n.d.). *Netflix’s organizational structure [Interactive chart].* [https://www.organimi.com/organizational-structures/netflix/](https://www.organimi.com/organizational-structures/netflix/?utm_source=chatgpt.com) [Organimi](https://www.organimi.com/organizational-structures/netflix/)

# Glossary

CS673 — The course code for your Software Engineering class.

SCRUM — An agile framework for managing work in short, iterative sprints with defined roles, ceremonies, and backlogs.

QA — Quality Assurance; the practices that prevent defects and verify the software meets requirements.

CI/CD — Continuous Integration/Continuous Delivery (or Deployment); automated building, testing, and releasing of code changes.

UI — User Interface; the visual and interactive layer through which users use the system.

ML — Machine Learning; algorithms that learn patterns from data to make predictions or decisions.

AWS — Amazon Web Services; a cloud platform providing compute, storage, databases, and other services.

N8N — An open-source workflow automation tool for connecting services and automating tasks.

PDF — Portable Document Format; a fixed-layout document format commonly used for reports and statements.

JSON — JavaScript Object Notation; a lightweight text format for structured data exchange.

CRUD — Create, Read, Update, Delete; the four basic operations for persistent data.

OAuth — Open Authorization; a standard protocol for delegated access (login/auth) without sharing passwords.

IDE — Integrated Development Environment; a developer tool that combines editor, build, debug, and other utilities.

VSCode — Visual Studio Code; a popular, extensible IDE/editor from Microsoft.

SQL — Structured Query Language; the standard language for querying and manipulating relational databases.

API — Application Programming Interface; a defined contract for programs to communicate with each other.

HTTP — Hypertext Transfer Protocol; the application protocol used by the web and many REST APIs.

REST — Representational State Transfer; an architectural style for stateless, resource-oriented web APIs.

CNN — Convolutional Neural Network; a class of ML models especially effective for pattern recognition in grid-like data (e.g., images).

SLA — Service Level Agreement; a stated performance/reliability target, such as latency or uptime.

p95 — The 95th-percentile value of a metric (e.g., latency), meaning 95% of requests are faster than this threshold.

LOC — Lines of Code; a simple size measure of a codebase.

KLOC — Kilo Lines of Code; thousands of lines of code, used to normalize metrics like defect rate.

PR — Pull Request; a proposed code change submitted for review and automated checks before merging.

ADR — Architecture Decision Record; a short document capturing a significant technical decision and its rationale.

CORS — Cross-Origin Resource Sharing; a browser security mechanism controlling cross-site HTTP requests.

E2E — End-to-End; tests or flows that exercise the system from the user entry point through all backend components.

DB — Database; the persistent storage system for application data.

PEP 8 — Python Enhancement Proposal 8; the canonical style guide for writing readable Python code.

AI — Artificial Intelligence; computer systems performing tasks that typically require human intelligence.

IRR — Internal Rate of Return; a finance metric estimating the annualized return of an investment.

PV — Present Value; the current worth of future cash flows discounted at a given rate.