

Finance Analyzer

A smart approach to get your finances under control

Modulework in the module Project 2 (BTW7102)

Studiengang:	B.Sc. Wirtschaftsingenieurwesen
Autor*in:	Aktas Melih, Iseli Marc, Strahm Livio
Betreuer*in:	Antenen Cédric
Auftraggeber*in:	Antenen Cédric
Datum:	05.06.2026

Berner Fachhochschule

Departement
Abteilung

Management Summary

The Finance Analyzer project aims to help people gain better control of and transparency over their personal finances. Many people struggle to understand their spending behaviour due to fragmented data, a lack of structure or missing tools. This is an issue that many people could benefit from addressing. This project provides a simple and efficient solution for importing, analysing and visualising financial transactions.

The system allows users to import bank transaction data in CSV format, categorise expenses automatically, and set budgets for each category. Based on this data, the application generates a clear dashboard with important financial indicators, such as total spending, category distribution, and budget deviations. These features help users identify spending patterns and make better financial decisions.
The project was developed step by step. First, stakeholders were identified and analysed to define clear functional and non-functional requirements. These requirements form the basis of the system and help ensure that the solution meets real user needs. The implementation was improved continuously through testing, feedback, and UX iterations. A key feature of the solution is its modular software architecture. The system is divided into user interface, business logic, and data persistence layers. This structure improves maintainability, scalability, and transparency.

The project's final outcome is a functioning prototype of a personal finance analysis tool that illustrates how financial data can be processed and visualised in a user-friendly format. The system fulfils the defined core requirements and provides a solid foundation for further extensions for example advanced analytics or multi-currency support but the features are endless.
In conclusion, the Finance Analyzer effectively integrates technical implementation with structured project management and requirements engineering. It demonstrates how a clearly defined problem can be transformed into functional software that provides real value to end users.

Table of contents

	A smart approach to get your finances under control	1
	Management Summary	2
Table of contents		3
1	Introduction	5
	1.1 Initial Situation and Problem Statement	5
	1.2 Project Objectives	5
	1.3 Project Scope and Delimitation	5
	1.4 Structure of the Documentation	5
2	Project Approach and Organization	6
	2.1 Methodical Approach	6
	2.2 Team Organization and Roles	6
	2.3 Semester Progress and Iterations	8
	2.4 Tools Used and Team Collaboration	9
	2.5 Version Control and Git Workflow	9
3	Stakeholder Analysis	10
	3.1 Identification of Stakeholders	10
	3.2 Stakeholder Categories	10
	3.2.1 Primary Stakeholders	10
	3.2.2 Secondary Stakeholders	11
	3.3 Stakeholder-Evaluation and Matrix	11
	3.4 Stakeholder Profiles	12
	3.4.1 Stakeholder 1: End User (Private Individual)	12
	3.4.2 Stakeholder 2: Lecturer (Evaluation Authority)	12
	3.4.3 Stakeholder 3: Development Team	13
	3.4.4 Stakeholder 4: Banks (Data Providers)	13
	3.4.5 Stakeholder 5: Data Security / Regulatory Perspective	13
	3.4.6 Stakeholder 6: Market / Potential Future Users	13
	3.5 Conclusions for the Project	14
4	Requirements Engineering	15
	4.1 Approach to Requirements Elicitation	15
	4.2 Core User Flow and Use Cases	15
	4.3 Requirements Prioritization	15
	4.4 Functional Requirements	16
	4.5 Non-functional Requirements	17
	4.6 Optional Requirements	19
	4.7 Acceptance Criteria and Requirements Status	19
5	System Concept and Architecture	22
	5.1 Business Concept and Key Use Cases	22
	5.2 System Architecture Overview	22
	5.3 Layered Architecture and Responsibilities	24
	5.3.1 UI Layer	24
	5.3.2 Application Layer	24
	5.3.3 Feature Layer	25
	5.3.4 Infrastructure Layer	30
	5.3.5 Domain Layer	32
	5.3.6 Utility Layer	32
	5.4 Data Model and Data Flow	34
	5.5 UML Modelling	34
	5.6 Key Architectural and Design Decisions	36
	5.6.1 Retrospective Technical Decision Justifications	37

6	Design and Implementation	39
	6.1 UX and UI Concept	39
	6.1.1 UX Review and Feedback Log	40
	6.2 Application Structure	42
	6.3 Implementation of Core Features	42
	6.4 Technical Challenges and Solutions	45
7	Traceability and Iteration	47
	7.1 Traceability from Requirements to Implementation	47
	7.2 Key Iterations and Improvements	48
8	Testing and Quality Assurance	48
	8.1 Test Strategy	48
	8.2 Automated Test Evidence	48
	8.3 Quality Assurance Conclusion	50
9	Results and Conclusion	50
	9.1 Achieved Functional Scope	50
	9.2 Goal Achievement and Evaluation of the Solution	50
	9.3 Limitations and Possible Future Improvements	51
10	Individual Reflection Livio Strahm	52
	10.1 Personal Contribution to the Project	52
	10.2 Key Learnings	53
	10.3 Challenges Faced	54
	10.4 Critical Evaluation of Own Work	54
11	Individual Reflection Melih Aktas	54
	11.1 Personal Contribution to the Project	54
	11.2 Key Learnings	55
	11.3 Challenges Faced	55
	11.4 Critical Evaluation of Own Work	56
12	Individual Reflection Marc Iseli	57
	12.1 Personal Contribution to the Project	57
	12.2 Key Learnings	57
	12.3 Challenges Faced	58
	12.4 Critical Evaluation of Own Work	58
13	List of Figures	59
14	List of Tables	59
15	Declaration of Originality	60
16	Appendix	60

1 Introduction

1.1 Initial Situation and Problem Statement

The increasing number of digital payments, subscriptions, and everyday transactions has made personal finance management more difficult for many individuals. Although bank statements contain the necessary financial data, they are often not presented in a way that allows users to quickly understand their spending behaviour. As a result, it becomes difficult to identify where money is spent, compare actual spending with personal budgets, and make informed financial decisions. Existing finance tools often provide many features, but they may be too complex, require a high setup effort, or do not offer the level of clarity needed for a fast and understandable personal finance overview. This creates a practical need for a lightweight solution that turns raw transaction data into a clear and useful financial overview.

1.2 Project Objectives

The objective of this project is to develop Finance Analyzer, a web-based application that helps users gain transparency over their personal finances. The system allows users to import bank transaction data from CSV files, process and store the data, automatically categorize transactions, and present the results in a structured dashboard.
In addition, the application supports budget-related analysis and allows users to manually correct transaction categories when needed. These corrections can be used to improve future categorization behaviour through reusable rules. The overall goal of the system is therefore not only to display financial data, but to transform transaction records into an understandable basis for financial insight and decision-making.

1.3 Project Scope and Delimitation

The project focuses on the development of a functional MVP that covers the core use cases of personal finance analysis. These include CSV import, transaction storage, semi-automatic categorization, dashboard-based visualization, budget comparison, and manual category correction. The implemented solution is designed as a structured and lightweight web application that prioritizes clarity, usability, and technical coherence.
The scope of the project is limited to the main functions required to create a reliable financial overview. More advanced extensions were considered during development, but the final implementation remains focused on the features that provide the highest value within the timeframe and objectives of the semester project. The project therefore emphasizes a clean and functional core solution rather than an unnecessarily broad feature set.

1.4 Structure of the Documentation

This project was developed as part of Project 2 (BTW7102) in the Bachelor's degree programme in Industrial Engineering and Management. In addition to the technical implementation, the project documentation follows a structured software engineering approach by addressing stakeholder analysis, requirements engineering, architecture, implementation, testing, and reflection. This corresponds to the project's emphasis on traceability, technical clarity, and continuous documentation throughout development.
The following documentation therefore presents both the resulting software solution and the engineering process behind it. It explains the project approach, identifies the relevant stakeholders, defines the system requirements, describes the architecture and implementation, and summarizes the main results of the project. In this way, the documentation provides a clear and traceable overview of how the Finance Analyzer was developed and how it addresses the initial problem.

2 Project Approach and Organization

2.1 Methodical Approach

The project follows an iterative and agile development approach based on small, manageable work packages. At the beginning of the project, the team defined the initial requirements and used them as the foundation for further development. These requirements were aligned both with the project idea itself and with the expectations and engineering elements required in the module.
The practical workflow is organized in weekly cycles. Once per week, the team holds a structured project meeting in which the current state of the application is reviewed, the next priorities are discussed, and new tasks are defined. These tasks are then entered into the project backlog and assigned to the responsible team members. During the following week, each member works independently on the assigned backlog items until the next coordination meeting.

This approach allows the project to progress in small functional increments rather than through one large implementation phase. New tasks and features are not defined randomly, but are derived from the initial requirements, the current project priorities, and the methodological expectations of the module. In this way, implementation remains focused and traceable throughout the semester.
An important methodological principle of the project is that newly developed functionality must work reliably before it is integrated into the main branch. Features are therefore implemented, reviewed, and tested in their respective working state before they become part of the central code base. This helps maintain a stable development baseline and reduces the risk of introducing incomplete or non-functional changes into the core project.

Overall, this method combines clear initial orientation with iterative refinement. It provides enough structure to keep the project aligned with its requirements, while still allowing flexibility to respond to new insights gained during development and testing.

2.2 Team Organization and Roles



The project is carried out by a team of three members, with responsibilities distributed according to the main needs of the project. Although all members contribute to the overall development of the Finance Analyzer, each person has a primary area of responsibility in order to ensure clarity, efficiency, and accountability within the team.

Livio takes the project lead role. He is primarily responsible for the overall organization of the project, the general product vision, and the definition of the basic code structure. In addition, he is responsible for the structural and methodological parts of the documentation, ensuring that the project is documented in a clear, coherent, and evaluation-oriented way.

Melih is mainly responsible for feature implementation and bug fixing. His focus lies on developing functional parts of the application and improving the technical stability of the system by identifying and resolving errors during implementation and testing.

Marc takes a supporting role within the team. He contributes through smaller code changes and focuses particularly on documenting the code and supporting the overall technical documentation of the project.
This role distribution allows the team to combine organizational coordination, functional development, and documentation work in a structured way. At the same time, it supports a balanced collaboration model in which responsibilities are clearly visible while still contributing to the shared project outcome.

2.3 Semester Progress and Iterations

The progress of the project was planned and monitored using a Gantt chart that compares the planned status with the actual status of the project over time. The planned status served as a time guideline and defined the intended sequence of the main project phases. The actual status was used to document the real development progress and to make deviations visible in a transparent way. This supports a structured and traceable project process, which is an important part of the module's expectations for documentation and technical project work.
The project was divided into several main phases. In the early phase, the focus was placed on clarifying the problem, identifying the relevant stakeholders, and defining the functional and non-functional requirements. Based on this foundation, the team developed the architectural concept and established the initial project structure. The following phase focused on the implementation of the technical basis and the main application features, including CSV import, transaction categorization, budgeting, and dashboard functionality. In the later phase of the project, the focus shifted more strongly toward refinement, bug fixing, testing, and documentation.
Although the project followed a planned timeline, the actual progress did not always match the initial plan exactly. This was expected, because the project was developed iteratively and new insights emerged during implementation and review. The comparison between planned and actual progress therefore does not only show schedule control, but also reflects the project's iterative character. Adjustments were made where necessary, while the overall project direction remained aligned with the defined requirements and objectives.
The Gantt chart is therefore not only a planning tool, but also a documentation instrument. It makes visible how the project evolved over the semester, which phases were completed as intended, and where priorities shifted in response to implementation realities. In this way, it supports a transparent representation of the development process and complements the methodological approach described in the previous section.

Task	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
Project initiation	X	X											
Stakeholder analysis		X											
Requirements engineering		X											
Architecture design			X										
Repo/project setup			X										
CSV import				X									
Database/persistence				X									
Categorization logic				X									
Budget feature					X								
Dashboard/KPIs					X	X							
Filter/export					X	X							
Testing/bug fixing							X	X	X				
UI refinement							X	X	X				
Documentation								X	X	X	X		
Presentation											X	X	X

Table 1: Semester Project Schedule and Iterations

2.4 Tools Used and Team Collaboration

Team collaboration in the project was supported by a small set of practical tools that enabled structured communication, task coordination, and shared documentation. The main communication channel was Microsoft Teams, which was used for ongoing exchange during the week and for coordinating the regular project meetings. This ensured that all team members had access to the same information and could stay aligned on current priorities and implementation progress.
Task coordination was managed through a shared Excel backlog file stored in Teams. During the weekly meetings, newly defined tasks were entered into this backlog, described briefly, and assigned to the responsible team member. In this way, the backlog served as the central operational planning tool of the project. It made responsibilities transparent and allowed the team to track which tasks were open, in progress, or completed. This supported a structured and traceable workflow throughout the semester.

The project documentation was maintained collaboratively in a shared Word document on Teams. This allowed the documentation to be developed continuously alongside the technical implementation rather than only at the end of the project. Such continuous documentation is consistent with the module's expectation that the project should be documented in a structured and traceable way over the course of development.

For the technical implementation itself, the team used a shared code repository with version control. This made it possible to work on different parts of the system in parallel while keeping the project code centrally organised. An important collaboration rule was that new functionality had to work before being integrated into the main branch. This helped maintain a stable project baseline and ensured that collaboration in the codebase remained controlled and reliable.
Overall, the chosen collaboration setup was intentionally simple but effective. Teams supported communication and shared access to project artefacts, the backlog provided operational task management, the shared Word document supported continuous documentation, and version control ensured coordinated implementation. Together, these tools created a working environment that matched the project's iterative and structured development approach.

2.5 Version Control and Git Workflow

Ddf

3 Stakeholder Analysis

3.1 Identification of Stakeholders

Stakeholders are all individuals, groups, or institutions that are affected by the system, influence the project, define requirements, or play a role in its approval and evaluation. In the context of the Finance Analyzer, stakeholder analysis was used to identify the most relevant perspectives that shape both the software itself and the way the project is documented and assessed. This corresponds to the structured project approach expected in the module, where stakeholder identification forms an important basis for requirements and subsequent implementation decisions.
To identify the relevant stakeholders for the Finance Analyzer in a systematic way, the team used a set of guiding questions.
These included:

- Who uses the system?
- Who evaluates or approves the project?
- Who develops and maintains the system?
- Who provides the financial data?
- Who is concerned with legal and security aspects?
- Who may be indirectly affected by the system?

Based on this analysis, the project identified six relevant stakeholder groups: the end user, the lecturer, the development team, banks as data providers, the data protection and security perspective, and the broader market of potential future users. These stakeholders were then analysed with regard to their role, goals, expectations, influence on the project and the risks arising if their perspective is not considered sufficiently.

3.2 Stakeholder Categories

To ensure a structured analysis, the identified stakeholders were grouped into primary and secondary stakeholders. This categorization helps distinguish between stakeholders who directly shape the project and those whose influence is more indirect but still relevant for the solution.
The primary stakeholders are the end user, the lecturer, and the development team. These stakeholders have the strongest direct impact on the project. The end user defines the practical value of the solution and strongly influences usability and functionality requirements. The lecturer acts as the project's academic evaluator and therefore shapes the methodological and documentation-related expectations. The development team is directly responsible for turning the requirements into a working software solution and therefore strongly influences technical feasibility, implementation quality, and project progress.
The secondary stakeholders are banks as data providers, the data protection and security perspective, and the broader market of potential future users. These stakeholders do not control the project directly, but they still influence important aspects of the system. Banks affect the structure and quality of imported data. Data protection and security perspectives influence how sensitive financial information should be handled. Potential future users and the wider market are relevant because they highlight usability, scalability, and long-term relevance beyond the immediate university context.
This categorization supports the project by making clear which stakeholders require close involvement and which stakeholders mainly influence constraints, quality expectations, and future potential.

3.2.1 Primary Stakeholders

The **end user** is the most important practical stakeholder of the Finance Analyzer. The system is intended to provide private individuals with a clearer overview of their personal finances. The end user expects an application that is easy to use, delivers understandable financial insights, and reduces manual effort when reviewing transactions and spending behaviour. If this stakeholder is not considered sufficiently, the system may remain technically functional but fail to deliver real user value.

The **lecturer** is a central stakeholder because the project is developed and evaluated within the framework of the module. The lecturer's role is not only to assess the technical result, but also to evaluate whether the project demonstrates a structured software engineering process, clear requirements, technical reasoning, and traceable documentation. If this perspective is not considered, the project may not satisfy the academic expectations of the module even if the application itself works.
The **development team** is also a primary stakeholder because the team must translate the idea into a feasible and maintainable implementation. The team requires a manageable project scope, a coherent architecture, and enough technical structure to work productively over the semester. If the development perspective is neglected, this may lead to unclear responsibilities, unstable implementation, avoidable technical debt, and delays in delivery.

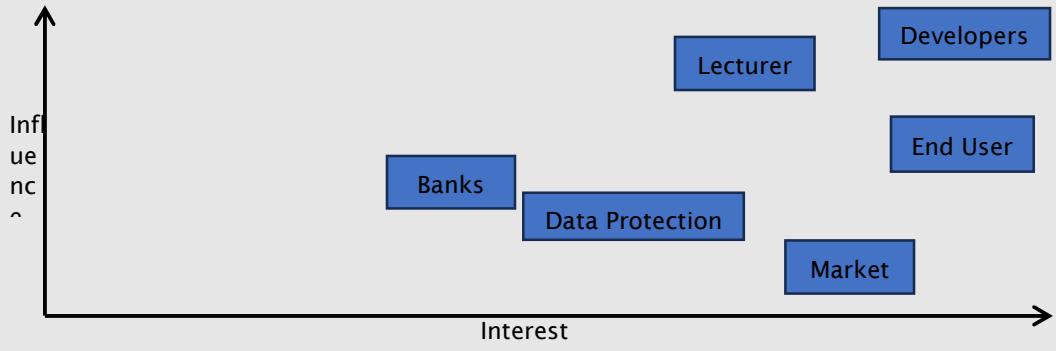
3.2.2 Secondary Stakeholders

Banks are relevant as data providers because the application depends on the structure and quality of exported transaction data. Their influence on the project is indirect, but they shape one of the most important technical use cases: the import and interpretation of bank transaction files. If this stakeholder perspective is neglected, the result may be unreliable CSV import behaviour and inconsistent processing of financial data.
The **data protection and security perspective** is relevant because the application processes personal financial information. Even though this is a student project and not a commercial product, the handling of such data requires care, especially regarding local storage, structured data handling, and minimizing unnecessary exposure of sensitive information. If this perspective is ignored, trust in the system may be reduced and the technical design may fail to reflect important software quality expectations.
The **broader market of potential future users** represents a secondary stakeholder group from a longer-term perspective. While this group does not influence the current project directly, it highlights the importance of usability, extendability, and a clear value proposition. Considering this stakeholder helps the team avoid building a solution that is too narrowly focused or difficult to expand in the future.

3.3 Stakeholder-Evaluation and Matrix

After identifying the relevant stakeholders, they were evaluated according to two criteria: influence on the project and interest in the system. This approach makes it possible to define an appropriate involvement strategy for each stakeholder group. The use of such a matrix is consistent with the stakeholder-oriented documentation approach proposed in the module material.

Stakeholder-Matrix



Stakeholder	Influence	Interest	Strategy
End User	High	High	Closely involve
Lecturer	High	High	Closely involve
Developerteam	High	High	Fully engaged
Banks	Medium	Medium	Keep satisfied

Data Protection / Security Perspective	Medium	High	Keep satisfied
Market	Low	High	Keep informed

Table 2: Stakeholder Matrix1

The matrix shows that the stakeholders with the highest direct relevance for the project are the end user, the lecturer, and the development team. These groups shape the functionality, quality, implementation, and evaluation of the system most strongly. They therefore require the greatest attention throughout the project.
The banks and the data protection / security perspective are relevant mainly because they define important constraints for data import and data handling. Their expectations must be considered carefully, even though they are not actively involved in the project process itself.
The broader market of potential future users has the lowest direct influence on the current project, but a high conceptual interest in usability and long-term value. This stakeholder is therefore relevant mainly from the perspective of scalability, clarity, and future development potential.

3.4 Stakeholder Profiles

3.4.1 Stakeholder 1: End User (Private Individual)

Name / Role:
Private person using the Finance Analyzer to manage personal finances.
Interest in the system:
Financial transparency and control over spending behavior.
Goals:

- Import bank transactions easily
- Automatically categorize expenses
- Define and monitor budgets
- Visualize spending in a clear dashboard

Influence on the project: High
Expectations:

- Simple and intuitive user interface
- Fast performance
- Secure local data storage
- Clear and understandable KPIs and other numbers

Risks if not considered
Low user acceptance, poor usability, system not used in practice.

3.4.2 Stakeholder 2: Lecturer (Evaluation Authority)

Name / Role:
Module supervisor and evaluator of the project.
Interest in the system:
Academic quality, methodological correctness, structured documentation.
Goals:

- Clear stakeholder analysis
- Structured and traceable requirements
- Measurable functional and non-functional requirements
- Professional project documentation

Influence on the project: High
Expectations:

- Compliance with module guidelines
- Logical architecture
- Transparent traceability between requirements and implementation

Risks if not considered:
Project does not meet academic standards and then the project will be lower graded.

3.4.3 Stakeholder 3: Development Team

Name / Role:
Students implementing the software solution.
Interest in the system:
Architecture that is both developed efficiently and maintained effectively.
Goals:

- Clean separation of UI and business logic
- Realistic workload per sprint
- Stable and scalable solution

Influence on the project: High
Expectations:

- Clearly defined requirements
- Manageable scope
- Good tool support (GitLab, version control)

Risks if not considered:
Technical debt, unclear implementation, project delays.

Beyond the primary stakeholder group it also exist the secondary stakeholders were identified whose influence is primarily indirect but still relevant. The relevant part is compliance, data handling, and long-term positioning of the system.

3.4.4 Stakeholder 4: Banks (Data Providers)

Name / Role:
Financial institutions providing transaction data via CSV export.
Interest in the system:
Correct interpretation of exported financial data.
Goals:

- Compatibility with CSV structures
- No misuse of banking information

Influence on the project: Medium
Expectations:

- Proper parsing of transaction data
- Structured data handling

Risks if not considered:
Import errors, inconsistent data processing.

3.4.5 Stakeholder 5: Data Security / Regulatory Perspective

Name / Role:
Data protection authorities and security standards.
Interest in the system:
Protection of sensitive financial data.
Goals:

- Secure local storage
- Encryption at rest
- No unauthorized data access

Influence on the project: Medium
Expectations:

- Implementation of encryption mechanisms
- Compliance with data protection principles

Risks if not considered:
Security vulnerabilities, legal risks, loss of trust.

3.4.6 Stakeholder 6: Market / Potential Future Users

Name / Role:
Broader market of potential users (future scalability perspective).

Interest in the system:
User-friendly and competitive finance management solution.
Goals:

- Modern features
- Good usability
- Competitive functionality

Influence on the project: Low
Expectations:

- Clear value proposition
- Expandable architecture

Risks if not considered:
Limited scalability and innovation potential.

3.5 Conclusions for the Project

The stakeholder analysis shows that the Finance Analyzer is shaped primarily by three perspectives: user value, academic evaluation, and technical feasibility. These are represented by the end user, the lecturer and the development team. Together, they define the practical and technical priorities of the project.
For the project itself, this leads to several clear conclusions. First, the system must remain understandable and easy to use, because practical transparency for the end user is the central product goal. Second, the project must be documented and structured in a traceable way. Third, the implementation must remain manageable and technically coherent, because the development team has to deliver a working solution within the constraints of a semester project.
The secondary stakeholders add further important constraints. Banks require that imported transaction data is handled robustly and consistently. The data protection and security perspective highlights the importance of responsible treatment of financial data. The broader market perspective encourages the team to design the system in a way that remains clear and extendable.
Overall, the stakeholder analysis provides an important foundation for the rest of the project. It creates a traceable link between the initial problem, the expectations of the different stakeholder groups and the requirements and design decisions that follow in the later chapters. That makes it not only a descriptive part of the documentation, but a methodological basis for the project as a whole.

4 Requirements Engineering

4.1 Approach to Requirements Elicitation

The requirements were derived from the stakeholder analysis, the core user workflow, technical constraints of CSV-based banking data, and the evaluation expectations of Project 2. The main user flow was defined as: 7



Input Source	Resulting Requirement Direction
End User	Simple CSV upload, understandable dashboard, category correction
Banks / CSV providers	Robust CSV parsing, validation, format handling
Data security perspective	Local storage, no unnecessary data exposure
Lecturer / module	Traceability, architecture, testing, UX documentation
Development team	Modular structure, maintainable code, realistic MVP

Table 3: Requirement Elicitation Sources

4.2 Core User Flow and Use Cases

Lucid chart flow-diagram mit:

1. User imports CSV
2. System validates and stores transactions
3. System categorizes transactions
4. User reviews unclear transactions
5. User defines budgets
6. Dashboard compares actual spending with budget

4.3 Requirements Prioritization

The requirements were prioritized according to their contribution to a working local-first MVP. The highest priority was given to the complete core user flow that is already reflected in the code: importing bank CSV data, validating and parsing transactions, storing the data locally in SQLite, automatically categorising transactions, allowing manual category correction, defining recurring category budgets, and showing the results in the dashboard. These requirements are essential because the application has no value for the user until transaction data can move reliably from CSV upload to understandable financial insight.

High-priority functional requirements therefore focus on the import -> setup -> review -> budget -> dashboard flow. FR-01 to FR-09 form this core path. They are supported by the implemented FastAPI routes, SQLite persistence, CSV parser, categorisation rules, profile setup, budget service, reporting logic, and server-rendered UI pages. Requirements in this group were treated as must-have functionality for the project demonstration and were marked as fulfilled after being aligned with the current implementation.

Medium-priority requirements were defined as useful extensions that improve review, reuse, and handover value, but are not required for the first successful analysis run. This includes the demo workspace reset, CSV exports, and transaction filtering. The code supports these areas through the environment-controlled reset, report export routes, and transaction filters for month, category, search text, start date, and end date. The documented acceptance criteria were adjusted so they match the implemented behaviour.

Low-priority functional requirements cover supporting guidance and convenience. The help/onboarding view was kept as a low-priority requirement because it improves usability but does not change the financial calculations. Optional requirements such as advanced rule management, custom categories, multi-currency support, dark mode, and advanced recurring subscription detection remain outside the MVP scope unless there is additional time after the core workflow and documentation are stable. Basic merchant-rule learning and recurring spend insights are already covered by the implemented MVP.

The non-functional requirements were prioritized separately because several of them require measurement rather than only code inspection. Local storage and no online registration are fulfilled by the SQLite-based local architecture. Performance, first-time usability, and cross-platform portability are partially fulfilled because the implementation is lightweight and platform-neutral, but formal benchmark tests and platform tests are still needed. Encryption at rest is not fulfilled in the current MVP and is documented as future work.

4.4 Functional Requirements

Functional Requirements (FR)				
ID	Description	Prio	Acceptance Criteria	Status
FR-01	The system must allow the user to create and save a local profile with occupation, financial stage, primary goal, budget style, investing style, savings target, and target months without online registration.	High	User can complete the setup form; profile data is stored locally and persists after restart.	Fulfilled
FR-02	The system must allow importing bank transactions from CSV files via file picker.	High	CSV can be selected and import starts; success or error message is shown.	Fulfilled
FR-03	The system must validate the uploaded CSV structure and show a clear readable error if the file cannot be parsed or required transaction data cannot be detected.	High	Unsupported or empty CSV files are rejected; the user receives a readable import error and existing workspace data is kept.	Fulfilled
FR-04	The system must parse transactions and store them locally in a SQLite database.	High	After import, transaction data remains available after restart.	Fulfilled
FR-05	The system must automatically categorize transactions using local rules, keywords, and learned merchant patterns.	High	Transactions with known keywords or saved merchant patterns are assigned to a matching category automatically; unmatched transactions fall back to Other.	Fulfilled
FR-06	The system must allow the user to manually change a transaction's category.	High	User edits a category; the change persists and affected views use the updated category.	Fulfilled
FR-07	The system must support recurring monthly budget limits per expense category and display them for the selected analysis month.	High	User can save a non-negative monthly limit for a budgetable category; the limit is reused when viewing budget reports for selected months.	Fulfilled

FR-08	The system must compute spending versus budget per category and show remaining or overrun values.	High	For each budgeted category, spent amount, budget limit, delta, and status are computed correctly.	Fulfilled
FR-09	The system must provide a dashboard with at least total spending, top categories, and budget status summary.	High	Dashboard shows the required KPIs for the selected month.	Fulfilled
FR-10	The system must support resetting the local demo workspace on startup when the FINANCE_ANALYZER_DEMO_RESET environment flag is enabled.	Medium	When demo reset is enabled, transactions, budgets, merchant rules, and app settings are cleared on application startup while seeded categories remain available.	Fulfilled
FR-11	The system must provide an onboarding/help view explaining CSV import, categories, budgets, and the dashboard flow.	Low	Help screen is accessible and readable; it contains steps and a sample CSV link.	Fulfilled
FR-12	The system must allow the user to export transaction data and summary reports to CSV files.	Medium	Summary CSV and transaction CSV downloads are available; the summary export includes month, category, budget limit, spending, and difference fields, while the transaction export includes id, date, description, amount, type, and category fields.	Fulfilled
FR-13	The system must allow users to filter the transaction review list by selected month and description search.	Medium	User selects a month or description search term and only matching transactions are displayed in the transaction review list.	Fulfilled

Table 4: Functional Requirements

4.5 Non-functional Requirements

Non-functional Requirements (NFR)				
ID	Description	Prio	Acceptance Criteria	Status
NFR-01	The dashboard should remain responsive for typical local demo datasets by using server-rendered pages, SQLite queries, and lightweight frontend JavaScript.	High	Dashboard route returns successfully for imported transaction data and shows monthly KPIs. A formal 10'000-transaction benchmark is still required for final performance proof.	Partially fulfilled
NFR-02	CSV import should process typical bank export files locally without external services and without deleting existing data when the upload fails.	High	Supported CSV files are parsed and imported locally; failed imports return an error message and keep existing workspace data. A formal 5'000-row import benchmark is still required.	Partially fulfilled

NFR-03	A first-time user should be able to follow the main flow from CSV upload to dashboard review with in-app guidance and without external documentation.	High	Upload page, profile setup, transaction review, budget view, dashboard, and help page are available. A timed usability test is still required to prove the three-minute target.	Partially fulfilled
NFR-04	The application must store financial data locally and avoid online registration or cloud synchronisation in the MVP.	High	Transactions, budgets, merchant rules, profile data, and settings are stored in the local SQLite database; no online account is required.	Fulfilled
NFR-05	Local stored data should be encrypted at rest in a future version, for example with SQLCipher or operating-system file encryption.	Medium	Future verification: database content is not readable as plaintext after encryption is implemented.	Not fulfilled
NFR-06	The application should be portable across Windows 11 and macOS using the documented Python, FastAPI, SQLite, and browser-based stack.	High	The dependency list and local run instructions are platform-neutral. Install and run tests on both Windows 11 and macOS are still required for final proof.	Partially fulfilled

Table 5: Non-functional Requirements

4.6 Optional Requirements

Optional Requirements (OR)			
ID	Description	Prio	Acceptance Criteria
OR-01	The system should provide advanced rule management beyond the existing Remember merchant learning, such as editing rule patterns directly and previewing the impact of a rule before applying it.	Medium	User can deliberately create or edit rule patterns and preview which transactions would be affected before saving.
OR-02	The system must support custom categories (create/rename/delete) with safeguards for already-assigned transactions.	Medium	Categories can be managed; deleting prompts reassignment.
OR-03	Dark mode	Low	Toggle exists
OR-04	Multi-currency support with exchange-rate conversion for reporting.	Low	User sets base currency ; conversions applied
OR-05	Advanced recurring subscription detection beyond the existing recurring merchant and recurring cost insights, including clearer subscription labelling for services such as Netflix or Spotify.	Medium	App flags likely subscriptions separately and lets the user review recurring patterns.

Table 6: Optional Requirements

4.7 Acceptance Criteria and Requirements Status

ID	Requirement Summary	Acceptance Criteria	Status	Comment
FR-01	Local profile setup	User can complete the setup form; profile data is stored locally and persists after restart.	Fulfilled	Implemented through the profile setup route and app_settings storage. The requirement now reflects the actual profile fields used by the application.
FR-02	CSV import via file picker	CSV can be selected and import starts; success or error message is shown.	Fulfilled	Implemented through the upload page and /upload route.
FR-03	CSV structure validation	Unsupported or empty CSV files are rejected; the user receives a readable import error and existing workspace data is kept.	Fulfilled	The parser detects delimiters, headers, dates, amounts, and supported structures. The UI shows a readable error, but it does not list every missing column individually.
FR-04	Local data storage	After import, transaction data remains available after restart.	Fulfilled	Transactions are stored in the local SQLite database.
FR-05	Automatic categorisation	Transactions with known keywords or saved merchant	Fulfilled	Implemented in the categorisation rules module with static and learned rules.

		patterns are assigned to a matching category automatically; unmatched transactions fall back to Other.		
FR-06	Manual category correction	User edits a category; the change persists and affected views use the updated category.	Fulfilled	Implemented on the transaction review page, including optional rule learning.
FR-07	Recurring category budgets	User can save a non-negative monthly limit for a budgetable category; the limit is reused when viewing budget reports for selected months.	Fulfilled	Budgets are stored per category, not as separate records per month. The wording was adjusted to match the implemented recurring monthly budget model.
FR-08	Budget comparison	For each budgeted category, spent amount, budget limit, delta, and status are computed correctly.	Fulfilled	Implemented in the budget service and reports logic.
FR-09	Dashboard KPIs	Dashboard shows the required KPIs for the selected month.	Fulfilled	Implemented on the dashboard page with monthly totals, top spending rows, and budget status metrics.
FR-10	Demo workspace reset	When demo reset is enabled, transactions, budgets, merchant rules, and app settings are cleared on application startup while seeded categories remain available.	Fulfilled	Implemented through the environment-controlled demo reset and a manual /workspace/reset route that requires confirmation.
FR-11	Help and onboarding view	Help screen is accessible and readable; it contains steps and a sample CSV link.	Fulfilled	Implemented through the info/help page and sample CSV route.
FR-12	CSV export	Summary CSV and transaction CSV downloads are available; the summary export includes month, category, budget limit, spending, and	Fulfilled	Implemented through /reports/export/summary.csv and /reports/export/transactions.csv.

		difference fields, while the transaction export includes id, date, description, amount, type, and category fields.		
FR-13	Transaction filtering	User selects a month or description search term and only matching transactions are displayed in the transaction review list.	Fulfilled	Implemented for the transaction review page with month, category, description search, and date range filters. The UI exposes month, category_filter, query, date_from, and date_to controls, and the route applies these filters to the visible transaction list.

Table 7: Acceptance Criteria and Requirements Status

5 System Concept and Architecture

5.1 Business Concept and Key Use Cases

Designed as a personal finance management application, the Finance Analyzer helps users gain transparency and control over their financial behaviour. The system's primary function is to simplify the analysis of bank transactions and transform raw financial data into clear, meaningful insights. Users can import transaction data from CSV files provided by their bank, automatically categorise their expenses, and view their financial situation on a structured dashboard. The system combines automation with user control, reducing manual effort while allowing users to adjust and correct results as needed.
A central concept of the application is its semi-automatic categorisation mechanism. Transactions are assigned to categories based on predefined rules and patterns that have been learned previously. The system improves its categorisation accuracy over time as users correct transactions and store new rules. This strikes a balance between efficiency, transparency and flexibility.
The main use cases of the system reflect the typical workflow of a private user managing personal finances. One of the most important use cases is the import of transaction data. Users upload a CSV file, after which the system processes, validates, and stores the data in a structured format.
Another key use case is transaction categorisation. After importing transactions, they are automatically assigned to categories. Users can manually correct any errors, and these adjustments can be stored as reusable rules for future transactions.
The dashboard is another central feature. It provides users with an overview of their financial situation by displaying key indicators such as total income, expenses and spending per category. Additionally, the system supports budget tracking, enabling users to compare their actual spending with predefined limits.
The business concept focuses on providing a simple, easy-to-understand and efficient tool for analysing personal finances. The system combines structured data processing, automated categorisation and clear visualisation to enable users to better understand their financial behaviour and make more informed decisions.

5.2 System Architecture Overview

The Finance Analyzer is implemented as a modular web application with a hybrid architectural structure. The architecture combines route-based request handling, service-level orchestration, feature modules, and a lightweight layered separation of concerns. This structure was chosen to keep the system understandable, maintainable, and extendable while still allowing iterative development during the semester.

At a high level, the system is divided into several architectural areas: the UI layer, the application and route layer, the service layer, the feature layer, the infrastructure layer, and the utility layer. These areas are not implemented as a strict theoretical architecture model, but as a pragmatic structure that reflects the responsibilities of the actual codebase.

The UI layer is responsible for presentation and user interaction. It contains the Jinja templates, static styling, and client-side JavaScript. The application and route layer handles incoming HTTP requests through FastAPI routes and connects user actions with the internal application logic. The service layer coordinates workflows that involve multiple parts of the system, such as budget handling, context building, authentication, and profile-related behaviour. The feature layer contains the core business functionality, such as CSV import, transaction categorisation, reporting, and rule management. The infrastructure layer manages technical concerns such as SQLite database access, schema initialization, reset logic, and seed data. Finally, the utility layer contains reusable helper functions for tasks such as text normalization, amount conversion, hashing, and parsing support.

Instead of using a separate heavy domain layer, the application represents domain concepts in a lightweight way through service functions, database records, and structured data objects. This keeps the code simple and suitable for the MVP scope, while still separating business decisions from routing, presentation, and database access.

This structure supports traceability between requirements, architecture, implementation, and testing. For example, requirements related to CSV import can be linked to the upload route, the import feature module, the validation logic, the database layer, and the corresponding user interface. Similarly, budget-related requirements can be traced from the budget view through budget services and reporting logic down to the SQLite database.

The following logical architecture overview shows how user interactions move through the system. A user action starts in the UI layer, is handled by FastAPI routes, coordinated by service modules where needed, processed by the relevant feature modules, persisted or retrieved through the infrastructure layer, and finally returned to the UI as rendered output.

Architectural Area	Main Responsibility	Examples in the Project
UI Layer	Presents data and collects user input	Jinja templates, CSS, JavaScript, dashboard, transactions, budgets
Application and Route Layer	Handles HTTP requests and connects user actions to application logic	FastAPI routes such as upload, transactions, budgets, reports
Service Layer	Coordinates workflows and shared application behaviour	Budget service, context service, authentication service, profile service
Feature Layer	Implements core business functionality	CSV import, categorisation, reporting, rule handling
Infrastructure Layer	Manages persistence and technical setup	SQLite database, schema initialization, reset logic, seed logic
Utility Layer	Provides reusable helper functions	Text normalization, amount conversion, hashing, parsing helpers

Table 8: Logical Architecture Overview

5.3 Layered Architecture and Responsibilities

5.3.1 **UI Layer**

The UI layer contains all components related to the front end and is responsible for presenting the application and facilitating user interaction.
It renders HTML views, displays financial summaries and transaction data, and provides input options such as CSV uploads and forms for manually correcting categories. It also manages the visual appearance of the application through styling.
The front end is intentionally lightweight and is implemented using Jinja2 templates, HTML and CSS. This approach is sufficient for the MVP's scope and enables rapid development and straightforward user interface adjustments.
The UI layer does not contain business logic or database access. Its only job is to show data and send user information to the application layer. This clear separation supports the layered architecture overall and improves maintainability.

```
<form action="/upload" method="post" enctype="multipart/form-data" class="home-upload-form" id="home-upload-form">
  <div class="home-upload-picker">
    <div class="file-picker-shell">
      <input type="file" id="csv-file-input" name="file" accept=".csv,text/csv" required />
      <label for="csv-file-input" class="btn btn-ghost file-picker-trigger">Choose file</label>

      <div class="file-picker-meta home-upload-file-meta">
        <strong id="file-picker-name">No file selected</strong>
        <span id="file-picker-help">Use Date, Description, and Amount columns.</span>
      </div>
    </div>
  </div>

  <div class="home-upload-actions">
    <a class="btn btn-ghost" href="/sample-csv">Download sample CSV</a>
    <button type="submit" class="btn btn-primary home-upload-submit" id="upload-submit" disabled>
      Analyze transactions
    </button>
  </div>
</form>
```

Figure 1: CSV Upload Form in the UI Layer

Explanation of Code Snippet
This snippet belongs to the UI layer because it defines the visible CSV upload form that the user interacts with. It allows the user to select a CSV file, shows basic file selection information, offers a sample CSV download, and provides the submit button for starting the import process. The form sends the selected file to the /upload route, but it does not process, validate, categorise, or store the transaction data itself. These tasks are handled by the application, feature, and infrastructure layers. Therefore, the snippet clearly shows the UI layer's responsibility: collecting user input and forwarding it to the backend while remaining separate from business logic and database access.

5.3.2 Application Layer

The application layer is the system's entry point and handles incoming HTTP requests through FastAPI.
The central file of this layer is app.py. It creates the FastAPI application, configures middleware, mounts static files, initializes the database on startup, and registers the different route modules. The actual request handling is implemented in dedicated route files, such as upload.py, profile.py, analytics.py, budgets.py, and transactions.py.

This layer acts as the application's orchestration component. It receives user input through HTTP requests, calls the relevant service or feature functions, and returns either rendered HTML pages or

redirects back to the UI. Instead of implementing detailed business logic itself, the application layer delegates processing tasks to service and feature modules.

This separation keeps routing and control flow independent from the core functionality and makes the system easier to understand, test, and extend.
Typical routes implemented in this layer include:

```
app = FastAPI(title="Finance Analyzer")

app.add_middleware(SessionMiddleware, secret_key=SESSION_SECRET)

app.mount("/static", StaticFiles(directory=str(BASE_DIR / "ui" / "static")), name="static")

app.include_router(core_router)
app.include_router(upload_router)
app.include_router(analytics_router)
app.include_router(budgets_router)
app.include_router(transactions_router)
app.include_router(profile_router)
app.include_router(auth_router)
app.include_router(rules_router)
app.include_router(reports_router)
```

Figure 2: FastAPI Application Setup in the Application Layer

Explanation of Code Snippet
This snippet represents the application and route layer because it shows how the FastAPI application is created and connected to the main parts of the system. The FastAPI instance defines the web application, while the session middleware enables session-based behaviour such as login or user-related state. The static file mounting makes CSS, JavaScript, and other frontend assets available to the UI layer.
The include_router statements register the different route modules of the application. Each router is responsible for a specific functional area, such as the core pages, CSV upload, dashboard analytics, budgets, transactions, profile setup, authentication, rules, and reports. This keeps the request handling organized and prevents all routes from being placed in one large file.
The snippet does not implement the detailed business logic itself. Instead, it connects incoming HTTP requests to the correct route modules, which then call the relevant services, feature functions, or infrastructure components. This supports separation of concerns and makes the application easier to understand, maintain and extend.

5.3.3 Feature Layer

The feature layer contains key parts of the application's core business logic. It implements functional areas such as CSV import, transaction categorisation, reporting and spending analysis.
The folder structure follows a feature-based approach, where each folder represents a specific system functionality. This design simplifies extension and fits well with the iterative development process used during the semester.
The feature layer is accessed by the application and service layers and interacts with the infrastructure layer when data needs to be read or persisted. It does not handle user interface concerns, which keeps presentation, request handling and business processing separated.

```python
def parse_csv_text(csv_text: str) -> list[tuple[str, str, float, str]]:
    if not csv_text.strip():
        raise ValueError("Empty file.")

    lines = [line for line in csv_text.splitlines() if line.strip()]
    sample = "\n".join(lines[:30])
    delimiter = detect_delimiter(sample)
    header_idx, mapping = find_header_row(lines, delimiter)

    cleaned = "\n".join(lines[header_idx:])
    reader = csv.DictReader(cleaned.splitlines(), delimiter=delimiter)

    date_col = mapping["date"]
    amount_col = mapping.get("amount")
    debit_col = mapping.get("debit")
    credit_col = mapping.get("credit")
    desc_col = mapping.get("description")

    rows = []
    for raw_row in reader:
        raw_date = norm_text(raw_row.get(date_col))
        description = norm_text(raw_row.get(desc_col))

        if debit_col or credit_col:
            debit = abs(norm_amount(raw_row.get(debit_col))) if debit_col else 0.0
            credit = norm_amount(raw_row.get(credit_col)) if credit_col else 0.0
            amount = credit - debit
        else:
            amount = norm_amount(raw_row.get(amount_col))

        rows.append((normalize_date(raw_date), description, amount, infer_type(amount)))

    return rows
```

Figure 3: CSV Parsing Logic in the Feature Layer

Explanation of Code Snippet 5.3:
The code snippet shows the CSV import logic in the feature layer. It checks the uploaded CSV content, detects the delimiter and header row, maps the relevant columns and converts each row into structured transaction data. It also normalizes dates and amounts and supports both single amount columns and separate debit and credit columns. This keeps the actual CSV processing separate from the UI and route handling.

5.3.3.1 import_csv

The import_csv module is responsible for importing transaction data from CSV files into the system. It converts raw bank export data into structured transaction records that can be stored, categorised and used for reporting.

The module handles the main import steps, including decoding uploaded files, detecting delimiters, identifying the header row, mapping relevant columns and normalising dates and amounts.

Validation is integrated into the import process. The module checks whether the file is empty, whether the CSV structure can be recognised and whether the transaction rows contain usable data. If the structure is not supported, an error is raised so that the user can be informed.

This keeps CSV-specific logic separated from route handling and UI rendering, which improves maintainability and makes future extensions easier.

```python
def parse_csv_text(csv_text: str) -> list[tuple[str, str, float, str]]:
    if not csv_text.strip():
        raise ValueError("Empty file.")

    lines = [line for line in csv_text.splitlines() if line.strip()]
    sample = "\n".join(lines[:30])
    delimiter = detect_delimiter(sample)
    header_idx, mapping = find_header_row(lines, delimiter)

    cleaned = "\n".join(lines[header_idx:])
    reader = csv.DictReader(cleaned.splitlines(), delimiter=delimiter)

    date_col = mapping["date"]
    amount_col = mapping.get("amount")
    debit_col = mapping.get("debit")
    credit_col = mapping.get("credit")
    desc_col = mapping.get("description")

    rows = []
    for raw_row in reader:
        raw_date = norm_text(raw_row.get(date_col))
        description = norm_text(raw_row.get(desc_col))

        if debit_col or credit_col:
            debit = abs(norm_amount(raw_row.get(debit_col))) if debit_col else 0.0
            credit = norm_amount(raw_row.get(credit_col)) if credit_col else 0.0
            amount = credit - debit
        else:
            amount = norm_amount(raw_row.get(amount_col))

        try:
            date = normalize_date(raw_date)
        except ValueError:
            continue

        rows.append((date, description, amount, infer_type(amount)))

    return rows
```

Figure 4: CSV Parsing Logic in the Feature Layer

Explanation of Code Snippet
This snippet shows the CSV parsing logic in the feature layer. It checks whether the uploaded file is empty, detects the delimiter, identifies the header row and maps the relevant columns. It then normalises dates and amounts and converts each valid CSV row into a structured transaction record. This keeps CSV-specific processing separate from the UI and route handling.

5.3.3.2 categorize

The categorize module assigns categories to transactions based on predefined and learned rules. It uses a two-step, rule-based approach. First, it checks learned merchant patterns stored in the database. These rules help match known transactions more accurately. If no learned rule matches, the module falls back to predefined keyword-based rules that classify transactions based on their description.
After a category has been determined, the corresponding category ID is assigned to the transaction and stored in the database. This category ID is then used for reporting, budgeting, transaction review and dashboard analytics.
This approach enables semi-automatic categorisation. Users can manually correct transaction categories in the transaction review view and optionally save a merchant pattern as a reusable rule. These learned rules can then be applied to future imports, improving the categorisation over time.

```python
def apply_categories() -> int:
    with get_conn() as conn:
        cat_ids = load_category_ids(conn)

        if "Other" not in cat_ids:
            raise ValueError('Category "Other" not found in DB. Run seed first.')

        tx_rows = conn.execute(
            "SELECT id, description FROM transactions WHERE category_id IS NULL"
        ).fetchall()

        updates = []
        merchant_rules = load_merchant_rules(conn)

        for tx_id, desc in tx_rows:
            d = normalize_description(desc)
            matched_category_id = None

            for pattern, category_id in merchant_rules:
                if pattern in d:
                    matched_category_id = category_id
                    break

            if matched_category_id is None:
                cat_name = guess_category(desc)
                matched_category_id = cat_ids.get(cat_name, cat_ids["Other"])

            updates.append((matched_category_id, tx_id))

        if updates:
            conn.executemany(
                "UPDATE transactions SET category_id=? WHERE id=?",
                updates
            )

    return len(updates)
```

Figure 5: Rule-Based Transaction Categorisation

Explanation of Code Snippet
This snippet shows the rule-based categorisation process. It selects transactions without a category, checks learned merchant rules first and then falls back to predefined keyword rules. After a matching category is found, the category ID is written back to the transaction in the database. This keeps the categorisation logic separated from the UI while still allowing future imports to benefit from learned rules.

Reports
The Reports module provides reusable analytical functions used by the dashboard, insights page and other reporting-related views.
It processes transaction data to calculate financial indicators such as income, expenses, net balance, available months, monthly expense trends and budget comparisons. It also supports more detailed analysis through spending snapshots, category averages, recurring costs, anomalies and savings-related indicators.
The module aggregates and structures these results so that they can be used by the application and UI layers for visualisation. This avoids placing analytical SQL logic directly inside the templates. Separating reporting logic into its own module makes the analytical functionality easier to maintain and extend. Future features, such as new KPIs, additional trend analyses or more detailed financial summaries, can be added without changing the UI structure.

```python
def monthly_expense_trend(selected_month: str | None = None) -> list[tuple]:
    with get_conn() as conn:
        rows = conn.execute("""
            SELECT
              substr(date, 1, 7) AS month,
              ABS(COALESCE(SUM(amount), 0)) AS spent
            FROM transactions
            WHERE type = 'expense'
            GROUP BY substr(date, 1, 7)
            ORDER BY month ASC
        """).fetchall()

    if selected_month:
        rows = [row for row in rows if row[0] <= selected_month]

    return rows
```

Figure 6: Monthly Expense Trend Calculation

Explanation of Code Snippet
This snippet shows how the Reports module creates a monthly expense trend. It groups expense transactions by month, calculates the total spending per month and returns the result in a structured format that can be used by dashboard charts or other reporting views.

5.3.4 Infrastructure Layer

The infrastructure layer contains the technical components related to database access, schema initialization and seed data. It supports the rest of the application but does not define the main business workflows.

Its primary responsibility is to provide access to the local SQLite database and ensure that the required database tables exist. The db.py module defines the database path, opens database connections, enables foreign key support, initializes the schema and provides reset functionality for demo workspaces. The seed.py module inserts the predefined transaction categories used by the categorisation logic.

Other parts of the system use the infrastructure layer when they need to persist or retrieve data. This keeps technical database setup and connection handling separated from the UI and route setup. Although SQL queries are still written in several service and feature modules, the infrastructure layer centralizes the database connection and schema management.

This separation improves maintainability because changes to the database location, schema initialization or seed data can be handled in one dedicated area.

```
DB_PATH = Path(__file__).resolve().parents[3] / "data" / "finance.db"

def get_conn():
    conn = sqlite3.connect(DB_PATH)
    conn.execute("PRAGMA foreign_keys = ON")
    return conn

def init_db():
    with get_conn() as conn:
        conn.execute("""
        CREATE TABLE IF NOT EXISTS transactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            date TEXT NOT NULL,
            description TEXT,
            amount REAL NOT NULL,
            type TEXT NOT NULL,
            category_id INTEGER,
            FOREIGN KEY (category_id) REFERENCES categories(id)
        )
        """)

        conn.execute("""
        CREATE TABLE IF NOT EXISTS app_settings (
            key TEXT PRIMARY KEY,
            value TEXT NOT NULL
        )
        """)
```

Figure 7: SQLite Database Connection and Schema Initialization

Explanation of Code Snippet
This snippet shows how the infrastructure layer centralizes access to the local SQLite database and initializes core tables. The database connection is created through get_conn(), while init_db() ensures that required tables such as transactions and app_settings exist before the application is used.

5.3.4.1 db.py

The db.py module establishes the connection to the local SQLite database and initializes the database schema.
It ensures that all necessary tables are created before the application starts processing data. This includes tables for transactions, categories, budgets, merchant rules and application settings. These tables form the basis for storing financial data, category assignments, learned rules, user profile data, login settings and other local workspace state.
The module also enables foreign key constraints to support data consistency between related tables, such as transactions and categories or budgets and categories.
By centralizing database configuration and schema initialization in one module, the application has a uniform and reliable way to access the database. This also allows the system to be ready to use without manual database setup, which simplifies local development and demo usage.

```python
DB_PATH = Path(__file__).resolve().parents[3] / "data" / "finance.db"

def get_conn():
    conn = sqlite3.connect(DB_PATH)
    conn.execute("PRAGMA foreign_keys = ON")
    return conn

def init_db():
    with get_conn() as conn:
        conn.execute("""
        CREATE TABLE IF NOT EXISTS transactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            date TEXT NOT NULL,
            description TEXT,
            amount REAL NOT NULL,
            type TEXT NOT NULL,
            category_id INTEGER,
            FOREIGN KEY (category_id) REFERENCES categories(id)
        )
        """)

        conn.execute("""
        CREATE TABLE IF NOT EXISTS merchant_rules (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            pattern TEXT UNIQUE NOT NULL,
            category_id INTEGER NOT NULL,
            active INTEGER NOT NULL DEFAULT 1,
            created_at TEXT DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (category_id) REFERENCES categories(id)
        )
        """)
```

Figure 8: SQLite Database Connection and Schema Initialization

Explanation of Code Snippet
This snippet shows how db.py defines the SQLite database location, opens database connections with foreign key support and initializes important tables used for transactions and learned merchant rules.

5.3.4.2 repo.py

The repo.py module is currently included as a structural placeholder for a future repository layer.
In the present implementation, most database operations are still performed directly in service, feature and route modules through the shared database connection provided by db.py. Therefore, repo.py does not yet provide active reusable database functions.
Its intended purpose is to support future development by centralizing common database operations, such as adding, updating and retrieving data. Moving repeated SQL operations into this module would reduce duplication and create a clearer separation between business logic and data access.
Although the module is not yet fully implemented, it reflects the intended architectural direction of the system and provides a clear extension point for improving maintainability in later iterations.
No code snippet is shown for repo.py because the module currently acts as a placeholder and does not yet contain implemented repository functions.

5.3.4.3 seed.py

The seed.py module prepares an initial system state.
It inserts predefined categories into the database and prevents duplicates from being created. Based on these categories, either default budget values are inserted or existing values are updated.
This approach guarantees that the application always starts with a consistent and usable dataset. By automatically initialising both categories and budgets, the system can be used immediately, eliminating the need for manual configuration.
Separating this functionality from the feature layer ensures that setup- and persistence-related logic remain isolated from the business logic

5.3.5 Domain Layer

The Finance Analyzer does not use a fully developed domain layer in the strict domain-driven design sense. Instead, it follows a lightweight domain-oriented approach.
Central domain concepts such as transactions, categories, budgets and merchant rules are currently represented mainly through database tables, service functions and structured data objects. Most business rules are implemented in the feature and service modules, for example in CSV import, categorisation, reporting and budget handling.
The files models.py and errors.py in the domain folder currently act as structural placeholders. They show where future domain models and application-specific errors could be introduced if the application grows further.

5.3.6 Utility Layer

The utility layer contains reusable helper functions that support multiple parts of the application. It provides generic functionality required across different modules, particularly for data preprocessing and normalization.
The text.py module includes helper functions for text normalization, value conversion and hashing. For example, norm_amount converts numerical values from CSV files into a standard float format, regardless of whether they use commas or decimal points. The function norm_text standardizes textual input by handling null values and trimming unnecessary whitespace. Additionally, sha256_text provides a hashing mechanism that can be used for generating consistent identifiers or handling sensitive data.
By centralizing these common operations in a dedicated layer, the system avoids code duplication and ensures consistent data handling across the application. This allows feature modules to remain focused on business logic instead of low-level data processing tasks.
Separating utility functions from core logic improves maintainability, readability, reusability and the overall system architecture.

```python
def sha256_text(s: str) -> str:
    return hashlib.sha256(s.encode("utf-8")).hexdigest()

def norm_amount(x) -> float:
    """
    Accepts "12.34" or "12,34" or 12.34
    """
    if x is None:
        return 0.0
    s = str(x).strip().replace("'", "").replace(" ", "")
    s = s.replace(",", ".")
    try:
        return float(s)
    except ValueError:
        return 0.0

def norm_text(x) -> str:
    return "" if x is None else str(x).strip()
```

Figure 9: Utility Functions for Text and Amount Normalization

Explanation of Code Snippet

This snippet shows reusable utility functions for hashing, amount conversion and text normalization. These helpers are used by feature modules such as CSV import to process inconsistent input data in a consistent way.

5.4 Data Model and Data Flow

The Finance Analyzer follows a clear end-to-end data flow from user interaction to persistent storage and back to the user interface. This flow supports traceability, because each step of request processing can be assigned to a specific architectural area.
A typical interaction begins in the frontend. The user performs an action in the interface, such as uploading a CSV file, opening the dashboard, or updating a transaction category. This request is passed to the FastAPI application, where the corresponding route handles the request and triggers the required service or feature logic. The business functionality then processes the request and interacts with the database through the infrastructure layer. Once the required data has been stored or retrieved, the result is returned to the frontend and rendered to the user.
The main application flow can therefore be summarized as follows: user interaction in the frontend leads to request handling in the application layer, which calls the relevant business logic, which in turn interacts with the persistence layer before returning the final output to the user interface.
This flow is especially visible in the CSV import use case, which represents one of the core processes of the application. First, the user uploads a transaction file through the interface. The request is received by the FastAPI application and passed to the import logic. The imported file is then decoded, parsed, validated, and normalized. After that, the transactions are stored in the SQLite database. Once the data is available in persistent storage, the categorization logic assigns categories to the imported transactions. Based on these stored records, the reporting logic calculates key financial values such as totals, category spending, and budget deviations. The resulting data is then displayed in the dashboard and related views.
The underlying data model is centered around the main business entities of the system. These include transactions, categories, budgets, and merchant rules. Transactions represent the imported financial records. Categories structure the analysis of spending behaviour. Budgets define target values against which spending can be compared. Merchant rules support the semi-automatic categorization mechanism by storing learned mappings between transaction descriptions and categories.
This data structure is intentionally simple, because the project focuses on a reliable and understandable MVP. At the same time, it is sufficient to support the system's main functions and provides a clear basis for reporting, budgeting, and iterative extension.

5.5 UML Modelling

For the Finance Analyzer, the team selected a technical UML modelling package consisting of a class diagram and an activity diagram. This decision was made because the project is strongly based on structured financial data and on a clearly defined import and processing workflow.
The class diagram is used to describe the central data objects of the system, such as transactions, categories, budgets, merchant rules, and local app settings. It supports the understanding of the data model and shows how the main concepts of the application are related to each other.
The activity diagram is used to describe the CSV import and processing workflow. This process is one of the most important user flows in the application. It starts with the CSV upload, continues with parsing, validation, normalization, database storage, categorization, and ends with the updated dashboard.
Together, these two diagrams provide a clear technical overview of the Finance Analyzer. The class diagram explains the structural view of the system, while the activity diagram explains the dynamic processing flow. This combination fits the project better than a purely user-centered or organizational diagram package, because the main complexity of the application lies in data handling and processing logic.

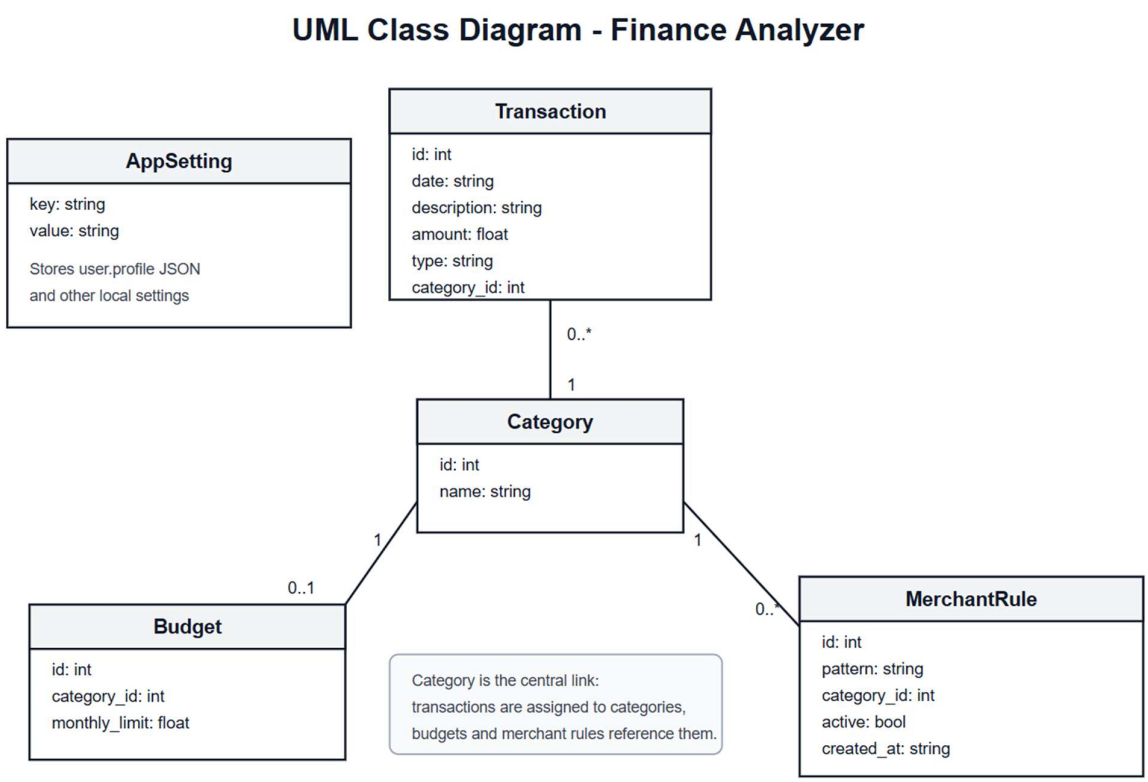


Figure 10: UML Class Diagram of the Finance Analyzer data model

The class diagram shows the central data objects of the Finance Analyzer. Transactions represent imported financial records, while categories are used to structure and analyse spending behaviour. Budgets are linked to categories and define one monthly spending limit per category. Merchant rules are also linked to categories and support the semi-automatic categorization mechanism through saved merchant patterns. User profile data is represented through AppSetting, because the current implementation stores profile information as local settings, for example as user.profile JSON, rather than as a separate profile table. Category remains the central relationship object in the financial data model.

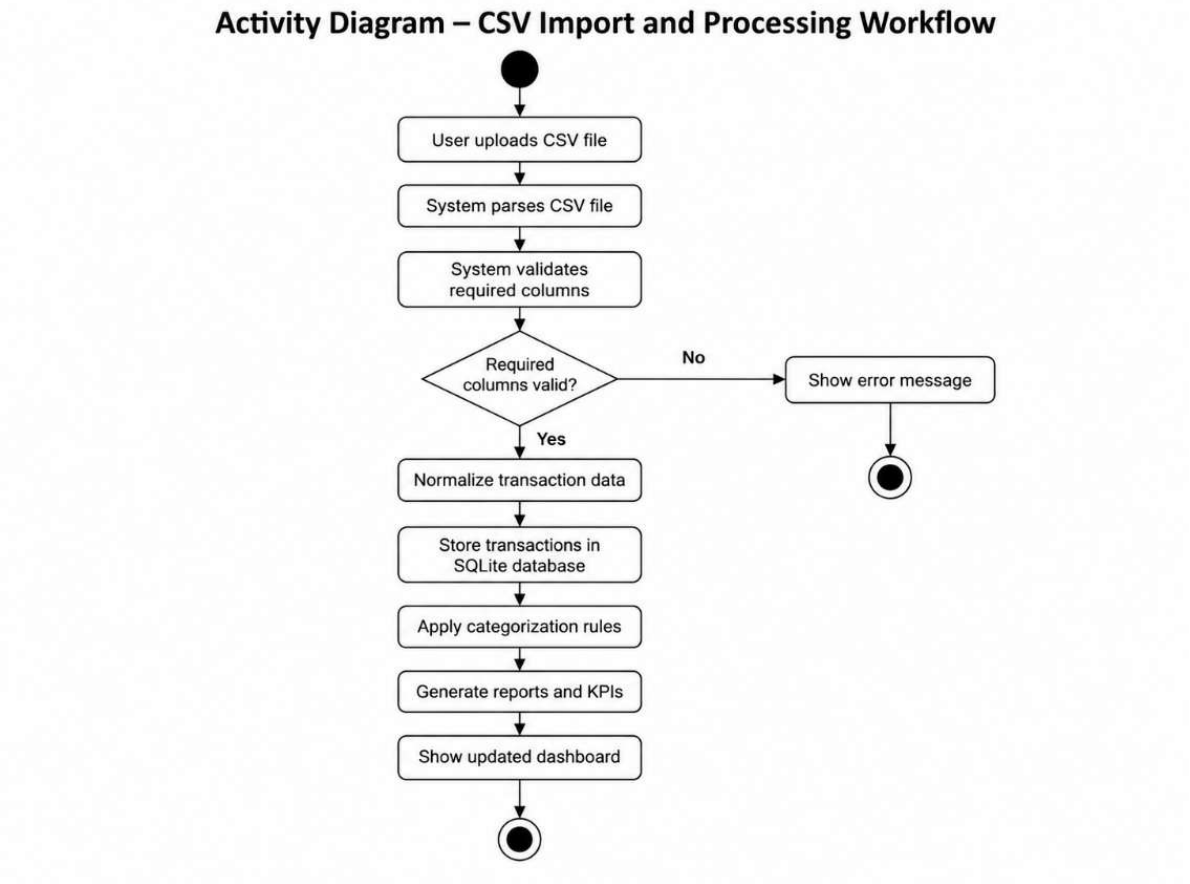


Figure 11: Activity Diagram of the CSV import and processing workflow

The activity diagram describes the main CSV import workflow of the application. The process starts when the user uploads a CSV file. The system then parses the file and validates whether the required columns are available. If the validation fails, an error message is shown to the user. If the validation is successful, the transaction data is normalized, stored in the SQLite database, categorized, and used to generate reports and dashboard KPIs. This diagram therefore shows how one of the most important user actions is transformed into stored and analysed financial data.

5.6 Key Architectural and Design Decisions

One of the most important architectural principles in the Finance Analyzer is separation of concerns. Each part of the application was designed to have a specific responsibility, so that the code remains easier to understand, modify, and test. This principle also improves traceability, because it becomes clearer how requirements are reflected in architecture and implementation.
A second key decision was to use a modular hybrid structure instead of a monolithic implementation. As the project evolved, the application grew beyond a very small prototype and required a more explicit separation of responsibilities. For this reason, request handling, service orchestration, business features, persistence, UI logic, and helper functionality were separated into dedicated modules and folders. This makes the system more scalable and helps avoid tightly coupled code.
Another important design decision was the explicit distinction between frontend responsibilities and backend responsibilities. The frontend handles rendering, navigation, and user input, while the backend processes requests, applies business logic, and manages persistence. This separation was important not only for technical clarity, but also because the project required visible engineering structure in addition to functionality.
A further central decision concerns the semi-automatic categorization mechanism. Instead of relying on a machine-learning-based solution, the project uses a rule-based approach. Previously learned merchant rules are checked first, predefined keyword-based rules are applied second, and unmatched

cases are assigned to the fallback category “Other”. If the user manually corrects a transaction and chooses to remember the change, a new rule is stored and reused in future categorization processes. This solution was selected because it is transparent, understandable, and feasible within the project scope while still creating practical value for the user.
The project also deliberately isolates persistence-related concerns from higher-level business functionality. Database initialization, connection handling, and repository operations are placed in the infrastructure layer so that the rest of the application remains independent from low-level database details. This improves maintainability and makes later changes to the persistence logic easier to control.
Overall, the chosen architecture supports the goals of the project in two ways. From a development perspective, it enables structured and iterative implementation. From a documentation and evaluation perspective, it demonstrates that the project was built with clear technical reasoning, visible software engineering concepts, and a logical connection between architectural choices and implemented functionality.

5.6.1 Retrospective Technical Decision Justifications

In addition to the architectural principles described above, several technical decisions shaped the development of the Finance Analyzer. These decisions were made under typical semester project constraints, including limited development time, different levels of technical experience within the team, the need for a working MVP, and the requirement to document the project in a structured and traceable way. For the most important decisions, the team also considered alternatives, rejected options that did not fit the project scope, and accepted clear trade-offs.

Decision 1: Python and FastAPI as backend technology

The team chose Python with FastAPI as the backend foundation. Python was already familiar from the study programme and allowed the team to implement the business logic quickly. FastAPI was selected because it provides a lightweight but structured framework for handling HTTP requests and building a web-based application.

This decision worked well because it enabled rapid development and kept the technical entry barrier manageable. At the same time, FastAPI offered enough structure to separate routes, services, and feature modules. A challenge was that the team had to understand routing, template rendering, and project structure at the same time. Overall, the decision was suitable for the MVP and would be chosen again for a similar local prototype.

Decision 2: SQLite as local-first database

The Finance Analyzer uses SQLite as its local persistence layer. This decision was made because the application is designed as a local-first MVP and does not require a central server database. SQLite is simple to set up, does not require an external database service, and fits the goal of storing transaction data locally.

This worked well for the project because it reduced setup complexity and made the application easy to run on a student laptop. The downside is that advanced security mechanisms such as encryption at rest are not provided by default. Therefore, encrypted local storage is documented as a possible future improvement. For the current project scope, SQLite was a reasonable and pragmatic decision.

Decision 3: Modular layered structure

The team decided to structure the application into clear areas such as UI, routes, services, features, infrastructure, utilities, and a lightweight domain layer. This was done to avoid a single monolithic script and to keep responsibilities visible in the folder structure.

This decision helped the team manage growing complexity as more features were added. CSV import, categorization, reporting, budgeting, and persistence could be described and maintained more clearly. The challenge was that the architecture evolved during development, and not all layers reached the same level of maturity. Today, the team would keep the modular structure, but define the responsibilities of each layer even earlier.

Decision 4: Rule-based categorization as core logic
The core categorization mechanism is based on rules and learned merchant patterns. This decision was made because rule-based logic is transparent, testable, and easier to explain than a fully machine-learning-based solution.

This worked well because the user can understand why transactions are assigned to categories and can improve the system by saving new rules. The limitation is that rule-based categorization may not handle every ambiguous transaction perfectly. For this reason, the implemented MVP keeps the deterministic rule-based approach as the core logic and does not include machine-learning or AI-based categorization.

Decision 5: Local reporting and insights instead of external services
The team decided to implement reporting and insight generation locally instead of relying on external APIs or cloud services. Financial summaries, budget comparisons, spending snapshots, recurring-cost indicators, anomalies, and savings projections are calculated from the local SQLite data through the reports and analytics modules.

This decision keeps the system more robust and easier to demonstrate, because the core features do not depend on external services. It also supports the local-first privacy concept and makes the results reproducible during tests and presentations. The limitation is that the application does not provide automated bank synchronization, cloud backup, or AI-generated financial advice.

Decision 6: Demo reset mechanism
The application includes reset mechanisms for local demonstrations and repeated team testing. When the FINANCE_ANALYZER_DEMO_RESET environment flag is enabled, dynamic workspace data is cleared on application startup. In addition, the application provides a manual workspace reset route that requires confirmation before deleting transactions, budgets, learned rules, and selected settings.

This worked well because each demo can start from a clean state and team members can test the same workflow repeatedly. The downside is that this behaviour must be clearly documented, because it is useful for a semester project but not suitable for a real production finance application. In a future version, the reset mechanism would be disabled by default or replaced by a controlled demo mode.

Decision 7: Server-rendered UI with Jinja templates
The frontend was implemented using server-rendered HTML with Jinja templates, CSS, and lightweight JavaScript. This decision was made because the project focus was on implementing a working finance workflow rather than building a complex frontend application.

This worked well for the MVP because it reduced technical overhead and made the application easier to understand. It also supported a multi-page structure with dashboard, budgets, transactions, rules, and information pages. A limitation is that highly dynamic frontend behaviour can become harder to implement compared to a modern frontend framework. For the current project, however, Jinja was sufficient and appropriate.

Critical comparison of selected decisions

Decision: Python and FastAPI as backend technology. Alternative considered: Flask or a pure command-line Python tool. Reason for rejection: Flask would also work, but FastAPI provided clearer route structure and validation support; a command-line tool would not demonstrate the intended web-based user workflow. Chosen decision: Python with FastAPI. Trade-off: FastAPI improved structure and testability, but the team had to learn routing, sessions, and template integration.

Decision: SQLite as local-first database. Alternative considered: PostgreSQL or cloud database storage. Reason for rejection: These options would add setup effort, external dependencies, and privacy questions that were not needed for the MVP. Chosen decision: SQLite. Trade-off: The application is easy to run locally and test with isolated databases, but encryption at rest and multi-user synchronization are not included.

Decision: Rule-based categorization. Alternative considered: machine-learning-based categorization. Reason for rejection: Machine learning would require training data, tuning, and more evaluation effort than the semester scope allowed. Chosen decision: deterministic rules plus learned merchant patterns. Trade-off: The logic is transparent and testable, but ambiguous transactions still need manual correction.

Decision: Server-rendered UI with Jinja templates. Alternative considered: a separate React single-page application. Reason for rejection: A separate frontend framework would increase complexity and shift time away from the core finance workflow. Chosen decision: Jinja templates with CSS and lightweight JavaScript. Trade-off: The UI is simpler to maintain for the MVP, but very dynamic frontend behaviour would be harder to implement than in a dedicated frontend framework.

6 Design and Implementation

6.1 UX and UI Concept

The Finance Analyzer's user interface is designed with a clear focus on simplicity, usability, and efficiency. The aim is to help users quickly understand their financial situation and use the system without needing technical expertise.

The application takes a structured, multi-view approach, separating different functionalities into dedicated pages, such as the dashboard, transactions, budgets and rules. This separation reduces complexity and enables users to focus on specific tasks. A consistent layout with reusable templates ensures navigation is implemented in a uniform way across all views

A key feature of the UI design is the clear presentation of financial data. This information is shown in a structured and readable format, for example in tables and summary indicators. This helps users quickly recognise spending patterns and understand their financial behaviour. Key interactions, such as uploading CSV files or correcting transaction categories, are designed to be simple and require little effort.

Special attention was given to providing users with immediate feedback. After performing actions such as importing transactions or updating categories, the interface reflects the changes directly, enabling users to see the results of their interactions immediately. This improves transparency and the overall user experience.

The UI was developed iteratively throughout the project. Based on continuous testing and refinement, the layout and interaction elements were adjusted to improve clarity and simplicity. This approach ensured that the interface evolved alongside the system’s functionality.

The UX and UI concept aims to strike a balance between clarity, functionality and ease of use. The result is an interface that effectively supports users in analysing financial data while remaining simple and intuitive to operate.

To get a first idea of how the UI should look, we created a sketchbook:



We used the sketchbook to build a shared understanding within the team and to decide which UI elements were really necessary.

Some adjustments were still needed during development, but the sketchbook gave the team a common visual direction.

6.1.1 UX Review and Feedback Log

To validate the usability of the Finance Analyzer, a small UX review was conducted with three test users. The goal was not to perform a full professional usability study, but to observe how users interact with the main workflows of the application and to identify concrete usability issues early enough for improvement.

The test users were asked to perform realistic tasks such as uploading a CSV file, checking the dashboard, reviewing transactions, and changing a transaction category. During the review, the team focused on observable user behaviour rather than only asking for general opinions. The feedback was documented in a structured log that connects each observed issue with the design change made afterwards and the resulting improvement.

The following feedback log summarizes the most relevant UX iterations using the structure observed issue, design change, and result.

ID	User	Screen / Feature	Task	Observed issue	User quote	Design change	Result
UX-01	Test user 1, student, basic finance knowledge	Dashboard / Upload	Upload a CSV file and start the analysis	User hesitated before selecting the CSV upload area and was not immediately sure where to start.	“I first looked for the upload button because I was not sure where to begin.”	The upload action was made more prominent and placed clearly in the first workflow area.	The first step became easier to identify, so users could start the analysis with less hesitation.
UX-02	Test user 2, private user, limited software experience	Transactions	Review imported transactions and change one category	User understood the transaction table but was unsure whether the category change was saved automatically.	“I am not sure if this is already saved or if I have to confirm it.”	Direct confirmation feedback was added after saving a category change.	Users receive clearer feedback after an edit, which improves trust in the correction workflow.
UX-03	Test user 3, student, regular online banking user	Budget	Check whether a category is within budget or over budget	User needed a moment to interpret the difference between budget limit, actual spending, and remaining amount.	“I understand the numbers, but the over-budget case could be more visible.”	Clearer status indicators and color states were used for remaining, warning, and over-budget values.	The budget status became easier to scan, especially when a category is near or over its limit.

Table 9: UX Review Feedback Log

The UX review showed that the general structure of the application was understandable, but that some interaction details needed refinement. The most important iterations concerned the visibility of the CSV upload action, direct feedback after saving category changes, and clearer visual indicators in the budget overview. Each issue was translated into a concrete UI change and checked against the intended user result. This made the UX work traceable instead of only descriptive.

6.2 Application Structure

The repository structure of the Finance Analyzer reflects the architectural principles described in the previous chapters and translates them into a concrete implementation. The project code is organised in a way that makes responsibilities visible directly at the code level.
The core application is located in src/finance_analyzer. Within this package, the main areas are domain, features, infrastructure, routes, services, ui and utils. This shows that the implementation follows a modular organisation rather than a monolithic approach.
The domain/ folder contains conceptual definitions and placeholders for domain models and application-specific error types. The features/ folder contains business-oriented modules, including CSV import, categorisation and reporting. The infrastructure/ folder contains modules related to persistence and setup, such as db.py, repo.py and seed.py.
The routes/ folder groups HTTP endpoints by concern, for example uploads, transactions, budgets, reports, rules, authentication, profile handling and analytics. The services/ folder contains coordination logic that supports request processing and cross-feature behaviour, for example in the areas of import handling, budget management, settings, authentication and user profile management.
The ui/ folder contains templates and static files. Its templates are structured across several views, including the dashboard, budgets, transactions, rules, insights, savings, login, profile setup, information pages and the common base layout. This shows that the frontend was designed as a multi-view interface rather than as a single-page upload form.
In addition to the main application code, the repository also contains supporting elements such as tests/, README.md, requirements.txt and execution-related files. These project-level elements support reproducibility, testing and maintainability.
Overall, the repository structure provides a coherent reflection of the application's architectural goals. By separating domain concepts, business features, infrastructure concerns, routing logic, services, user interface assets and utility functions into distinct areas, the project supports separation of concerns and makes responsibilities visible at the code level. This organisation improves readability, facilitates testing and maintenance, and provides a clear basis for future extension. Together with the supporting project-level files, this reinforces that the Finance Analyzer was developed as a structured and reproducible software project rather than as an ad hoc implementation.

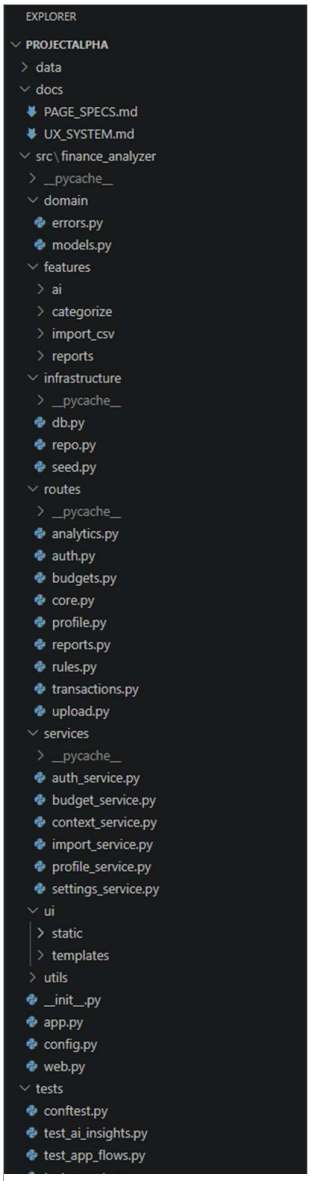


Figure 12: Project Folder Structure of the Finance Analyzer

6.3 Implementation of Core Features

The core functions of the Finance Analyzer are implemented in separate business modules and supporting services. This helps to keep the application modular and easier to understand. Each main feature has its own responsibility in the system, which supports maintainability, traceability and future improvements.
One of the most important features is the CSV import workflow. Users can upload transaction files through the web interface. After the upload, the system processes the data with dedicated import logic. This includes reading the file content, parsing the CSV structure, checking the imported data, normalising values and preparing the transactions for storage in the database. By separating these steps, the import process becomes easier to understand, test and improve later.
A second important feature is transaction categorisation. Imported transactions are assigned to categories with a semi-automatic rule-based system. The system first checks learned merchant rules. If no learned rule matches, predefined keyword rules are used. If no suitable category is found, the transaction is assigned to a fallback category. This reduces manual work for the user and keeps the

logic simple and understandable. Users can also correct categories manually and save rules for future imports.
A third key feature is reporting and dashboard generation. The reporting logic calculates important financial values such as income, expenses, net values, spending by category and budget differences. These results are then shown in the dashboard and other views. This turns raw transaction data into a clear financial overview for the user.
Budget handling is also an important part of the implementation. The system compares actual spending with planned budget limits for each category. The budget view shows whether a category is still within the limit or has exceeded it. This helps users identify critical spending areas and make better financial decisions.
Besides these core functions, the application also includes supporting functions such as profile-related behaviour, rules management and savings-related views. These parts show that the system was developed step by step and that the architecture can support new functions without major restructuring.
Overall, the implementation covers the most important functional requirements of the project. The separation of CSV import, categorisation, reporting, budgeting and supporting services shows that the Finance Analyzer was built with a clear technical structure. This supports maintainability, testing, traceability and future extension.

```python
def norm_amount(x) -> float:
    """
    Accepts "12.34" or "12,34" or 12.34
    """
    if x is None:
        return 0.0
    s = str(x).strip().replace("'", "").replace(" ", "")
    s = s.replace(",", ".")
    try:
        return float(s)
    except ValueError:
        return 0.0
```

Figure 13: Financial Amount Normalisation Function

Explanation of Code Snippet
This snippet shows how financial input values are normalised during the import process. The norm_amount() function converts different amount formats into one standard float format. This is important because CSV files can use different decimal separators, spaces or formatting styles. By using this reusable helper function, the import logic becomes cleaner, more consistent and more reliable.

```python
def budget_report(month: str = "2026-01") -> list[tuple]:
    with get_conn() as conn:
        rows = conn.execute("""
            SELECT
              c.name AS category,
              COALESCE(b.monthly_limit, 0) AS monthly_limit,
              COALESCE(SUM(t.amount), 0) AS spent
            FROM categories c
            LEFT JOIN budgets b ON b.category_id = c.id
            LEFT JOIN transactions t
              ON t.category_id = c.id
             AND t.type = 'expense'
             AND substr(t.date, 1, 7) = ?
            GROUP BY c.id, c.name, COALESCE(b.monthly_limit, 0)
        """, (month,)).fetchall()

    result = []
    for category, limit, spent in rows:
        spent_abs = abs(spent)
        difference = limit - spent_abs

        if spent_abs > 0 or limit > 0 or category == "Other":
            result.append((category, float(limit or 0), spent_abs, difference))

    return result
```

Figure 14: Budget Report Calculation Logic

Explanation of Code Snippet
This snippet shows how the application calculates budget values. The budget_report() function reads the categories, budget limits and transaction amounts from the database. It only uses expenses from the selected month. Then it calculates how much was spent in each category and compares this amount with the planned budget limit. The difference shows whether the user is still within the budget or has spent too much. This helps to show clear budget information in the budget view.

Feature	Main Modules Involved	User Value
CSV import	features/import_csv, routes/upload, infrastructure/db.py	Allows users to upload and process bank transaction data from CSV files.
Transaction categorisation	features/categorize, routes/transactions, rules logic	Reduces manual work and improves transaction organisation.
Reporting and dashboard	features/reports, routes/reports, routes/analytics, dashboard templates	Turns transaction data into financial insights and KPIs.
Budget handling	services/budget_service, routes/budgets, budget templates	Helps users compare spending with planned budget limits.
Profile and setup	routes/profile, profile services, app settings	Supports a personalised local user setup.

Rules management	routes/rules, categorisation logic, merchant rules table	Allows users to reuse learned rules for future imports.

Table 10: Core Feature Implementation Overview

6.4 Technical Challenges and Solutions

One of the main technical challenges of the project was to keep the system manageable while its scope expanded over time. What began as a relatively focused MVP evolved into a broader finance application with multiple views and additional supporting functionality. To prevent the codebase from becoming monolithic, the project structure was progressively modularised. The separation into routes, services, features, infrastructure, UI, domain, and utilities was therefore not only an abstract architectural idea, but also a practical response to increasing complexity.

Another important challenge was the import of financial transaction data. CSV exports can differ in structure, formatting, delimiters, and value representation. The system therefore needed a workflow that could reliably decode, parse, validate, and normalize the input before it entered the database. This challenge was addressed by separating the import process into dedicated responsibilities and supporting it with reusable helper logic. As a result, the import workflow became more robust and easier to maintain.

A further challenge concerned the categorization process. A fully manual solution would have reduced user value, while a more complex machine-learning-based approach was disproportionate for the project's scope and constraints. The chosen solution was therefore a transparent rule-based mechanism with reusable learned patterns. This created a useful balance between automation, simplicity, explainability and implementation feasibility.

Database management was another technically relevant area. The application needed a persistence layer that could initialize the system, handle data consistently, and remain independent from higher-level business logic. This challenge was solved by introducing dedicated infrastructure logic for SQLite connection setup, schema initialization, reset behaviour, and category seeding. This improves maintainability and supports a cleaner separation between technical and functional concerns.

Finally, the project had to remain flexible enough for iterative UI and UX development throughout the semester. Since the interface was refined several times, the frontend needed to support change without requiring constant restructuring of the whole application. The use of reusable templates together with a lightweight web stack made it possible to improve the user interface incrementally while keeping the implementation manageable.

Taken together, these challenges and solutions show that the system was not only implemented functionally, but also engineered with conscious technical decisions. This is important because the quality of the project does not depend only on working features, but also on whether the implementation demonstrates structured and technically justified development.

```python
def init_db():
    with get_conn() as conn:
        conn.execute("""
        CREATE TABLE IF NOT EXISTS categories (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT UNIQUE NOT NULL
        )
        """)

        conn.execute("""
        CREATE TABLE IF NOT EXISTS transactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            date TEXT NOT NULL,
            description TEXT,
            amount REAL NOT NULL,
            type TEXT NOT NULL,
            category_id INTEGER,
            FOREIGN KEY (category_id) REFERENCES categories(id)
        )
        """)
```

Figure 15: Database Schema Initialization in the Infrastructure Layer

Explanation of Code Snippet
This snippet shows how the application initializes the local SQLite database. The init_db() function creates the required tables for categories and transactions if they do not already exist. The foreign key between transactions and categories supports data consistency. This shows that persistence-related logic is handled in the infrastructure layer rather than in the feature or route logic.

Challenge	Solution	Benefit
Growing complexity	Modular structure with routes, services, features, infrastructure, UI, domain and utilities	Clear responsibilities and better maintainability
Different CSV formats	Separate import steps for parsing, validation and normalisation	More robust and adaptable CSV import
Transaction categorisation	Rule-based mechanism with learned patterns	Less manual work and transparent logic
Database management	Dedicated infrastructure modules for connection, schema and seed logic	Reliable persistence and cleaner separation
UI and UX changes	Reusable templates and lightweight web stack	Easier iterative improvements

Table 11: Technical Challenges and Solutions

7 Traceability and Iteration

7.1 Traceability from Requirements to Implementation

To ensure a structured and traceable engineering process, the defined functional requirements are explicitly linked to the system architecture, implementation modules, and user interface components. This allows the project to demonstrate how requirements are translated into concrete technical solutions.

The following table shows the traceability between selected functional requirements, their corresponding implementation modules, UI components, and validation approaches.

Requirement	Implementation (Module)	UI Component	Validation / Testing
FR-02 CSV Import	features/import_csv/service.py, routes/upload.py	Upload Page	tests/test_reports.py, tests/test_app_flows.py
FR-03 CSV Validation	features/import_csv/service.py, features/import_csv/validator.py	Upload Page	tests/test_reports.py
FR-04 Data Storage	infrastructure/db.py, infrastructure/repo.py	System-wide	tests/test_reports.py, tests/test_workspace_reset.py
FR-05 Categorization	features/categorize/rules.py, routes/transactions.py	Transactions View	tests/test_reports.py, tests/test_app_flows.py
FR-06 Manual Category Change	routes/transactions.py, features/categorize/rules.py, services/context_service.py	Transactions Page	tests/test_app_flows.py
FR-08 Budget Calculation	services/budget_service.py, features/reports/service.py, routes/budgets.py, routes/reports.py	Budget View	tests/test_reports.py, tests/test_app_flows.py
FR-09 Dashboard	features/reports/service.py, routes/analytics.py, ui/templates/dashboard.html, ui/static/dashboard.js	ui/templates/dashboard.html	tests/test_reports.py, tests/test_app_flows.py
FR-13 Filtering	routes/transactions.py, ui/templates/transactions.html	Transactions Page	tests/test_app_flows.py

Table 12: Requirements Traceability Matrix

This traceability mapping ensures that the selected requirements are clearly linked to their implementation and validation. It improves transparency and demonstrates the logical connection between requirements, architecture, implementation, and system behavior. Non-functional requirements are validated through performance measurements, usability tests, and system behavior observations, as described in the testing section.

7.2 Key Iterations and Improvements

During the project, the Finance Analyzer was improved through several iterations based on implementation experience, testing, and usability considerations. The goal of these iterations was to make the application more reliable, easier to use, and closer to the intended finance workflow.
One important improvement was the CSV import workflow. At the beginning, the import was more dependent on a specific CSV structure. During development, the system was improved with better delimiter, header, and column detection. This made the import more flexible for different bank exports.
The categorization process was also refined. Instead of relying only on manual category assignment, the system was extended with rule-based categorization and learned merchant patterns. This reduced manual effort and allowed repeated transactions to be categorized more consistently. At the same time, the transaction review page keeps the user in control by allowing manual corrections.
Another important iteration was the improvement of the budget and overview pages. The budget page was developed into a structured view where users can set monthly limits and compare them with actual spending. The overview page was extended with KPIs such as income, expenses, net balance, top spending categories, and budget comparison. This made the application more useful for understanding the current financial situation.
The user workflow was improved by separating the application into clear pages for import, review, overview, budgets, and savings. Navigation buttons were added to guide users to the next relevant step. This makes the application easier to use and supports a more logical step-by-step workflow.
A further improvement was the addition of the savings projection page. This feature extends the application beyond past spending analysis and allows users to estimate future capital development based on starting capital, monthly contribution, time horizon, and different return scenarios.
Overall, these iterations show that the project evolved from a basic finance import tool into a more complete local finance analysis application. The improvements addressed functional gaps, usability issues, and technical risks discovered during development and testing.

8 Testing and Quality Assurance

The quality assurance approach is based on automated pytest tests and focused manual review of the main user workflow. The automated tests are part of the repository and validate the implemented code paths for CSV import, rule-based categorisation, budgets, reports, local insights, profile setup, local login, exports, and workspace reset behaviour.

8.1 Test Strategy

The test suite combines feature-level tests and application-flow tests. Feature-level tests check deterministic business logic such as CSV parsing, amount handling, categorisation, budget calculations, and insight generation. Application-flow tests use the FastAPI test client to verify that important routes, redirects, UI states, form submissions, and persistence effects behave as expected.
The tests are intentionally aligned with the local-first MVP architecture. They do not depend on external banking APIs, cloud services, or third-party authentication providers, because the implemented application stores and processes financial data locally with SQLite.

8.2 Automated Test Evidence

The test suite was executed with the command python -m pytest -q. The result was 33 passed, 20 warnings. The warnings are deprecation warnings from the FastAPI and httpx test environment. They do not affect the functional correctness of the application. The tests use an isolated SQLite test database. In tests/conftest.py, the database path is changed to a temporary finance-test.db file for the test run, and the database is seeded before the tests start. This keeps the real local workspace separate from the test data and makes the test results reproducible. The test files cover the main implemented areas of the project: CSV import and validation, budget calculations, reports, transaction filtering, manual category correction, learned merchant rules, profile setup, local login, workspace reset, insights, and central FastAPI route flows.

Test Area	Code Evidence	Test Evidence	Purpose
CSV import and validation	features/import_csv/service.py features/import_csv/validator.py routes/upload.py	tests/test_reports.py: test_import_csv_supports_ubs_debit_credit_export test_import_csv_supports_tab_separated_export test_import_csv_supports_european_thousands_amounts test_import_csv_validation_lists_missing_required_columns	Verifies that different bank-export formats can be parsed, normalized, validated, and rejected safely when required columns are missing.
Categorisation and learned rules	features/categorize/rules.py routes/transactions.py infrastructure/db.py merchant_rules table	tests/test_app_flows.py: test_set_category_updates_transaction_and_rule_learning test_undo_last_category_change_restores_previous_value test_rule_toggle_disables_rule tests/test_reports.py: test_keyword_matching_does_not_categorize_bargeldbezug_as_restaurant	Checks automatic category assignment, manual correction, learned merchant rules, undo behaviour, and a regression case for incorrect keyword matching.
Budget calculation and reports	services/budget_service.py features/reports/service.py routes/budgets.py routes/reports.py	tests/test_reports.py: test_budget_report_handles_limit_spent_and_zero_limit test_budget_metrics_add_savings_adjustment_and_balance_to_zero tests/test_app_flows.py: test_budget_update_success_preserves_month_and_message test_budget_update_rejects_negative_values test_export_summary_csv_returns_csv	Validates budget limits, spent values, remaining differences, input validation for budget updates, budget difference logic, and CSV report export.
Application routes and user flows	app.py routes/core.py routes/profile.py routes/auth.py ui/templates/*.html	tests/test_app_flows.py: test_clean_workspace_hides_seeded_defaults_until_csv_upload test_profile_setup_saves_personalization_and_generates_budget test_local_login_can_be_enabled_and_then_requires_unlock test_local_login_setup_cannot_be_overwritten_without_unlock test_failed_upload_keeps_existing_workspace_data	Verifies central user-facing flows such as clean initial state, profile setup, local login behaviour, failed upload handling, and route responses.
Insights and savings analysis	features/reports/snapshot.py routes/analytics.py ui/templates/insights.html ui/templates/savings.html	tests/test_insights.py: test_spending_snapshot_calculates_local_averages test_insights_page_renders_local_analysis tests/test_app_flows.py: test_clean_workspace_hides_seeded_defaults_until_csv_upload	Checks local spending averages, savings-plan calculations, insights rendering, and the savings projection states before transaction data is available.
Workspace reset and local persistence	infrastructure/db.py services/context_service.py routes/core.py	tests/test_workspace_reset.py: test_reset_demo_workspace_clears_dynamic_project_state tests/test_app_flows.py: test_workspace_reset_route_requires_confirmation_and_preserves_profile	Confirms that demo reset and manual workspace reset clear dynamic financial data while preserving the expected local profile state where required.

Table 13: Automated Test Evidence

8.3 Quality Assurance Conclusion

The automated tests provide evidence that the core MVP workflow is not only implemented but also technically verified against the actual repository structure. The tested areas cover the most important functional requirements: importing bank transactions, validating input files, categorising transactions, correcting categories, calculating budgets, rendering local insights, exporting reports, and resetting the local workspace for demonstrations.
The test suite does not replace all possible manual checks. Visual layout, subjective usability, and uncommon bank-specific CSV formats still require manual review. However, the existing pytest suite provides a reproducible baseline for regression testing and supports the traceability between requirements, implementation, and validation.

9 Results and Conclusion

9.1 Achieved Functional Scope

The implemented application achieved the core functional scope defined for the Finance Analyzer project. The system supports a complete local personal finance workflow, starting with the import of CSV bank statements and continuing through transaction categorization, budget planning, financial overview, and savings projection.
One of the central implemented features is the CSV upload and import process. Users can select and upload a bank transaction file directly in the application. The system imports transaction dates, descriptions, income, and expenses and stores the processed data in a local SQLite database. After a successful import, the application provides immediate feedback and displays the imported financial values for the selected period.

The application also includes a transaction review workflow. Imported payments are automatically categorized where possible. If transactions require manual review, the user can assign or correct categories in the review section. This supports data quality and ensures that the financial overview is based on meaningful category assignments.
A further core feature is the spending overview. The overview page displays important financial key figures such as income, expenses, net balance, assigned budgets, actual spending, and budget difference for the selected month. In addition, it shows the largest spending categories and provides a visual comparison between planned budgets and actual spending. Savings projection is available through a separate savings page link. This allows users to quickly understand their current financial situation and identify relevant spending patterns.
Budget planning was also implemented as part of the MVP. Users can define monthly budget limits for individual categories. The budget page compares these limits with actual spending and shows the difference between planned and real values. This helps users identify categories without a budget, recognize overspending, and manage their financial planning more consciously.
The application also includes a savings projection feature. Based on the current net balance, monthly contribution, investment horizon, and different return scenarios, the system calculates projected capital development over time. This extends the application beyond short-term spending analysis and supports long-term financial planning.

Overall, the delivered scope covers the intended end-to-end user journey: importing financial data, reviewing and improving transaction categories, setting budgets, analyzing the current financial situation, and planning future savings. The application remains local-first, meaning sensitive financial data is stored and processed on the user's own machine rather than in an external cloud service.

9.2 Goal Achievement and Evaluation of the Solution

The main goal of the Finance Analyzer was to create a functional MVP that helps users gain a clearer understanding of their personal financial situation. This goal was achieved to a large extent. The application successfully transforms raw bank transaction data into structured financial information and presents it in a way that is easier to understand and use for decision-making.

The solution addresses the initial problem that bank statements often contain useful financial data, but do not provide a clear overview of spending behaviour, budget status, or savings potential. By allowing users to import CSV data, categorize transactions, review payments, set budgets, and view financial KPIs, the application creates a clear end-to-end workflow from raw data to financial insight. From a functional perspective, the MVP fulfils the most important project objectives. The CSV import provides the basis for working with real transaction data. The categorization and review workflow improves the quality of the imported data. The overview page gives users a quick understanding of income, expenses, net balance, and top spending categories. The budget page supports financial planning by comparing planned limits with actual spending. Finally, the savings projection extends the system by helping users estimate possible future capital development.

The solution is also successful from a usability perspective. The application is structured into clear pages for import, review, overview, budgets, and savings. This makes the workflow understandable and avoids overloading the user with too much information on one screen. In addition, the application provides clear navigation buttons that guide the user to the next relevant page in the workflow, for example from import to overview or from budget planning to the budget overview. This supports a smoother user journey and helps users move through the main process step by step. The dark and consistent dashboard design supports readability and gives the application a modern and professional appearance. Important financial values are visually highlighted, which helps users quickly recognize their current financial situation.
From a technical perspective, the chosen local-first approach fits the project scope well. The use of FastAPI, Jinja templates, and SQLite allowed the team to build a working web application with a manageable architecture. The modular structure also supports maintainability, because the main responsibilities are separated into routes, services, feature modules, infrastructure, UI, and utilities. However, the solution should still be evaluated as an MVP and not as a fully finished commercial finance product. Some functions are intentionally limited in scope, and the application is mainly designed for local use and demonstration. Nevertheless, the implemented system provides a solid foundation for further development. The core workflow works, the main requirements are covered, and the application demonstrates how personal financial data can be transformed into useful insights through structured software engineering.

Overall, the project goal was achieved successfully. The Finance Analyzer delivers a functioning, understandable, and visually polished prototype that supports users in importing financial data, reviewing transactions, controlling budgets, and planning savings. It therefore provides real user value while also demonstrating the technical and methodological competencies required in the Project 2 module.

9.3 Limitations and Possible Future Improvements

Although the Finance Analyzer achieved its main functional goals, the current version still has several limitations. These limitations are mainly related to the defined project scope, the limited semester timeframe, and the conscious decision to focus on a functional MVP instead of a fully developed commercial finance platform.
Prioritized future work
High priority: encryption at rest and stronger local data protection. Financial data is sensitive, so a future version should first improve the protection of the local SQLite database, for example with SQLCipher or operating-system file encryption.
Medium priority: broader CSV compatibility and clearer validation feedback. More bank-specific CSV formats should be supported, and import errors should give users more precise information about what has to be fixed.

Medium priority: improved categorization and rule management. The existing rule-based approach could be extended with better pattern matching, rule priorities, and conflict detection.
Low priority: cloud synchronization, shared accounts, and mobile-specific UI improvements. These features would be useful for a larger product, but they are less important than security, import robustness, and the core local workflow for the current MVP.

One important limitation concerns the CSV import. The application can process common CSV structures by detecting delimiters, header rows, and relevant columns. However, bank exports can differ significantly depending on the financial institution, language settings, file structure, and formatting. Therefore, unusual or highly specific CSV formats may still require manual adjustment or further parser improvements. In a future version, the import logic could be extended to support more bank-specific formats and provide more detailed validation feedback to the user.
Another limitation is the rule-based categorization approach. The implemented mechanism works well for repeated merchants and recognizable transaction descriptions, but it may struggle with unclear, inconsistent, or completely new transaction texts. Within the project scope, this approach was appropriate because it is transparent, understandable, and feasible to implement during one semester. For future development, the categorization logic could be improved through more advanced pattern matching, better rule management and conflict detection.
The application is currently designed as a local-first tool. This supports privacy because financial data is stored and processed on the user's own machine. However, this also means that there is no synchronization across devices, no shared accounts, and no automatic cloud backup in the implemented MVP. These features are outside the current codebase. Future improvements should first focus on stronger local data protection, broader CSV compatibility, and clearer user feedback in the existing local workflow.

A further limitation concerns data security. The current MVP stores financial data locally in a standard SQLite database. This was suitable for the project scope and local demonstration, but encryption at rest was not implemented within the available semester timeframe. Since financial data is sensitive, a future version should evaluate technologies such as SQLCipher or file-based encryption to improve the protection of locally stored data.
The savings projection is also limited by its simplified assumptions. It can show possible future capital development based on user-defined values, contribution amounts, time horizon, and return scenarios. However, it does not replace professional financial advice and does not include factors such as inflation, changing income, taxes, or unexpected expenses. Future versions could improve this feature with more flexible assumptions and more detailed scenario comparisons.
In terms of usability, the application already provides a structured workflow and clear navigation between the main pages. Nevertheless, further UX improvements would be possible. For example, the mobile responsiveness could be improved, the budget overview could include more visual indicators, and the transaction review workflow could provide even clearer feedback after category changes.

Overall, the current version provides a strong foundation and demonstrates the core idea of the Finance Analyzer successfully. Due to the limited duration of the semester project, the development focused on the most important functions: CSV import, transaction categorization, budgeting, financial overview, and savings projection. Further development could make the application more flexible, secure, user-friendly, and suitable for long-term personal finance management.

10Individual Reflection Livio Strahm

10.1 Personal Contribution to the Project

My personal contribution to the project was mainly focused on project leadership, the initial setup of the application, the user experience, and the structural parts of the documentation. From the beginning of the project, I took on the role of project lead. This meant that I was responsible for keeping an overview of the project, defining the general direction, organizing tasks within the team, and making sure that the project stayed aligned with the planned timeline and the module expectations.

At the start of the implementation phase, I created the basic project structure and the first technical foundation of the application. This included setting up the general code structure and creating the basis on which the main functions of the Finance Analyzer could later be built. After the core mechanisms of the software were in place, my focus shifted more strongly toward improving the user experience and the overall usability of the application. A major part of my work was therefore related

to UX and UI refinement. Based on feedback from test users, I adjusted the interface so that the user path became more logical and easier to follow. For example, the application had to guide users more clearly from CSV import to transaction review, budget setup, and dashboard analysis. I also focused on reducing unnecessary information and limiting functions where they could overload the user. The goal was not to show everything that was technically possible, but to present only the information that helped the user understand the next step and gain financial insight.

Another important contribution was the development of a more consistent design language. I worked on making the application feel more coherent across the different pages by improving the layout, navigation, visual hierarchy, and general appearance. This was important because the Finance Analyzer should not only work technically, but also feel understandable and trustworthy for the user.
In addition to the software-related work, I contributed to the organization, documentation, and final presentation of the project. I assigned tasks, supported coordination within the team, kept track of open deliverables, created the basic structure of the documentation, and prepared the presentation slide deck and presentation flow. My focus was to make sure that both the report and the final presentation followed a logical order and communicated the project clearly.

Overall, my role combined technical work, UX improvement, project coordination, documentation structure, and presentation preparation. I tried to make sure that the project did not only result in a working application, but in a structured and understandable software project that could be explained, evaluated, and further developed.

10.2 Key Learnings

One of my most important learnings was that creating a first working version of an application has become much faster with AI-supported tools. With the help of such tools, it is possible to generate code, build first features, and create a functional prototype relatively quickly. However, I learned that the real work often starts after the first version is working. A working application is not automatically a good application.
Especially through vibe coding, the user interface can quickly become overloaded. Features, buttons, cards, and information can be added very fast, but this does not necessarily improve the user experience. In our project, I learned that a clear user path, a clean design layout, and a reduced amount of information are essential for making software understandable. Designing a logical workflow for the user was much more time-consuming than I first expected.

Another important learning was that good UX requires many small decisions. It is not only about making the interface look modern. It is about deciding what the user should see first, what the next logical step is, how feedback is shown after an action, and how complex information such as budgets or category spending can be presented in a simple way. This showed me that usability is a central part of software quality and not just a visual detail.
I also learned how important communication and organization are in a team project. The individual skills of the team members are important, but they only create value if they are used in a coordinated way. Clear task distribution, clear deadlines, and clearly defined deliverables helped us keep the project manageable. Without this structure, it would have been easy to lose focus or work on parts of the application that were not essential for the MVP.

A further learning was that leadership in a project does not mean doing everything alone. It means understanding the strengths of the team, assigning tasks in a useful way, supporting others where necessary, and keeping the overall direction clear. The challenge is to use the available resources in the team in a way that leads to the best possible product within the given timeframe.
Through this project, I gained a better understanding of how software development, user experience, documentation, and project management are connected. A successful project is not only based on code. It also depends on structure, communication, priorities, and the ability to decide what is important enough to implement and what should remain outside the current scope.

10.3 Challenges Faced

One of the main challenges I faced during the project was communication within the team. In some project phases, one team member worked very intensively on a specific part of the code. After that, it was challenging to explain the implemented changes to the other team members in a short and understandable way. Since the application consisted of several connected parts, such as routes, services, templates, database logic, and frontend pages, it was important that everyone understood enough of the current state to continue working effectively.

Another challenge was keeping the whole team aligned while the software evolved. When new functions or improvements were added, the documentation, requirements, and user flow also had to remain consistent. This required regular coordination and a clear overview of what had already been implemented, what still had to be improved, and what no longer fit into the project scope.
A further challenge was knowing when the application was good enough. A software project can always be improved. There are always more features, better designs, cleaner code structures, or additional ideas that could be added. During the project, I had to learn that trying to make the application perfect can become a risk, because time and resources are limited.

For our project, “good enough” meant that the application fulfilled the initial core requirements and that additional feedback from the project process was integrated where it clearly improved the product within the semester scope. This required prioritization. Some ideas had to be reduced, postponed, or left out, even if they would have been interesting from a product perspective.
The UX refinement was also challenging because small interface problems can strongly affect how users experience the software. For example, if the upload action is not visible enough, if the user does not receive clear feedback after saving a category, or if budget values are hard to interpret, the application may work technically but still feel unclear. Solving these problems required several iterations and a stronger focus on the user perspective.

Overall, these challenges helped me understand the difference between building software and coordinating a software project. It is not enough to implement features. The project also needs a clear structure, understandable communication, realistic priorities, and a focus on the user. This was one of the most valuable experiences for me during Project 2.

10.4 Critical Evaluation of Own Work

Looking back at my own work, I think my strongest contribution was keeping the project structured and focused. I helped define the direction of the application, supported the team organization, and improved the user experience so that the software became easier to understand. However, I also see that I could have defined responsibilities and communication routines even earlier. In some phases, technical changes became difficult to explain afterwards because too much had changed at once.
I also learned that I sometimes focused strongly on improving the product and design, which created the risk of spending too much time on details. For a future project, I would set clearer decision points for when a feature or design change is finished. This would help balance quality, time, and project scope better.

11 Individual Reflection Melih Aktas

11.1 Personal Contribution to the Project

My main responsibility during the Finance Analyzer project was feature implementation, bug fixing, and technical development of the application. Throughout the semester, I worked primarily on transforming requirements into working software components and ensuring that the system remained stable as new functionality was added.

A large part of my contribution focused on implementing and improving core features of the application. This included supporting the CSV import workflow, transaction processing, categorization

functionality, reporting features, and various backend-related improvements. During development, I regularly tested new functionality, identified errors, and fixed issues that appeared as the project evolved.

Another important part of my work was maintaining the stability of the codebase. As new features were integrated, unexpected bugs and compatibility problems occasionally appeared. I therefore spent considerable time debugging, testing, and refining existing functionality to ensure that different modules worked together correctly.

I also contributed to improving the overall structure of the application by helping integrate new features into the existing architecture. This required understanding how routes, services, feature modules, templates, and the database interacted with one another. Through this work, I gained a deeper understanding of full-stack software development and the importance of keeping code organized and maintainable.

In addition to implementation work, I participated in team discussions, reviewed technical decisions, tested functionality, and provided feedback on improvements. My goal was not only to add new features but also to ensure that the application remained reliable and user-friendly throughout the development process.

Overall, my contribution was mainly focused on coding, debugging, testing, and continuously improving the technical quality of the Finance Analyzer.

11.2 Key Learnings

One of the most valuable things I learned during this project was how software development works in a larger project environment. Before this project, most of my programming experience came from smaller assignments. Working on the Finance Analyzer showed me how multiple components must work together to create a complete application.

I significantly improved my Python programming skills and gained practical experience with FastAPI, SQLite, Jinja templates, and modular application structures. I learned how backend functionality, frontend pages, databases, and business logic interact with each other to create a complete user experience.

Another important learning was debugging and problem solving. During development, many issues were not immediately obvious and required systematic investigation. I learned how to analyze error messages, trace application behaviour, test assumptions, and identify the root cause of problems instead of only fixing symptoms.

I also gained a better understanding of version control and collaborative software development. Working with Git helped me understand how to manage code changes, integrate new features, and collaborate more effectively within a team environment.

A further learning was the importance of clean architecture and code organization. As the project grew, it became clear that well-structured code is easier to maintain, debug, and extend. This experience showed me that software engineering is not only about writing code but also about designing systems that remain manageable over time.

Overall, the project strengthened both my technical skills and my confidence in developing larger software systems.

11.3 Challenges Faced

One of the biggest challenges I faced was integrating new functionality without breaking existing features. As the application became larger and more complex, even small changes could sometimes affect other parts of the system unexpectedly. This required careful testing and a structured development approach.

Another challenge was debugging complex issues. Some problems were easy to identify, while others required examining several modules before the actual cause could be found. Learning how to systematically investigate these issues was an important part of my development throughout the semester.

Managing project scope was also challenging. There were many ideas for additional features and improvements, but the available time was limited. This meant that priorities had to be defined carefully so that the most important functionality could be completed first. Learning when to stop adding features and focus on stability and quality was an important lesson.

Communication within the team was another challenge. When different team members worked on different parts of the application, it was important to keep everyone informed about changes and ensure that all components remained compatible. Regular discussions and reviews helped reduce misunderstandings and maintain progress.

Finally, balancing technical implementation with documentation requirements was challenging. Besides developing the application itself, the project also required structured documentation, testing evidence, architectural explanations, and preparation for the final presentation. Managing both the technical and academic aspects of the project helped me develop a more complete understanding of software engineering projects.

Overall, these challenges helped me improve my programming skills, problem-solving abilities, and understanding of professional software development processes.

11.4 Critical Evaluation of Own Work

Looking back at my own work, I think my strongest contribution was the technical implementation and the continuous improvement of the application. I was able to solve problems, debug errors and help keep the system stable while new features were added. At the same time, I could have documented some technical changes more directly while implementing them. This would have made it easier for the whole team to understand later why certain decisions were made.
I also see that I could have communicated complex technical changes earlier and in smaller steps. Sometimes a problem was solved in the code, but the explanation for the team came later. In a future project, I would combine implementation, short documentation notes, and team updates more consistently so that technical progress remains easier to follow.

12 Individual Reflection Marc Iseli

12.1 Personal Contribution to the Project

My personal contribution to the project was mainly focused on supporting the documentation, understanding the technical structure of the application, and gradually contributing to the implementation. At the beginning of the project, I had some difficulties with programming because I had not worked on a software project of this size before. Compared to the other team members, I therefore initially took on a smaller role in the coding part of the project.
During the semester, I became more familiar with the project structure, GitLab workflow and the general development process. This helped me understand how the different parts of the application interact with each other, such as routes, services, templates, database functions, and frontend pages. As my understanding improved, I was able to contribute more actively to the codebase. One example of this was my work on the savings page, where I supported the implementation and improvement of the savings projection functionality.
In addition to smaller code contributions, I supported the team by helping with testing, checking the application from a user perspective, and reviewing whether the implemented functions were understandable and usable. This was especially important because the Finance Analyzer should not only work technically, but also guide the user clearly through the workflow from CSV import to overview, budgets, and savings planning.

A major part of my contribution was the technical documentation. I worked on describing the project structure, implementation logic, architectural decisions, code snippets, functional scope, limitations, and reflection parts. I focused on making the documentation understandable and traceable, so that the connection between the project idea, requirements, architecture, implementation, and final result becomes clear.

Overall, my role developed during the project. At the beginning, I mainly supported the team through documentation and smaller tasks, while later I was able to contribute more to the technical implementation as well. This progression helped me gain a better understanding of software development and allowed me to contribute to both the written report and the final application.

12.2 Key Learnings

One of my biggest learnings during the project was writing and understanding code in a larger software project. At the beginning, I had difficulties because I had not worked on an application of this size before. During the semester, I learned how important it is to understand the structure of the code before making changes. This helped me see how the different parts of the application work together.
Another important learning was how to use GitLab correctly. I learned how version control supports teamwork and makes changes traceable. Working with GitLab helped me understand why commits, clear changes, and a structured workflow are important when several people work on the same project.
I also learned to see the bigger picture of a software project. The Finance Analyzer is not just one page or one function, but a system with many connected parts, such as routes, services, templates, database logic, frontend pages, and documentation. Understanding this overall structure helped me better understand where a change belongs and how it affects other parts of the application.

A further important learning was the value of making small changes step by step. Instead of changing many things at once, I learned to adjust one part in the code, test whether it works, and then continue with the next change. This made the development process safer and easier to understand. It also helped me find mistakes more quickly, because I could see which specific change caused a problem.
Through this project, I became more confident in working with code, using GitLab, testing small changes, and understanding how a software application is built. The project showed me that programming is a process of learning step by step and that even small improvements can contribute to the final result when they are tested and documented properly.

12.3 Challenges Faced

One of the main challenges I faced was understanding the codebase at the beginning of the project. Since I had not worked on a software project of this size before, it was difficult for me at first to understand how the different files, folders, and functions were connected. The application included several parts. Understanding this overall structure took time.
Another challenge was writing code without accidentally affecting other parts of the application. At the beginning, I was unsure where changes should be made and how they would influence the rest of the system. I learned that it is important to make small changes step by step and test them directly afterwards. This helped me reduce mistakes and understand the effect of each code change better.

Working with GitLab was also challenging at first. I had to learn how to use version control correctly, how to handle changes, and how to work in a shared project structure. Over time, I became more comfortable with this workflow and understood why GitLab is important for collaboration, traceability, and keeping the project organized.

A further challenge was the limited time during the semester. The project required not only coding, but also requirements engineering, architecture, UX considerations, testing, documentation and preparation for the final presentation. Because of this, it was necessary to focus on the most important functions and accept that some advanced improvements had to remain outside the current project scope.

Writing the technical documentation was also challenging. It was not enough to only describe what the application does. The documentation also had to explain why certain decisions were made, how the system is structured and how the implementation connects to the requirements. This required a good understanding of both the code and the overall project logic.

Overall, these challenges helped me improve throughout the semester. I learned that it is normal to need time to understand a larger software project. By working step by step, testing changes regularly, using GitLab more confidently, and supporting the documentation, I was able to contribute more effectively to the final result.

12.4 Critical Evaluation of Own Work

Looking back at my own work, I think my contribution became stronger over time. At the beginning, I was less confident with the codebase and therefore focused more on documentation and smaller tasks. This helped the team, but I could have become more active in the technical implementation earlier. Asking more specific questions and working through smaller code tasks from the beginning would have helped me build confidence faster.
I also see that I sometimes needed too much time to understand how different parts of the application were connected. For a future project, I would structure my learning process earlier, for example by writing short notes about important files, routes, and functions while exploring the code. This would make it easier to contribute to both the code and the documentation from an earlier stage.

13 List of Figures

List of Figures

Figure 1: CSV Upload Form in the UI Layer 24
Figure 2: FastAPI Application Setup in the Application Layer 25
Figure 3: CSV Parsing Logic in the Feature Layer 26
Figure 4: CSV Parsing Logic in the Feature Layer 27
Figure 5: Rule-Based Transaction Categorisation 28
Figure 6: Monthly Expense Trend Calculation 29
Figure 7: SQLite Database Connection and Schema Initialization 30
Figure 8: SQLite Database Connection and Schema Initialization 31
Figure 9: Utility Functions for Text and Amount Normalization 33
Figure 10: UML Class Diagram of the Finance Analyzer data model 35
Figure 11: Activity Diagram of the CSV import and processing workflow 36
Figure 12: Project Folder Structure of the Finance Analyzer 42
Figure 13: Financial Amount Normalisation Function 43
Figure 14: Budget Report Calculation Logic 44
Figure 15: Database Schema Initialization in the Infrastructure Layer 46

14 List of Tables

Table 1: Semester Project Schedule and Iterations
Table 2: Stakeholder Matrix
Table 3: Requirement Elicitation Sources
Table 4: Functional Requirements
Table 5: Non-functional Requirements
Table 6: Optional Requirements
Table 7: Acceptance Criteria and Requirements Status
Table 8: Logical Architecture Overview
Table 9: UX Review Feedback Log
Table 10: Core Feature Implementation Overview
Table 11: Technical Challenges and Solutions
Table 12: Requirements Traceability Matrix
Table 13: Automated Test Evidence

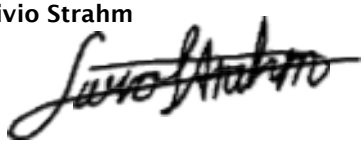
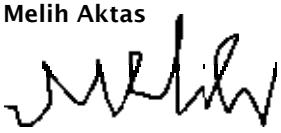
15 Declaration of Originality

I confirm that I have completed this paper independently and without the use of any sources or aids other than those listed in the bibliography. All passages that do not originate from me are marked as quotations and include a precise reference to their source.

During the preparation of this paper, we used AI-supported tools such as ChatGPT or Copilot for research and revision.

Place, Date: Biel,

Signature:

Livio Strahm	Marc Iseli	Melih Aktas
		

16 Appendix