Contents

Chapter 1 Capstone Project Document

In this project, students will build a Simple Banking Application in Python using Clean Architecture, SOLID principles, and several object-oriented design patterns. Students will incrementally deliver a working system each week, clearly reflecting user requirements and demonstrating thoughtful design.

Students are required to use **ClickUp** for project management. They must invite both the instructor and **Dr. Rashida** to their workspace so that progress can be monitored and timely feedback can be provided.
Below are the four weekly sprints, each incrementally refining the application with new features and system modifications. The User Requirements are phrased as stories to capture real-world scenarios.

1.1 Week 1: Core Account & Transaction Management

1.1.1 User Requirements

1. **Create Checking or Savings Accounts**
   - *User Story:* "As a user, I want to open a new checking or savings account so I can store my money in the system."
   - *Design Criteria:* The solution must allow the easy addition of future account types (e.g., fixed deposit accounts) with minimal changes.
2. **Deposit & Withdraw Funds**
   - *User Story:* "As a user, I want to deposit money into my account or withdraw from it if my balance allows."
   - *Design Criteria:* The deposit/withdraw processes follow a consistent workflow but allow for account-type-specific rules or overrides.
3. **View Account Balance & Transaction History**
   - *User Story:* "As a user, I want to see my current/available balance and past transactions to track my money usage."

1.1.2 Domain Layer

1. **Account Entity**
   - *Must store:* `accountId`, `accountType`, `balance`, `status` (active, closed), `creationDate`.
   - *Behavior:* Should allow for specialized behavior if needed (e.g., limit on how low a balance can go for certain account types).
2. **Transaction Entity**
   - *Represents:* deposit or withdrawal.
   - *Fields:* `transactionId`, `transactionType` (DEPOSIT, WITHDRAW), `amount`, `timestamp`, plus a reference/ID of the related account.
3. **Domain Services (optional)**
   - Could add a small *"business rule check"* service, e.g., ensuring no negative balances or verifying deposit amounts.

Provide minimal but well-structured classes that can evolve in future weeks.

1.1.3 Application Layer

1. **AccountCreationService**
   - *Responsibility:* Creating accounts (checking, savings).
   - *Method:*
     - `create_account(accountType, initialDeposit=0.0) -> accountId`
   - *Notes:* Enforces any domain rules (e.g., if the bank requires a minimum deposit for savings).
2. **TransactionService**
   - *Responsibility:* Handling deposits and withdrawals.
   - *Methods:*
     - `deposit(accountId, amount) -> Transaction`
     - `withdraw(accountId, amount) -> Transaction`
   - *Notes:* Checks domain invariants (e.g., no negative balance if that's disallowed).

Goal: Each service orchestrates domain objects (`Account`, `Transaction`) without referencing frameworks or storage details.

1.1.4 Infrastructure Layer

1. **AccountRepository**
   - *Responsibility:* Stores and retrieves accounts.
   - *Methods:*
     - `create_account(Account) -> accountId`
     - `get_account_by_id(accountId) -> Account or None`
     - `update_account(Account) -> None`
   - *Notes:* Could be an in-memory Python dictionary or a real database.
2. **TransactionRepository**
   - *Responsibility:* Stores and retrieves transactions.
   - *Methods:*
     - `save_transaction(Transaction) -> transactionId`
     - `get_transactions_for_account(accountId) -> List[Transaction]`

Goal: Keep all database, file, or any other I/O logic out of the domain and application layers.

1.1.5 Presentation Layer (API Endpoints)

Below is a minimal set of endpoints for Week 1 (assuming a framework like FastAPI or Flask):

1. **Create Account**
   - *Method:* `POST /accounts`
   - *Body:* `{ "accountType": "CHECKING" | "SAVINGS", "initialDeposit": 100.0 }`
   - *Description:* Creates a new account.
   - *Calls:* `AccountCreationService.create_account(...)`
   - *Response:* `201 Created` + JSON with new account details.
2. **Deposit**
   - *Method:* `POST /accounts/{accountId}/deposit`
   - *Body:* `{ "amount": 50.0 }`

- *Description:* Deposits funds into an account.
- *Calls:* `TransactionService.deposit(accountId, amount)`
- *Response:* `200 OK` + updated balance or transaction info.

3. **Withdraw**
   - *Method:* `POST /accounts/{accountId}/withdraw`
   - *Body:* `{ "amount": 20.0 }`
   - *Description:* Withdraws funds if allowed by domain rules.
   - *Calls:* `TransactionService.withdraw(accountId, amount)`
   - *Response:* `200 OK` + updated balance or transaction info.
4. **Get Account Balance**
   - *Method:* `GET /accounts/{accountId}/balance`
   - *Description:* Returns current balance.
   - *Calls:* `AccountRepository.get_account_by_id(accountId)`
   - *Response:* `200 OK` + JSON: `{ "balance": ..., "availableBalance": ... }`
5. **Get Transaction History**
   - *Method:* `GET /accounts/{accountId}/transactions`
   - *Description:* Returns a list of past transactions for that account.
   - *Calls:* `TransactionRepository.get_transactions_for_account(accountId)`
   - *Response:* `200 OK` + JSON array of transaction objects.

Goal: Provide enough endpoints to demonstrate creation, deposit/withdraw, and transaction history queries.

1.1.6 Deliverables for Week 1

- **Domain**
  - Python classes for `Account`, `Transaction` with basic tests.
- **Application**
  - `AccountCreationService`, `TransactionService` implementing the main use cases.
- **Infrastructure**
  - Minimal repository implementations for accounts and transactions.
- **Presentation**
  - Functional REST API endpoints for account creation, deposit/withdraw, balance retrieval, and transaction history.
  - Basic tests or manual verification (`Postman`/`curl`).
- **Documentation**
  - UML diagrams (class + optional sequence).
  - A short explanation of how each layer works and how they integrate.

Success Criterion: A running application that supports core account features. The design must remain open to easy expansion (e.g., adding new account types later) and maintain separation of concerns for maintainability and clarity.

1.2 Week 2: Transfers, Notifications & Logging

1.2.1 User Requirements

1. **Fund Transfers Between Owned Accounts**
   - *User Story:* "As a user, I want to move funds from one of my accounts to another quickly."
   - *Design Criteria:* Transfers should be encapsulated in a discrete operation that allows flexible execution or potential reversal if needed.
2. **Automatic Transaction Notifications**
   - *User Story:* "As a user, I want to receive email or SMS alerts whenever a deposit, withdrawal, or transfer happens."
   - *Design Criteria:* Notifications must be dispatched automatically once a transaction is finalized, without scattering notification code throughout transaction logic.
3. **Transaction Logging**
   - *User Story:* "As a user, I want detailed logging without the core code being cluttered, so I can audit all my transactions."
   - *Design Criteria:* Logging should be applied flexibly (and selectively if needed), without requiring changes to the core transaction classes.

1.2.2 Domain Layer: Account and Transaction (Refinements)

1. **Account**
   - Already contains fields like `accountId`, `balance`, and potentially `ownerId`.
   - Might need a method or domain rule check for transferring funds out (similar to withdrawal checks).
2. **Transaction**
   - May now include a new `transactionType` for `TRANSFER`.
   - Possibly store a `sourceAccountId` and `destinationAccountId` if it's a transfer.
3. **Transfer-Specific Domain Logic (Optional Separate Service/Class)**
   - If you treat transfers as unique from a simple "withdraw + deposit," your domain might define an entity or method to handle them atomically (so either both deposit and withdrawal succeed or both fail).

Goal: Domain logic ensures consistent changes to source/destination accounts, as well as storing a correct transaction record.

1.2.3 Application Layer: Fund Transfer, Notification, and Logging

1. **FundTransferService**
   - *Responsibility:* Expose an operation like `transfer_funds(sourceAccountId, destinationAccountId, amount)`.
   - *Internally:*
     - Ensure source account can withdraw (balance check).
     - Withdraw from source.
     - Deposit into destination.
     - Store a `TRANSFER` transaction record referencing both accounts.

2. **NotificationService**
   - *Responsibility:* When a transaction is completed, trigger emails/SMS to relevant users.
   - This service might be invoked by the `TransactionService` or a suitable domain event approach.
   - *Method:*
     - `notify(transaction: Transaction) -> None` (sends out relevant messages to the account owner(s)).
3. **Logging Setup**
   - You could add a specialized `LoggingService` or incorporate a logging "layer" around your existing services.
   - For example, if your `TransactionService` was implemented in Week 1, you might now wrap it with a component that logs each operation (deposit, withdraw, transfer) without altering the original code.

Goal: Keep transaction logic and notifications/logging separate, ensuring the main business logic remains uncluttered.

1.2.4 Infrastructure Layer (Refined)

1. **Repositories**
   - **AccountRepository:**
     - Add a method to handle concurrency or at least check balances before transferring.
     - Optionally, have a single method that can update both source and destination accounts in a transaction-like approach if using a real database.
   - **TransactionRepository:**
     - Supports saving the new `TRANSFER` type transaction.
     - Possibly provide a `find_by_transaction_id` method if needed.
2. **Notifications Integration**
   - **NotificationAdapter:**
     - Provide an interface for sending messages (SMS, email).
     - Real or mock providers should implement methods like:
       - `send_email(recipient, subject, body)`
       - `send_sms(number, message)`
3. **Logging Mechanism**
   - **Logger:**
     - A central logger or separate logs per transaction type.
     - Could use Python's built-in `logging` module or more advanced solutions that wrap your services to track method calls.

Goal: Infrastructure classes handle external concerns (e.g., database, email gateway, log files) so that the Application layer remains focused purely on domain-specific steps.

1.2.5 Presentation Layer (API Endpoints - Week 2)

Below are suggested endpoints for Week 2. They assume the existence of the basic endpoints from Week 1.

1. **Fund Transfer**
   - *Method:* `POST /accounts/transfer`

- *Body:* `{ "sourceAccountId": 1, "destinationAccountId": 2, "amount": 100.0 }`
- *Description:* Transfers funds from source to destination.
- *Calls:* `FundTransferService.transfer_funds(...)`
- *Response:* `200 OK` + JSON with the resulting transfer transaction.

2. **Notifications**
   - *Note:* Notifications are typically triggered automatically upon transaction completion.
   - If users should be able to configure notifications:
     - `POST /notifications/subscribe`
     - `POST /notifications/unsubscribe`
     - *Body:* `{ "accountId": 1, "notifyType": "email" }`
3. **Retrieve Logs (Optional)**
   - *Method:*
     - `GET /logs/transactions`
     - `GET /accounts/{accountId}/logs`
   - *Description:* View logs of transactions for users or admins.
   - *Note:* Logging is mainly an internal concern, but can be exposed if needed for audit or transparency.

Goal: Show how new features (transfers, notifications, logging) are surfaced or triggered from the outside.

1.2.6 Week 2 Deliverables

- **Updated Domain**
  - Transfer logic integrated into the existing `Transaction` entity or defined through a new `TransferTransaction` class.
  - Minor expansions to the `Account` entity to support concurrency handling or ensure consistent state updates.
- **Application Services**
  - `FundTransferService` with full test coverage.
  - `NotificationService` responsible for dispatching events (e.g., SMS or email) after a transaction is completed.
- **Infrastructure**
  - New or updated `Repositories` to support transfer atomicity (e.g., simultaneous update of source and destination accounts).
  - `NotificationAdapter` implementations for email/SMS providers.
  - Logging solution or wrapper component to track service calls and transaction completions.
- **Presentation**
  - New or updated REST API endpoints for fund transfers and (optionally) notification preferences.
  - Verified through Postman/cURL or automated tests to ensure correctness and consistency of transfers and logging.
- **Documentation**
  - UML diagrams showing newly introduced classes and their interactions.
  - A brief technical explanation of how the transfer, notification, and logging workflows are integrated into the existing architecture.

Success Criterion: A functional system where transferring funds automatically triggers notifications and is

captured in logs, all while retaining the separation of concerns that Clean Architecture prescribes.

1.3 Week 3: Interest, Limits & Monthly Statements

1.3.1 User Requirements Recap

1. **Interest Computation**
   - *User Story:* "As a user, I want interest automatically calculated for my account, with different rates or formulas depending on my account type."
   - *Design Criteria:* The interest calculation logic should be easily interchangeable, so you can add or modify interest formulas without breaking existing code.
2. **Transaction Limits**
   - *User Story:* "As a user, I want daily or monthly transaction limits on my account to protect me from overspending or fraud."
   - *Design Criteria:* Limits should be attachable or removable without rewriting the account's core logic, thus enabling flexible toggling of additional constraints.
3. **Monthly Statements**
   - *User Story:* "As a user, I want a monthly statement in PDF or CSV format that summarizes my balance, interest earned, and transactions."
   - *Design Criteria:* Statement generation should follow a stepwise/structured approach, easily integrating external libraries for PDF/CSV. Code that converts statement data into final output should remain separate from the domain's business rules.

1.3.2 Domain Layer: Interest Logic, Transaction Limits, and Monthly Statement

1. **Interest Logic**
   - *Account Entity Update:*
     - Potentially store a reference or field indicating which interest approach/formula to use (e.g., `interestStrategyId` or another configurable attribute).
   - *Interest Calculation Options:*
     - **Option A:** Each account type implements its own `calculate_interest()` method.
     - **Option B:** Use a separate domain service or an *interest strategy* object attached to the account.
2. **Transaction Limits**
   - *Approach 1:* Store limit settings directly in the `Account` entity:
     - Fields such as `dailyLimit`, `monthlyLimit`, and counters for amount spent in the current day or month.
   - *Approach 2:* Use a decorator-like or constraint-based approach:
     - A `LimitConstraint` object that intercepts or wraps domain methods like deposit/withdraw to enforce rules.
3. **Monthly Statement**
   - Domain must track additional data needed for monthly statements:
     - Possibly the monthly interest accrued.
     - Summary data such as transaction totals, final balance, etc.

1.3.2.0.1 Goal: The domain remains open for future changes—if new interest formulas or limit logic are introduced, only minimal and localized code modifications should be required.

1.3.3 Application Layer: Interest, Limits, and Statements

1. **InterestService**
   - *Responsibility:* Orchestrate periodic interest calculations.
   - *Example Methods:*
     - `apply_interest_to_account(accountId)`
     - `apply_interest_batch(accountIds)`
2. **LimitEnforcementService (Optional Approach)**
   - *Responsibility:* Manage daily/monthly limit constraints.
   - This service could be:
     - Tied into deposit/withdraw flows.
     - A separate "interceptor" that checks usage before allowing transactions to proceed.
   - *Methods:*
     - `check_limit(accountId, transactionAmount) -> bool`
     - `reset_limits_daily()` or `reset_limits_monthly()` — for use in scheduled jobs or system-level resets.
3. **StatementService**
   - *Responsibility:* Gather relevant transaction and interest data for a specified period.
   - Passes this data to a PDF/CSV generator implemented in the Infrastructure Layer.

1.3.4 Infrastructure Layer

1. **Interest Implementation**
   - If interest rates or rules are defined externally (e.g., in a config file or fetched from a service), the Infrastructure layer can manage storage and retrieval.
   - The domain or application layer retrieves the rates via a repository or adapter abstraction.
2. **Transaction Limit Storage**
   - Extend the `AccountRepository` to store and retrieve daily/monthly usage data or limit thresholds.
   - Optionally define a separate `AccountConstraintsRepository` to modularize limit logic and storage.
3. **Statement Generation (PDF/CSV)**
   - Use a `StatementBuilder` or `StatementAdapter` that receives structured data from the `StatementService`.
   - Use external libraries such as:
     - `ReportLab` for PDF generation.
     - Python's built-in `csv` module for CSV generation.
   - You may:
     - Have separate adapters for each format.
     - Use a single adapter that handles multiple formats for simplicity.

1.3.5 Presentation Layer (API Endpoints)

1. **Interest Endpoint**

- *Method:* `POST /accounts/{accountId}/interest/calculate`
- *Body:* Optional context, e.g., `{ "calculationDate": "YYYY-MM-DD" }`
- *Description:* Triggers interest calculation for the specified account.
- *Calls:* `InterestService.apply_interest(accountId, calculationDate)`
- *Response:* `200 OK` + JSON describing interest applied and updated balance.

2. **Transaction Limits Endpoints (Optional)**
   - **Update Limits**
     - *Method:* `PATCH /accounts/{accountId}/limits`
     - *Body:* `{ "dailyLimit": 500, "monthlyLimit": 2000 }`
     - *Description:* Updates the account's daily/monthly limit settings.
   - **Retrieve Limits**
     - *Method:* `GET /accounts/{accountId}/limits`
     - *Description:* Retrieves current limit settings and usage stats if tracked.
3. **Monthly Statements**
   - *Method:* `GET /accounts/{accountId}/statement`
   - *Query/Body:* e.g., `?year=2025&month=03&format=pdf`
   - *Description:* Generates a monthly statement for the specified period in PDF or CSV format.
   - *Calls:* `StatementService.generate_monthly_statement(accountId, year, month, format)`
   - *Response:* `200 OK` with either file download or direct binary streaming.

1.3.5.0.1 Goal: Students see how domain logic for interest, limits, and statements is surfaced in a straightforward REST API.

1.3.6 Deliverables for Week 3

- **Updated Domain**
  - Mechanisms for interest calculation, including new fields or strategy references per account type.
  - Transaction limit data or logic integrated into account behavior.
  - A pathway for monthly statement-related data (e.g., accrued interest, summaries).
- **Application Layer**
  - `InterestService`, `StatementService`, and any limit enforcement mechanism.
  - Tests verifying interest calculation accuracy, limit enforcement, and correctness of statement data.
- **Infrastructure**
  - Optionally, an `InterestStrategyRepository` or configuration-based interest rate handling.
  - `StatementAdapter` or `StatementBuilder` for generating PDF/CSV monthly reports.
  - Database or model schema updates to store and manage transaction limit usage data.
- **Presentation**
  - REST API endpoints for interest calculation, limit configuration, and statement generation.
  - Endpoints verified using tools like `Postman`, `cURL`, or through automated unit/integration testing.
- **Documentation**
  - UML diagrams showing how interest logic, limit systems, and statement generation integrate with existing modules.
  - Explanation of the statement-building pipeline and how new interest strategies/formulas can be in-

tegrated seamlessly.

Success Criterion: A system that seamlessly handles different interest requirements, toggles transaction limits, and exports monthly statements in PDF/CSV—while preserving domain-focused logic and enabling minimal code changes when introducing new account behaviors.

1.4 Week 4: Loan Management, Fraud Detection & Final Security Enhancements

User Requirements (Advanced)

1. **Loan Applications & Management**
   - *User Story:* "As a user, I want to apply for a loan, see my repayment schedule, and keep track of my outstanding loan balance."
   - *Design Criteria:* Loan-related capabilities should be exposed through a simple interface that hides underlying complexity, such as interest calculations or repayment schedules.
2. **Fraud Detection Alerts**
   - *User Story:* "As a user, I want suspicious transactions automatically flagged, with immediate alerts."
   - *Design Criteria:* Multiple checks should be applied in a sequence (a pipeline or chain) to detect suspicious activity, and the system should be easily extensible with new or reordered checks.
3. **Final System Release & Documentation**
   - *User Story:* "As a user, I want to confidently rely on a stable, fully featured banking platform."
   - *Deliverables:*
     - Fully integrated loan management module.
     - Fraud checks orchestrated in a chain/pipeline.
     - Final integrated system deployment (e.g., Heroku, Docker, AWS).
     - Complete documentation including user guide, developer notes, and final UML diagrams.

1.4.1 Domain Layer: Loan Entities and Logic

1. **Loan**
   - *Fields:* `loanId`, `principal`, `interestRate`, `repaymentSchedule`, `balanceRemaining`, etc.
   - *Optional:* Track a list of `LoanRepayment` entities, each storing `date`, `amountPaid`, and `newBalance`.
2. **Domain Logic**
   - Calculation of monthly installments, total interest, or next payment date.
   - Track loan status (e.g., `"approved"`, `"pending"`) to control state transitions within the domain.
3. **Fraud Detection**
   - You may define a domain-level concept for fraud checks or delegate it to the application layer.
   - Example checks:
     - `HighAmountCheck`
     - `UnusualFrequencyCheck`

Goal: Keep the loan's core logic encapsulated in methods or domain services so that adding new loan types or repayment schemes is straightforward and non-invasive to existing code.

1.4.2 Application Layer: Loan & Fraud Services

1. **LoanService**
   - *Responsibility:* Orchestrate all operations related to loans—application, approval, due amount calculations, and schedule updates.
   - *Methods:*
     - `apply_for_loan(accountId, principal, interestRate, termMonths) -> loanId`
       Creates a Loan entity, sets up the initial schedule, and saves it.
     - `make_repayment(loanId, amount) -> updatedLoan`
       Reduces the loan balance, stores repayment transaction, and updates state.
     - `get_loan_info(loanId) -> Loan`
       Returns a domain Loan object or summary data.
2. **FraudDetectionService**
   - *Responsibility:* Evaluate transactions or transfers for suspicious indicators.
   - *Method:*
     - `check_transaction(transaction: Transaction) -> FraudResult`
   - *Implementation Pattern:*
     - Uses a pipeline or chain structure of checks, such as:
       - `HighAmountCheck`
       - `VelocityCheck`
       - `BlacklistCheck`
     - Each check returns a pass/fail or severity rating.
     - The final result can trigger a fraud flag or an alert.

Goal: Keep loan and fraud logic encapsulated in separate service classes, each responsible for orchestrating domain-level behavior, validations, and updates.

1.4.3 Infrastructure Layer: LoanPersistence and FraudChecks

1. **LoanPersistence**
   - **LoanRepository:**
     - `create_loan(Loan) -> loanId`
     - `update_loan(Loan) -> None`
     - `get_loan_by_id(loanId) -> Loan or None`
2. **FraudChecks (if checks rely on external data)**
   - For example, a fraud check may query an external *blacklist* service or account database.
   - Each check can be integrated via specialized adapters for external APIs or databases.
3. **Alerts/Notifications (Optional)**
   - If a fraudulent transaction is detected, use the existing `NotificationAdapter` from Week 2.
   - Alternatively, introduce a specialized `FraudAlertAdapter` for fraud-specific notifications.

Goal: The infrastructure layer holds the actual database persistence logic for loans and integrates with external sources (e.g., blacklist services or alerting systems) to support advanced fraud detection and notifications.

1.4.4 Presentation Layer (API Endpoints)

1. **Loan Management**
   - **Apply for Loan**
     - *Method:* `POST /loans`
     - *Body:* `{ "accountId": 123, "principal": 5000, "interestRate": 0.05, "termMonths": 12 }`
     - *Description:* Creates a new loan for the given account and sets up a repayment schedule.
     - *Calls:* `LoanService.apply_for_loan(...)`
     - *Response:* `201 Created` + JSON with the new `loanId`.
   - **Get Loan Info**
     - *Method:* `GET /loans/{loanId}`
     - *Description:* Retrieves details about the loan: principal, outstanding balance, and payment history.
     - *Calls:* `LoanService.get_loan_info(loanId)`
     - *Response:* `200 OK` + JSON with loan info.
   - **Make Loan Repayment**
     - *Method:* `POST /loans/{loanId}/repay`
     - *Body:* `{ "amount": 100.0 }`
     - *Description:* Pays the specified amount toward the loan balance.
     - *Calls:* `LoanService.make_repayment(loanId, amount)`
     - *Response:* `200 OK` + JSON with updated loan balance or repayment summary.
2. **Fraud Detection**
   - **Optional Manual Check Endpoint**
     - *Method:* `POST /fraud/check`
     - *Body:* `{ "transactionId": 9999 }`
     - *Description:* Invokes `FraudDetectionService` to evaluate a specific transaction.
     - *Response:* `200 OK` + result (e.g., `"clear"` or `"suspicious"`).
   - **Note:** Main transaction endpoints (deposit, withdraw, transfer) can automatically call `FraudDetectionService` behind the scenes. A separate endpoint is not strictly required unless supporting manual workflows or testing scenarios.

1.4.5 Week 4 Deliverables

- **Domain**
  - `Loan` entity with any associated structures (e.g., repayment schedule).
  - Fraud detection logic or domain hooks as appropriate.
- **Application Layer**
  - `LoanService` with methods for applying, repaying, and retrieving loan information.
  - `FraudDetectionService` orchestrating a chain or pipeline of fraud checks.
- **Infrastructure**
  - `LoanRepository` for database or in-memory storage of loan data.
  - Potential integration with external systems for advanced fraud checks or notifications.
- **Presentation (APIs)**

    - Endpoints for loan creation, loan info retrieval, and loan repayment.
    - *Optional:* An endpoint for manual fraud checks, or rely on automatic invocation during transactions.
- **Final Deployment**
    - Deployable using Docker, Heroku, AWS, or a similar platform.
    - Accompanied by a comprehensive `README` or internal wiki with deployment instructions.
- **Complete Documentation**
    - Final UML class and sequence diagrams.
    - A user manual describing the complete functionality of the banking application.
    - Developer notes on code organization, testing, and how to extend functionality.

Success Criterion: A robust, fully integrated banking system supporting loan management and a flexible fraud detection pipeline, confidently deployable and thoroughly documented for real or near-real usage.

1.5 Expected Learning Outcomes

- **Clean Architecture in Practice:** Students will appreciate how domain logic remains independent from frameworks and external services.
- **SOLID Principles & OO Design:** Through incremental additions, students learn the necessity of SOLID for maintainable, flexible code.
- **Critical Thinking on Design Patterns:** By encountering user-driven requirements, students must decide (or confirm) which patterns fit best.
- **Incremental Refinement:** Experience in modifying an existing codebase without major rewrites, a crucial real-world skill.
- **Technical Communication:** Confidence in producing professional-level documentation and system diagrams.