AI Usage Report

Tools, prompts, iterations, and human interventions

Author: Karan Mehta
Date: 2026-02-25

Table of Contents

1. Tools Used

Primary AI Assistant: Antigravity IDE Agent (@Workspace), Gemini Pro 3.1 (prompt engineering), Chatgpt (Review) **Execution Environment**: Antigravity IDE, Docker Desktop. **Testing/QA**: Browser DevTools, Swagger UI (/docs), Pytest

2. Challenges Faced and How AI Helped

Boilerplate Acceleration: The AI rapidly scaffolded domain-driven directories, Pydantic schemas, and Tailwind CSS grids. **The "Happy Path" Bias**: The AI frequently optimized for the "happy path," omitting boundary validations and negative test scenarios. I had to explicitly prompt for these. **Context Loss**: Shifting to a production Docker setup caused CORS blocks due to port changes, requiring manual explicit context injection for the AI to resolve.

3. Areas Where Manual Intervention Was Necessary

Financial Data Integrity: Overruled the AI's `float` recommendation to enforce `Integer` (cents) for currency fields. **ACID Compliance**: Explicitly commanded the AI to wrap database transfers in `try/except` blocks with `await session.rollback()`. **Security Architecture**: Intervened to ensure IDOR validations (`account.user_id == current_user.id`) were present on all protected routes. **DevOps & Deployment**: Manually audited Docker configurations for multi-stage builds and implemented defensive Bash scripting in `start.sh`.

NOTE: I have included manual interventions step with every prompt and iteration with the AI. Please refer to the "Human Review & Intervention" in each iterations below to learn more.

4. Prompts and Iterations

4.1 Phase 1: Database Scaffolding, Schema Design, & DevOps Baseline

Phase Objective: Initialize the domain-driven FastAPI structure, design the SQLite database schema, and establish the version control baseline.

4.1.1 Iteration 1: Initial Architecture Scaffolding

Objective: Establish the foundational folder structure and map the SQLAlchemy ORM models.

The Prompt:

```
@Workspace /plan

<system_constraints>
- Role: Senior Backend Architect.
- Tech Stack: Python 3.12+, FastAPI, SQLAlchemy 2.0 (Async), SQLite, Pydantic
v2, Alembic.
```

```
- Strict Rule 1: DO NOT write any code or create files yet.
- Strict Rule 2: Generate an `Implementation Plan` artifact first. Wait for
human approval.
- Strict Rule 3: Maintain a domain-driven modular structure, avoiding
monolithic files.
</system_constraints>

<context>
We are building a production-ready Banking REST Service.
The objective for this phase is to establish the project scaffolding and
configure the database layer.
</context>

<database_schema>
Design the SQLAlchemy ORM models tailored for SQLite:
- `User`: id (UUID), email, hashed_password, is_active.
- `Account`: id (UUID), user_id (FK to User), account_number, balance,
currency.
- `Transaction`: id (UUID), account_id (FK to Account), amount, type
(credit/debit), timestamp, related_account_id (nullable).
- `Card`: id (UUID), account_id (FK to Account), card_number, cvc, expiry.
</database_schema>

<execution_steps>
1. **Self-Reflection & Architecture Mapping**: Briefly outline how the domain-
driven folders (`app/core`, `app/api`, `app/models`, `app/schemas`, `app/db`,
`tests`) will interact to fulfill a clean architecture.
2. **Artifact Generation**: Produce the `Implementation Plan` detailing the
exact folder tree, the configuration for `app/db/session.py` using SQLite, and
the explicit field types for the SQLAlchemy models.
</execution_steps>
```

Resulting AI Action: The agent generated an `Implementation Plan` with the correct domain-driven structure, but defaulted to using `float` for currency fields. **Human Review & Intervention:** I identified the `float` datatype as a critical financial anti-pattern and prepared a corrective prompt before allowing code generation.

4.1.2 Iteration 2: Schema Correction (Data Integrity)

Objective: Correct the database schema to handle financial data safely and initialize migrations.

The Prompt:

```
@Workspace

The implementation plan looks solid, but I am rejecting the use of `float` for
the currency fields. This is a critical financial system constraint.

**Required Plan Modifications:**
1. **Database Schema Fix:** Change the datatype for `balance` in the `Account`
model and `amount` in the `Transaction` model from `float` to `Integer`. We
will store all currency values as cents (the smallest currency unit) to avoid
```

```
floating-point precision errors. Pydantic will handle the conversion.
2. **Missing Component:** Ensure `alembic init alembic` is part of your
execution steps so the migration environment is scaffolded.
3. **AI Usage Report:** Add the creation of `AI_Usage_Report.md` to your plan.
You must log this specific intervention.
```

Resulting AI Action: The agent updated the plan, modifying the fields to `Integer` and adding the `alembic init` command. **Human Review & Intervention:** I explicitly overrode the AI's choice, enforcing the storage of currency as `Integer` (cents) to prevent floating-point precision errors natively inherent to SQLite.

4.1.3 Iteration 3: Project Organization & Version Control

Objective: Establish a clean Git baseline and organize project documentation correctly.

The Prompt:

```
@Workspace

We need to correct the project structure and establish our version control
baseline.

**Required Operational Changes:**
1. **Relocate Documentation:** Move `Future_Considerations.md` into a new
`docs/` folder at the root of the project.
2. **Relocate AI Report:** Move `AI_Usage_Report.md` to the root directory of
the project. We will not be tracking the `ai_brain/` folder in version control.
3. **Version Control Initialization & Push:** Execute git init, configure
.gitignore (ignoring ai_brain/, __pycache__/, .venv/, and *.db), commit the
initial scaffolding, and push to a new private repository on GitHub.
```

Resulting AI Action: The agent moved the markdown files out of its internal directories, initialized Git, configured `.gitignore`, and pushed to the remote repository. **Human Review & Intervention:** I forced the agent to move files out of its hidden `ai_brain/` folder to ensure the required assessment documents were properly visible in the source code deliverable.

4.1.4 Iteration 4: Architectural Audit & Verification

Objective: Verify the agent correctly implemented the domain-driven structure and the critical database type overrides before proceeding to business logic.

The Prompt:

```
@Workspace

We are pausing to perform an architectural audit before moving to Phase 2.

**Action:** Please analyze the current workspace and provide a comprehensive
status report formatted in Markdown. Include the following:
```

```
1. **Directory Tree:** A visual tree of the current project structure
(excluding `.git`, virtual environments, and `__pycache__`).
2. **Database Verification:** Print the exact code currently written inside
`app/db/session.py` and `app/models/account.py`. I need to verify that the
SQLite configuration and the `Integer` datatype for currency were implemented
correctly.
3. **Documentation Check:** Confirm the exact file paths for
`AI_Usage_Report.md` and `Future_Considerations.md`.
```

Resulting AI Action: The agent output the directory tree and the contents of the database files, confirming successful setup of `aiosqlite` and the `Integer` override for the `balance` field.
Human Review & Intervention: The agent issued a warning that the `currency` field itself was still a string (`Mapped[str]`) and suggested changing it to an integer. I manually rejected this suggestion to strictly adhere to the ISO 4217 3-letter string standard (e.g., "USD", "EUR") which is the industry best practice for JSON API payloads.

4.2 Phase 2: Core Business Logic & Atomic Transfers

Phase Objective: Define strict Pydantic DTO schemas and implement the isolated Service Layer for atomic money transfers.

4.2.1 Iteration 1: Service Layer Architecture & Transaction Scoping

Objective: Design the business logic for `transfer_funds` ensuring ACID compliance and proper validation.

The Prompt:

```
@Workspace /plan

<system_constraints>
- Role: Senior Backend Architect.
- Tech Stack: Python 3.12+, FastAPI, SQLAlchemy 2.0 (Async), Pydantic v2.
- Strict Rule 1: DO NOT write any code or create files yet. Generate an
`Implementation Plan` artifact first.
- Strict Rule 2: Business logic MUST be isolated in an `app/services/` layer
and must NOT be tightly coupled to the FastAPI routers.
- Strict Rule 3: The Money Transfer operation MUST utilize explicit
asynchronous database transaction blocks (`await session.commit()` and `await
session.rollback()`) to guarantee atomicity.
</system_constraints>

<context>
We are moving to Phase 2 of the production-ready Banking REST Service.
The objective is to define the Pydantic DTO schemas for strict data validation
and implement the core business logic (Service Layer) for handling accounts and
money transfers.
```

```
</context>

<execution_steps>
1. **Schema Design (`app/schemas/`):** Define the Pydantic v2 schemas for
`User`, `Account`, and `Transaction`. Ensure you have separate schemas for
input creation (e.g., `AccountCreate`) and output responses (e.g.,
`AccountResponse` utilizing `model_config = ConfigDict(from_attributes=True)`).
2. **Service Layer Design (`app/services/`):** - Outline an `AccountService`
for basic ledger queries.
   - Outline a `TransferService` with an async method
`transfer_funds(from_account_id: UUID, to_account_id: UUID, amount: int,
session: AsyncSession)`.
   - Explicitly detail the logical flow of `transfer_funds`: How it will check
sufficient balances, update both account balances, create two offsetting
`Transaction` records (one debit, one credit), and safely wrap everything in a
try/except block with rollback.
3. **Artifact Generation:** Produce the `Implementation Plan` detailing the
Pydantic fields and the exact pseudocode or logical flow of the
`transfer_funds` service.
</execution_steps>
```

Resulting AI Action: The agent produced an implementation plan with the requested ACID transaction block (`session.commit()` and `session.rollback()`). However, it incorrectly attempted to use PostgreSQL row-level locking (`.with_for_update()`) on an SQLite database, missed `.scalar_one_or_none()` syntax, and omitted negative amount validations. **Human Review & Intervention:** I rejected the plan. I instructed the agent to remove the incompatible locking mechanism, correct the SQLAlchemy 2.0 syntax, and implement strict validations for negative amounts and same-account transfers.

The Prompt:

```
@Workspace

I am rejecting this Implementation Plan. You have made a few critical errors
regarding SQLite compatibility and SQLAlchemy 2.0 syntax, as well as missing
some basic financial validations.

**Required Plan Modifications:**

1. **SQLite Compatibility:** Remove `.with_for_update()`. SQLite does not
support row-level locking. Rely on standard transaction isolation for this MVP.
2. **SQLAlchemy Syntax:** `await session.execute(select(...))` returns a Result
object, not the model. You MUST append `.scalar_one_or_none()` to extract the
actual `Account` object before checking balances.
3. **Business Validations:** At the very top of `transfer_funds`, add
validation checks to raise a ValueError if:
   - `amount <= 0` (Transfers must be strictly positive).
   - `from_account_id == to_account_id` (Cannot transfer to the same account).
4. **Error Handling:** If either account is not found (`None`), raise a
ValueError("Account not found") before attempting to check balances.

**Action:** Update the Implementation Plan to reflect these changes. Once
```

```
updated, you have my approval to **EXECUTE** the plan and generate the actual
Python files in `app/schemas/` and `app/services/`.
```

4.2.2 Iteration 2: Architectural Audit & Verification

Objective: Audit the generated Python code to ensure strict ACID transaction compliance and SQLite compatibility before building the external API layer.

The Prompt:

```
@Workspace

We are pausing to perform an architectural audit of Phase 2 before moving to
Phase 3.

**Action:** Please read the files you just generated and provide the following
for my review:
1. **Transfer Logic:** Print the exact, complete code for the `transfer_funds`
method inside `app/services/transfer_service.py`. I need to verify the
`try/except` block, the SQLAlchemy `.scalar_one_or_none()` syntax, and the
rollback mechanism.
2. **Schema Verification:** Print the exact code for the `AccountResponse`
class in `app/schemas/account.py`. I need to verify that Pydantic v2
`ConfigDict(from_attributes=True)` was implemented correctly for ORM parsing.
```

Resulting AI Action: The agent retrieved the exact implementation of the `transfer_funds` method and `AccountResponse` schema. The code correctly utilized `.scalar_one_or_none()` and implemented a robust `try...except` block with explicit `await session.rollback()`.
Human Review & Intervention: I manually reviewed the business logic and verified that all guardrails (preventing negative transfers, preventing same-account transfers, and ensuring atomicity) were securely in place. The core logic is certified production-ready, allowing us to safely proceed to the API routing phase.

4.3 Phase 3: The API Layer & JWT Authentication

Phase Objective: Implement secure JWT authentication and expose the core business logic via strictly isolated FastAPI routers.

4.3.1 Iteration 1: Security Architecture & API Routing Scope

Objective: Design the JWT security dependency and map the required service endpoints to FastAPI routers.

The Prompt:

```
@Workspace /plan
```

```
<system_constraints>
- Role: Senior Backend Architect.
- Tech Stack: Python 3.12+, FastAPI, passlib (bcrypt), PyJWT.
- Strict Rule 1: DO NOT write any code or create files yet. Generate an
`Implementation Plan` artifact first.
- Strict Rule 2: Ensure strictly typed FastAPI Dependency Injection (`Depends`)
is used for database sessions and the `get_current_user` security mechanism.
- Strict Rule 3: Do NOT put all endpoints in `main.py`. Use `APIRouter` to
maintain domain segregation in `app/api/routers/`.
</system_constraints>

<context>
We are moving to Phase 3 of the Banking REST Service.
The objective is to implement the Security layer (JWT Authentication) and wire
up the FastAPI routers to expose the Services built in Phase 2.
</context>

<execution_steps>
1. **Security Layer (`app/core/security.py`):** Outline the setup for password
hashing (e.g., passlib with bcrypt) and JWT token generation/decoding using
`PyJWT`. Define a `get_current_user` FastAPI dependency that extracts the user
ID from the JWT.
2. **Auth Router (`app/api/routers/auth.py`):** Outline the `/signup` endpoint
(creating a User) and the `/login` endpoint (returning an access token using
FastAPI's `OAuth2PasswordRequestForm`).
3. **Business Routers (`app/api/routers/accounts.py` & `transfers.py`):**
Detail how you will use `APIRouter` to expose endpoints like `POST /accounts/`,
`GET /accounts/{id}`, and `POST /transfers/`. Explicitly state how
`get_current_user` will be injected to ensure users can only access their own
accounts.
4. **Main Application (`app/main.py`):** Outline the initialization of the
FastAPI app and the inclusion of these routers.
5. **Artifact Generation:** Produce the `Implementation Plan` detailing this
exact routing and security structure.
</execution_steps>
```

Resulting AI Action: The agent correctly implemented the `OAuth2PasswordBearer` dependency and perfectly scoped the IDOR protection logic (verifying `account.user_id == current_user.id`). However, it missed several key deliverables from the original specification (Cards, Statements, and Transaction history). **Human Review & Intervention:** I approved the security logic but rejected the scope. I instructed the agent to expand the routing plan to include the missing `/cards`, `/transactions`, and `/statements` endpoints before authorizing code generation.

The Prompt:

```
@Workspace

This Implementation Plan is excellent in terms of security. The authorization
check ensuring `from_account.user_id == current_user.id` is exactly what I was
looking for to prevent IDOR vulnerabilities.

However, to fully satisfy the project requirements, we need to expose the
```

```
remaining service interfaces.

**Required Plan Modifications:**
1. **Missing Endpoints:** Add router outlines for the following required
features to your Business Routers section:
   - `Transactions`: Endpoint to get the transaction ledger for an account
(e.g., `GET /accounts/{id}/transactions`). Ensure it verifies account
ownership.
   - `Cards`: Endpoints to issue a debit/credit card to an account (`POST
/cards/`) and view user cards (`GET /cards/`). Must enforce that the user owns
the associated account.
   - `Statements`: An endpoint to generate or retrieve a monthly statement
summary for an account (e.g., `GET /accounts/{id}/statement`).
2. **AI Usage Report:** Do not forget to log this intervention.

**Action:** Update the Implementation Plan to include these missing endpoints.
Once updated, you have my approval to **EXECUTE** the plan and generate all the
Phase 3 security and router files, and wire them into `main.py`.
```

4.3.2 Iteration 2: Architectural Audit & Verification

Objective: Verify the JWT security implementation and IDOR (Insecure Direct Object Reference) protections in the generated API routers before local execution.

The Prompt:

```
@Workspace

We are pausing to perform an architectural audit of Phase 3 before moving on. I
need to verify the security implementations and router wiring.

**Action:** Please read the files you just generated and provide the exact code
for the following:
1. **The Security Dependency:** Print the `get_current_user` function from
`app/core/security.py`. I need to verify how the JWT is decoded and how the
user is fetched.
2. **The IDOR Protection:** Print the `create_transfer` (or equivalent POST)
endpoint from `app/api/routers/transfers.py`. I must verify the exact line of
code that checks `from_account.user_id == current_user.id` before the
`TransferService` is called.
3. **The Application Entrypoint:** Print the `app/main.py` file to verify that
all 6 routers (auth, accounts, transfers, transactions, cards, statements) are
successfully included in the FastAPI app instance.
```

Resulting AI Action: The agent output the requested security functions and endpoint logic. The `get_current_user` dependency correctly decodes the JWT and queries the active user. The `create_transfer` endpoint explicitly enforces `from_account.user_id == current_user.id` prior to invoking the transactional service layer. All six required domain routers were successfully registered in `main.py`. **Human Review & Intervention:** I manually audited the authentication flow and IDOR guardrails. The security implementation meets production standards, ensuring users cannot access or transfer funds from unauthorized accounts. The backend is now cleared for local execution and frontend integration.

4.3.3 Iteration 3: Usability & Proactive System Polish

Objective: Improve API usability by adding a root health-check endpoint and proactively auditing the codebase for production-grade scaling issues (like missing pagination).

The Prompt:

```
*(Manual Developer Intervention)*: I manually added a standard `GET /` endpoint
to `main.py` returning a basic status payload `{"message": "Banking REST
Service API is running"}` to improve developer experience and provide an
immediate visual confirmation of server health before users navigate to
`/docs`.
```

The Prompt (Architectural Audit):

```
@Workspace

The application is successfully running locally, and I have manually added a
health check on the `/` endpoint. Before we move to testing and Dockerization,
I want to perform a proactive code quality audit to ensure this meets Senior-
level production standards.

**Action:**
1. **Audit:** Review the existing routers (`accounts.py`, `transfers.py`,
`transactions.py`, etc.).
2. **Improvement 1 (Pagination):** Identify any `GET` endpoints that return
lists (like getting all transactions or all cards for a user). Add `limit: int
= 100` and `offset: int = 0` query parameters to paginate the database queries
and prevent memory overloads.
3. **Improvement 2 (Standardized Errors):** Ensure that any custom exceptions
raised in the `app/services/` layer are properly caught in the routers and
returned as standard FastAPI `HTTPException` objects with clear, consistent
JSON details.

Implement these refinements directly in the code and provide a brief summary of
the files you modified.
```

Resulting AI Action: The agent audited the list-based endpoints (transactions, cards) and successfully implemented `limit` and `offset` pagination to prevent payload overloading. It also standardized the try/except blocks across the routers to ensure consistent JSON error shapes.
Human Review & Intervention: I reviewed the pagination implementation to ensure default limits were reasonable (100) and that the database queries correctly utilized SQLAlchemy's `.limit()` and `.offset()` methods. The API is now highly resilient and ready for the next phase.

4.3.4 Iteration 4: Async Database Migration & State Management

Objective: Configure Alembic for asynchronous SQLAlchemy execution and apply the initial schema to the SQLite database.

The Prompt:

```
@Workspace
```

```
We are ready to bring the database to life. I need you to configure Alembic,
generate the initial migration, and apply it to create our SQLite database
tables.

**Action:**
1. **Configure `alembic.ini`:** Ensure the SQLAlchemy URL is correctly pointing
to our local async SQLite database (e.g., `sqlalchemy.url =
sqlite+aiosqlite:///./banking.db`).
2. **Configure `alembic/env.py`:** Update the environment file to import our
SQLAlchemy `Base` and all of our models (`User`, `Account`, `Transaction`,
etc.) so Alembic can detect them. Set `target_metadata = Base.metadata`. Make
sure it is configured for async execution if necessary.
3. **Execute Migration Generation:** Run the terminal command to generate the
migration script: `alembic revision --autogenerate -m "initial_schema"`
4. **Execute Migration Upgrade:** Run the terminal command to apply the schema
to the database: `alembic upgrade head`

**Output:** Please execute these steps, automatically fix any
import/configuration errors that arise during the `autogenerate` process, and
confirm once the `.db` file has been successfully created with all our tables.
```

Resulting AI Action: The agent correctly configured `alembic.ini` and `env.py` for async execution via `create_async_engine`. It identified a missing dependency (`greenlet`) required for async Alembic operations, installed it, updated `requirements.txt`, and successfully generated and applied the initial migration to create `banking.db`. **Human Review & Intervention:** I verified that the database file was successfully created in the root directory and that the schema changes were securely tracked in version control before starting the local server for end-to-end manual testing.

4.3.5 Iteration 5: Repository Hygiene & Security

Objective: Enforce industry-standard project organization by removing generated database artifacts from the project root and securing version control.

The Prompt:

```
@Workspace

The user correctly pointed out that leaving `banking.db` in the project root is
bad practice. We need to reorganize the workspace to meet industry standards.

**Action:**
1. **Create Directory:** Create a new directory at the root level called
`data/`.
2. **Move Database:** Move `banking.db` into the `data/` directory.
3. **Update Connections:** - Update `alembic.ini` to point to the new location
(e.g., `sqlalchemy.url = sqlite+aiosqlite:///./data/banking.db`).
   - Update the application's database connection string (likely in
`app/db/session.py` or `app/core/config.py`) to point to
`sqlite+aiosqlite:///./data/banking.db`.
4. **Secure Version Control:** Check the `.gitignore` file. Ensure that `data/`
and `*.db` are explicitly ignored so we never commit local database files to
the repository.
```

```
5. **Verify:** Run a quick python script to connect to `./data/banking.db` to
ensure the application can still successfully reach the tables.

Execute this reorganization and confirm once the app is successfully re-wired
to the new data directory.
```

Resulting AI Action: The agent created a `data/` directory, migrated the `banking.db` file, and successfully updated the connection strings in both `alembic.ini` and the application's session configuration. It also updated the `.gitignore` to strictly exclude the `data/` directory. **Human Review & Intervention:** I initiated this reorganization because the AI initially placed the `.db` file in the project root. Leaving data artifacts in the root is an anti-pattern that clutters the workspace and risks accidental commits of sensitive database files. The project structure is now secure and clean.

4.4 Phase 4: Testing, Observability, & Advanced Health Checks

Phase Objective: Fulfill the strict requirements for comprehensive test coverage, structured logging, and robust lifecycle management before containerization.

4.4.1 Iteration 1: Test Architecture & Observability Planning

Objective: Design a test suite that isolates database state and implement structured JSON logging for production monitoring.

The Prompt:

```
@Workspace

We need to perform a strict gap analysis of the current codebase against our
project requirements. Do not write any code yet. Just audit the workspace and
confirm what is missing.

**Context: The Missing Requirements**
1. **Test Suite:** We need comprehensive test coverage, including unit tests
for business logic (`app/services/`) and integration tests for API endpoints
(`app/api/routers/`).
2. **Logging & Monitoring:** We need structured logging (JSON format
preferred), defined log levels, and error tracking.
3. **Health Checks:** We need a robust `/health` endpoint that performs
database connectivity checks and service readiness probes. We also need
graceful shutdown handling in the FastAPI lifecycle.
4. **Containerization:** We need a `Dockerfile` and `docker-compose.yml`.

**Action:**
1. Scan the repository and confirm the absence of these four components.
2. Generate an `Implementation Plan` for Phase 4, focusing strictly on
fulfilling **Test Suite**, **Logging**, and **Advanced Health Checks**. Leave
Containerization for Phase 5.
```

```
3. In the plan, outline the exact folder structure for the tests (e.g.,
`tests/unit/`, `tests/integration/`) and how you will configure `pytest` with
`pytest-asyncio` to test our async database sessions.
```

Resulting AI Action: The agent correctly identified the missing components and drafted an implementation plan. The plan proposed `pytest` with an in-memory SQLite database (`sqlite+aiosqlite:///:memory:`) for strict test isolation, `python-json-logger` for structured observability, and FastAPI `@asynccontextmanager` for graceful database connection pooling and teardown. **Human Review & Intervention:** I audited the proposed architecture. The use of an in-memory database for testing prevents side effects on the local development database, which is a senior-level best practice. I approved the plan for execution without modification.

4.4.2 Iteration 2: Test Scope Refinement & Execution

Objective: Approve the Phase 4 architecture while enforcing comprehensive integration test coverage across all domain routers.

The Prompt:

```
*(I provided a prompt approving the in-memory database and JSON logging
strategy, but explicitly instructed the agent to expand its test generation to
include `test_accounts.py`, `test_transactions.py`, `test_cards.py`, and
`test_statements.py` to ensure full coverage).*
```

Resulting AI Action: The agent installed the testing dependencies, scaffolded the `conftest.py` fixtures, and generated the comprehensive test suite spanning both the `services/` and `routers/` layers. It also implemented the JSON logging middleware and the deep `/health` database probe. **Human Review & Intervention:** I manually audited the AI's implementation plan before allowing it to write code. I noticed the AI had minimized the integration testing scope to just Auth and Transfers. I intervened to force the generation of tests for all remaining endpoints, ensuring compliance with the assessment's strict testing requirements.

4.4.3 Iteration 3: Test Coverage & Negative Scenario Auditing

Objective: Move beyond "happy path" testing by enforcing strict code coverage metrics and explicitly testing negative/boundary edge cases.

The Prompt:

```
@Workspace

The user correctly pointed out that a 100% pass rate does not mean the test
suite is sufficient. We need to ensure we are heavily testing "sad paths" and
negative scenarios, not just "happy paths."

**Action:**
1. **Coverage Analysis:** - Install `pytest-cov` and update `requirements.txt`.
   - Run the test suite with coverage: `pytest --cov=app tests/`
   - Report the total coverage percentage. Identify which files in
`app/services/` or `app/api/routers/` have the lowest coverage.
2. **Negative Scenario Audit:** Review the existing tests in `tests/` and
```

```
explicitly tell me if we are currently testing the following negative
scenarios:
   - Transferring a negative amount (e.g., -$100).
   - Transferring exactly $0.
   - Accessing endpoints with an invalid or expired JWT token.
   - Attempting to fetch a paginated list with an absurdly high `limit` (e.g.,
limit=100000).
3. **Execution:** If any of the above negative scenarios are missing, write the
tests for them now, add them to the suite, and re-run the coverage report.

**Output:** Provide the coverage report summary before and after your
additions, and list the exact negative tests you added.
```

Resulting AI Action: The AI installed `pytest-cov` and generated a coverage report. The audit revealed that while IDOR and insufficient funds were tested, boundary attacks (negative transfer amounts, zero amounts) and malformed JWT scenarios were missing. The agent generated the missing negative tests and re-ran the suite. **Human Review & Intervention:** I initiated this audit because AI agents natively default to writing "happy path" tests. By forcing a coverage report and explicitly injecting boundary-value testing parameters, I ensured the suite evaluates the API's resilience against malicious or malformed client payloads.

4.4.4 Iteration 4: Test Suite Restructuring & Final Audit

Objective: Improve repository hygiene by managing coverage artifacts, applying industry-standard naming conventions to the test directories, and performing a final gap analysis on test coverage.

The Prompt:

```
*(Manual Developer Intervention)*: I directed the AI to move the default
`.coverage` artifact out of the project root and configure it to output to a
dedicated, git-ignored `tests/coverage/` directory. I also instructed the AI to
rename `tests/integration/` to `tests/api/` for clearer domain boundary
separation, and forced a final exhaustive mapping of all endpoints and services
to ensure no edge cases were left untested.
```

Resulting AI Action: The agent configured `pytest.ini` to manage the coverage output, renamed the integration directory to `tests/api/`, and re-ran the coverage suite to verify all positive and negative permutations for the routers and services were accounted for. **Human Review & Intervention:** I initiated this because default testing tools often clutter the root directory. Clean workspace management and explicit folder naming (`tests/api/` instead of generic `integration/`) demonstrate a mature approach to maintainable project architecture.

4.5 Phase 5: Containerization & Deployment Architecture

Phase Objective: Package the application into an isolated, reproducible environment using Docker, fulfilling the core containerization requirements and securing the multi-stage build bonus points.

4.5.1 Iteration 1: Multi-Stage Dockerfile & Compose Setup

Objective: Generate a secure, optimized Dockerfile and a docker-compose configuration for local development.

The Prompt:

```
@Workspace

We have successfully completed Phase 4. Our test suite passes perfectly. We are
now moving to Phase 5: Containerization.

**Context: Assessment Requirements**
We must implement a Dockerfile, a docker-compose.yml, environment variable
configuration, and secure the "Bonus Points" for using a multi-stage build.

**Action:**
1. **`Dockerfile`**: Create a multi-stage Dockerfile.
   - **Stage 1 (Builder):** Use an official Python 3.12 slim image. Install
build dependencies, copy `requirements.txt`, and install the Python packages
into a virtual environment or specific directory.
   - **Stage 2 (Runner):** Use a clean Python 3.12 slim image. Copy the built
dependencies from Stage 1. Copy the `app/` directory. Expose port 8000. Set the
`CMD` to run the uvicorn server.
2. **`.dockerignore`**: Create this file to explicitly exclude `.venv`,
`__pycache__`, `tests/`, `data/`, and `.git/` to keep the image lean.
3. **`docker-compose.yml`**: Create a compose file defining the `api` service.
Map port 8000 to 8000. Inject environment variables (e.g.,
`DATABASE_URL=sqlite+aiosqlite:///./data/banking.db`). Map a volume for the
`./data` directory so the SQLite database persists across container restarts.

**Output:** Generate these three files and provide a summary of your
implementation.
```

Resulting AI Action: The agent correctly generated a multi-stage `Dockerfile` utilizing `python:3.12-slim`, effectively separating the build dependencies from the final runtime image to secure the multi-stage build bonus point. It also produced the `.dockerignore` and `docker-compose.yml` with the requested volume mappings and environment variables.

4.5.2 Iteration 2: Docker Verification & Execution

Objective: Execute the generated containerization strategy to ensure environment reproducibility.

The Prompt:

```
*(Manual Developer Intervention)*: I directed the AI to generate the multi-
stage `Dockerfile`, `.dockerignore`, and `docker-compose.yml`. I specifically
enforced strict volume mapping for the SQLite database (`./data:/app/data`) to
ensure data persistence across container lifecycles, and excluded coverage
artifacts from the build context.
```

Resulting AI Action: The agent successfully generated the Docker configuration files, utilizing `python:3.12-slim` for both the builder and runner stages. It properly configured `PYTHONDONTWRITEBYTECODE=1` and exposed port 8000. **Human Review & Intervention:** I manually reviewed the `.dockerignore` to ensure sensitive/unnecessary directories (`.git/`, `tests/`) were excluded. I then ran `docker-compose up --build` to verify the image compiled successfully and the FastAPI application booted correctly within the isolated container.

4.5.3 Iteration 3: End-to-End Manual QA & Validation

Objective: Perform manual, in-container testing to verify data persistence, authentication state, and business logic validation boundaries.

The Prompt:

```
*(Manual Developer Intervention)*: I executed a strict manual QA protocol via
the Swagger UI (`http://localhost:8000/docs`) against the running Docker
container.


1. **Persistence & Auth:** I created a user, destroyed the container (`docker-
compose down`), rebooted it, and successfully authenticated, proving the
`./data` volume mapping was flawless.

2. **Validation Funnel:** I tested the transaction endpoints by attempting to
transfer funds to a non-existent account (yielding a proper `404 Account Not
Found`), and then attempting an overdraft transfer to a valid account (yielding
a proper `400 Insufficient Funds`).
```

Human Review & Intervention: This manual intervention was critical to verify that the theoretical Pytest coverage translated perfectly into the live containerized environment. The backend successfully defended against malformed state and unauthorized overdrafts.

4.5.4 Iteration 4: Database Seeding & Synthetic Data Generation

Objective: Populate the SQLite database with realistic synthetic data to facilitate frontend UI development and visual QA.

The Prompt:

```
*(Manual Developer Intervention)*: I directed the AI to create a `seed_data.py`
script to bypass the API and inject a robust dataset directly into the
database. I required a destructive wipe (`DELETE FROM`) for idempotency,
followed by the generation of multiple users, funded checking/savings accounts,
dynamically generated debit cards, and a web of historical transfers.
```

Resulting AI Action: The agent generated and executed an advanced seeder script that successfully initialized 5 users, 10 accounts (funded between $5k-$50k), 20 matrixed transfers, and 8 debit cards. **Human Review & Intervention:** This step was critical to ensure the Phase 6 React frontend has a rich, realistic state to render upon initialization, proving the backend's relational mapping works perfectly at scale.

4.6 Phase 6: Frontend UI Integration (React)

Phase Objective: Develop a modular, consumer-facing Single Page Application (SPA) using React and Tailwind CSS to interface with the FastAPI backend.

4.6.1 Iteration 1: Architecture Blueprint, Scaffolding & Networking

Objective: Define the component hierarchy, state management strategy, scaffold the Vite environment, and implement JWT authentication global state with Axios HTTP interceptors.

The Prompt:

```
*(Manual Developer Intervention)*: I enforced strict project management
protocols, commanding the AI to document its proposed architecture in
`ai_brain/implementation_plan_phase6.md` before writing code. I then instructed
it to execute terminal commands to scaffold a Vite/React app and install
`axios`, `react-router-dom`, `lucide-react`, and `tailwindcss` without globally
installing packages. I furthermore directed the AI to establish the global
`AuthContext` with native JWT decoding, and create a centralized Axios instance
that automatically attaches the `Authorization: Bearer <token>` header from
`localStorage`.
```

Resulting AI Action: The agent documented a strict architecture utilizing Context API for global state and Axios for interceptors. It successfully executed the scaffolding commands within the terminal, configured the environment, and implemented a robust `AuthContext` that natively decodes the base64 JWT payload to extract expiration and user identifiers without relying on external bloatware libraries. **Human Review & Intervention:** I mandated this workflow because jumping straight into UI code without a defined state/networking strategy usually results in fragmented, unscalable React applications.

4.6.2 Iteration 2: CORS, UI/UX Refinement & Docker State Resolution

Objective: Resolve browser-level security blocks (CORS), fix 422 Unprocessable Entity errors during dashboard initialization, update application metadata, and enhance the authentication interface UX.

The Prompt:

```
*(Manual Developer Intervention)*: I provided console logs showing a CORS error
and a 422 error on `GET /accounts/me`. I directed the AI to:


1. Update `app/main.py` CORS middleware to explicitly allow
`http://localhost:5173` and `127.0.0.1:5173`.
```

```
2. Refactor `Login.jsx` to use a Flexbox container to fix the password toggle
icon disappearing on focus.

3. Audit backend routes and Axios headers to resolve the 422 error, updating
code and `index.html` metadata.
```

Resulting AI Action: The agent successfully modified `app/main.py` with expanded CORS origins to cover both localhost and 127.0.0.1. It also refactored the `Login.jsx` component using a robust Flexbox wrapper with focus-within styling. Additionally, it identified the 422 error root cause as Docker Volume caching (the container was running a stale image), executed a hard container rebuild (`docker-compose up -d --build`), and updated the HTML metadata. **Human Review & Intervention:** I diagnosed the origin mismatch and schema validation failure in the browser console. By directing the specific CORS fix and identifying the missing execution context (stale container state), I allowed the AI to solve the integration bugs, enabling the dynamic seeded data to securely render on the dashboard.

4.6.3 Iteration 3: User Journey Expansion - Signup & Navigation

Objective: Implement a complete end-to-end user onboarding flow with secure registration and global navigation.

The Prompt:

```
@Workspace


The user correctly pointed out that our frontend lacks a complete user journey.
We need to implement a Signup flow and a proper Navigation header.


**Action 1: Implement Signup (`src/pages/Signup.jsx`)**


1. Create a registration form taking `email` and `password`.

2. On submit, use Axios to make a `POST` request to our `/auth/signup` endpoint
with the JSON payload.

3. On success, show a success message and use React Router's `useNavigate` to
redirect the user to `/login`.

4. Add a "Don't have an account? Sign up" link on the `Login.jsx` page, and an
"Already have an account? Log in" link on the `Signup.jsx` page.


**Action 2: Update Routing (`src/App.jsx`)**
```

```
1. Add the new `/signup` route to the React Router configuration.


**Action 3: Global Navigation (`src/components/Navigation.jsx`)**


1. Ensure the Navigation bar appears at the top of the Dashboard.
2. It should display the "Banking Service" logo/text on the left.
3. It should display the current user's email (from AuthContext) and a "Logout"
button on the right.
```

Resulting AI Action: The agent built the `Signup.jsx` component, wired it to the `/auth/signup` backend endpoint, updated React Router, and polished the `Navigation.jsx` header to display the decoded JWT user identifier alongside a functional logout mechanism. **Human Review & Intervention:** I identified that the frontend architecture lacked a complete product journey. I directed the AI to expand the scope beyond a simple static dashboard by implementing a dedicated registration flow and global navigation. This ensures a production-ready user experience that can be tested end-to-end without relying solely on backend seeder scripts.

4.6.4 Iteration 4: Statement Generator Implementation

Objective: Implement the final interactive UI feature to allow users to fetch and view their monthly ledger summaries.

The Prompt:

```
@Workspace

The Transfer form and Transaction ledger are now mathematically and visually
perfect. We need to implement the final interactive feature: The Statement
Generator.

**Action 1: Implement `src/components/StatementGenerator.jsx`**

1. Create a clean, Tailwind-styled UI card for generating monthly statements.

2. Add a button (e.g., "View Monthly Statement"). Include a loading state while
fetching.

3. When clicked, use our Axios instance to make a `GET` request to the
statement endpoint.

4. Display the resulting data in an elegant layout below the button. Show the
`starting_balance`, `ending_balance`, `total_credits`, and `total_debits`.


**Action 2: Integrate into `src/pages/Dashboard.jsx`**
```

```
1. Import the new `StatementGenerator` component and place it thoughtfully in
the Dashboard layout.

2. Ensure you pass the active `accountId` as a prop.
```

Resulting AI Action: The agent built the `StatementGenerator.jsx` component, successfully wired it to the `/statements` API endpoint using the established Axios instance, and integrated it into the Dashboard's grid layout. **Human Review & Intervention:** I directed the AI to build this final component to ensure 100% feature parity with the backend API built in Phase 3.

4.7 Phase 7: Deployment, Robustness & Polish

Phase Objective: Finalize Docker configurations, implement defensive deployment scripts, and resolve production network issues.

4.7.1 Iteration 1: Production Dockerization & Bootstrapping

Objective: Unify the frontend into a production-grade Nginx container and provide OS-agnostic startup scripts for a one-click deployment.

The Prompt:

```
@Workspace

We are finalizing the application for a Senior Engineering assessment
submission. We need to deploy a built, production-ready version of the frontend
and provide two highly robust entry points for the grader.

**Action 1: Production Frontend Dockerfile**

1. Create `frontend/Dockerfile` using a multi-stage build (Node alpine for
build, Nginx alpine for serving).

**Action 2: Update Docker Compose**

1. Update `docker-compose.yml` to include the `frontend` service building from
`./frontend`. Map port 8080:80.

**Action 3: Interactive Start Script (`start.sh`)**

1. Create a robust Bash script that checks for Docker, runs `docker-compose up
-d --build`, asks the user if they want to seed the database, and prints the
final URLs.
```

Resulting AI Action: The agent successfully created the multi-stage frontend Dockerfile, updated the orchestrator, and generated the interactive `start.sh` and `start.ps1` bootstrap scripts. **Human Review & Intervention:** I directed this to replace the manual `npm run dev` process with a true

production build, ensuring the grader receives a professional, isolated runtime environment that does not require local Node.js installations.

4.7.2 Iteration 2: Defensive Scripting & Seeder Versioning

Objective: Prevent fatal deployment crashes when synthetic data scripts are missing and track the seeder in version control.

The Prompt:

```
@Workspace

We need to make our deployment robust. We need to track the seeder script in
Git and defensively update our startup scripts.

**Action 1: Version Control for seed_data.py**

1. Audit the `.gitignore` file. Ensure `seed_data.py` is NOT being ignored.
Stage it for version control.

**Action 2: Make `start.sh` and `start.ps1` Robust**

1. Inject a defensive file-existence check before running the Docker execution
command for the seeder.

2. If the file DOES NOT exist, `echo` a clear warning message and allow the
script to gracefully continue without exiting with an error code.
```

Resulting AI Action: The agent updated `.gitignore`, staged the seeder file, and wrapped the `docker-compose exec` seeding commands inside defensive `if [ -f "seed_data.py" ]` blocks. **Human Review & Intervention:** I performed a "Stranger Test" by cloning the repo to a fresh machine and discovered the bootstrap script crashed because it blindly assumed the presence of ignored files. I intervened to implement defensive Bash programming to ensure graceful degradation.

4.7.3 Iteration 3: Production Network Configuration & CORS

Objective: Resolve cross-origin blocks caused by migrating the frontend to a production Docker port.

The Prompt:

```
@Workspace

We have a cross-origin (CORS) blockade on fresh environments because our
production Docker deployment changed the frontend port, but our backend
whitelist wasn't updated to match.

**Action: Update CORS Whitelist (Backend)**


1. Update `app/main.py` CORS middleware `allow_origins` to explicitly include
the new production Docker ports (`http://localhost:8080`,
`http://127.0.0.1:8080`).
```

```
**Action: Verify Axios URL (Frontend)**

1. Open `src/api/axios.js` and ensure the `baseURL` points to the backend
Docker port (`http://localhost:8000`).
```

Resulting AI Action: The agent updated the FastAPI CORS whitelist to accept traffic from the Nginx container port and verified the frontend Axios routing. **Human Review & Intervention:** During my clean-environment QA testing, I identified a severe network failure preventing login. I diagnosed the issue as a CORS preflight failure caused by moving the frontend from Vite's dev port (5173) to Nginx's production port (8080) and directed the AI to patch the backend security policies accordingly.