# Migration Strategy

### 1. Project Overview & Code Review

#### Project Structure and Architecture

* The existing project follows a simple structure common to Flask applications:
* `app.py`: Contains application routing and business logic.
* `database.py`: Manages MySQL database connections and operations.
* `static/`: Holds CSS style sheets.
* `templates/`: Contains the HTML templates rendered by Flask.
* `feedback\_data.sql`: Used to insert sample data into the MySQL database.

#### Languages, Frameworks, and Dependencies

* \*\*Languages\*\*: Python, HTML, CSS
* \*\*Frameworks\*\*: Flask
* \*\*Database\*\*: MySQL
* \*\*Dependencies\*\*:
* Flask==2.1.1
* Flask-MySQLdb==0.2.0
* mysqlclient==2.1.1

#### Deprecated Technologies, Security Risks, and Technical Debt

* \*\*Deprecated Technologies\*\*: None explicitly noted, Flask and MySQL are still widely used.
* \*\*Security Risks\*\*:
* User inputs are not validated against SQL injection, as raw user input is being directly used in SQL queries.
* \*\*Technical Debt\*\*:
* Lack of separation of concerns (business logic intermingled with app routing).
* No tests present in the codebase to ensure functionality.

### 2. Feasibility Analysis & Metrics

#### Code Complexity

* \*\*Cyclomatic Complexity\*\*:
* The current functions have low complexity, with a direct correlation between the number of paths through the program logic in `app.py` and `database.py`.
* \*\*Maintainability Index\*\*:
* Reasonably maintainable due to its simplicity.

#### Coupling, Cohesion, Redundancy, and Duplication

* \*\*Coupling\*\*: Low – Each component is relatively independent.
* \*\*Cohesion\*\*: Moderate – Functions in `database.py` correspond closely to database operations, but application logic overlaps in `app.py`.
* \*\*Redundancy and Duplication\*\*: No significant duplication found.

#### Dependency Risks, Outdated Libraries, and Compatibility Issues

* All dependencies are modern, but using Flask-MySQLdb may risk future maintainability since direct database integrations may become obsolete with ORM solutions like SQLAlchemy.

#### Database Migration Complexity

* Migration complexity is low, as the schema is straightforward and data can be easily migrated to a new system.

#### External Service & API Dependencies

* No external APIs or services are utilized in the current implementation.

#### Test Coverage and Gaps

* No unit or integration tests are in place, creating a significant gap in test coverage, which could lead to issues during migration.

### 3. Migration Strategy Proposal

#### Step-by-Step Migration Strategy

1. **Install FastAPI and Required Libraries**:

* Adapt to using `databases` and `sqlalchemy` for database integration.

2. **Convert Routing**:

* Transition routes from Flask to FastAPI.

3. **Implement Dependency Injection**:

* Use FastAPI’s dependency injection system for managing database connections.

4. **Migrate Database Logic**:

* Refactor `database.py` to use SQLAlchemy with Async support if needed.

5. **Update Frontend Rendering**:

* FastAPI does not include a templating system but can serve static files.

6. **Confirm Functionality**:

* Test all endpoints using a dedicated test framework or library.

#### Migration Approach

* \*\*Refactor\*\*: Given the maintainability and moderate complexity, refactoring rather than a full rewrite fits best.

#### Recommended Tech Stacks

* \*\*FastAPI\*\*: As the new framework.
* \*\*SQLAlchemy\*\*: As the ORM to handle database interactions.
* \*\*MySQL or PostgreSQL\*\*: For the database.
* \*\*Pydantic\*\*: For data validation.

#### Phased Migration Plan

1. **Phase 1**: Set up FastAPI environment and configurations.

2. **Phase 2**: Convert the routing and implement basic endpoints.

3. **Phase 3**: Migrate database logic and integrate Pydantic models.

4. **Phase 4**: Rigorously test each component and update frontend.

5. **Phase 5**: Deploy and monitor for issues.

#### AI-Based Automation Tools for Efficiency

* \*\*Apache Airflow\*\*: For orchestrating data migration.
* \*\*SQLAlchemy-Migrate\*\*: For managing database migration scripts.
* \*\*pytest\*\*: For automated testing of the newly built FastAPI application.

### 4. Potential Challenges & Risk Mitigation

#### Key Roadblocks

* Lack of testing could lead to bugs not surfacing until later.
* Syntax and behavior shift between Flask and FastAPI could introduce errors.
* Database interaction patterns may need to change significantly.

#### AI-Driven Solutions for Risk Mitigation

* Utilize AI-based code analysis tools to identify areas possibly unintentionally modified.
* Implement an automatic testing framework with coverage reports to identify gaps.
* Use mocking tools to simulate database interactions during tests.

#### Areas Requiring Manual Intervention

* Manual creation and migration of SQL scripts to adapt to the new framework's requirements.
* User interface adjustments to ensure compatibility with FastAPI.

### 5. Estimated Effort & Resources

#### Time Estimate

* Estimated overall time: 4-8 weeks depending on feature complexity and testing requirements.

#### Cost Estimate

* Estimated costs will vary based on personnel and automation tools used, likely ranging from $5,000 to $15,000.

#### AI Agent Workload

* \*\*Fragmentor\*\*: 10% - Fragment each file for analysis and migration.
* \*\*Migrator\*\*: 60% - Actual migration of logic and features.
* \*\*Reviewer\*\*: 20% - Ensure quality and code correctness post-migration.
* \*\*Executor\*\*: 5% - Deploy to staging/production environments.
* \*\*Documentor\*\*: 5% - Ensure documentation aligns with new code structure.

#### AI Agent Roles and Expertise Required

* \*\*Fragmentor\*\*: Skilled in code organization and breakdown.
* \*\*Migrator\*\*: Proficiency in FastAPI, SQLAlchemy, async programming.
* \*\*Reviewer\*\*: Experienced in code reviews and bulk testing practices.
* \*\*Executor\*\*: Hands-on experience with deployment practices and CI/CD.
* \*\*Documentor\*\*: Expertise in technical writing and documentation.

Overall, this structured migration strategy provides a robust plan for shifting the legacy Flask application to FastAPI while addressing the complexities of the existing codebase.