Project A.S.E.T. Design Specification

Autonomous Software Engineering Team Version: 1.0

Architecture: Modular / Fragment-Based

1. Executive Summary

ASET is a Python-based framework that autonomously generates Streamlit applications. It utilizes a **Fragmentation Architecture** I came up with to break complex user requirements into isolated, context-light units (Fragments). This ensures high accuracy from LLMs by preventing context window overflow.

2. System Architecture

The system is composed of four distinct layers:

1. **The Brain:** An abstract interface for LLMs (Gemini, OpenAI, Local).
2. **The Protocol:** Strict Pydantic schemas that define data contracts.
3. **The Agents:** Specialized roles (Architect, Logic Coder, UI Composer,…).
4. **The Assembler:** File generation and dependency management.

3. Directory Structure

This structure separates core framework logic from the generated applications.

```
aset/
|— config.yaml          # API keys, Model selection (Gemini/Ollama), Max retries
|— main.py              # CLI Entry Point (e.g., python main.py "Build a CRM")
|— requirements.txt     # Dependencies (pydantic, google-generativeai, streamlit)
|
|— core/                # --- The Framework Kernel ---
|  |— __init__.py
|  |— llm.py            # Abstract Base Class for LLM Providers
|  |— state_manager.py  # Validates Pydantic schemas
|  └— sandbox.py        # Interfaces with E2B or Local Exec for testing
|
|— schemas/             # --- The Data Contracts ---
```

```
│   ├── __init__.py
│   ├── fragment.py        # LogicFragment & UIFragment definitions
│   └── project.py         # ProjectBlueprint & GlobalState definitions
│
├── agents/                # --- The AI Workers ---
│   ├── __init__.py
│   ├── architect.py       # Breaks prompt -> Blueprint (JSON)
│   ├── coder.py           # Blueprint -> Python Code (Logic)
│   ├── ui_composer.py     # Blueprint -> Streamlit Code (UI)
│   ├── debugger.py        # The "Self-Correction" Agent
│   └── # and any additional agents
├── templates/             # --- Prompt Engineering ---
│   ├── architect.md       # System prompt for decomposition
│   ├── logic_coder.md     # System prompt for pure Python generation
│   ├── ui_composer.md     # System prompt for Streamlit wiring
│   └── debugger.md        # System prompt for error fixing
│
└── output/                # --- Generated Apps Live Here ---
    └── [project_name]/    # (Created dynamically)
        ├── app.py         # Main Streamlit Entry point
        ├── logic.py       # Backend functions
        └── state.py       # State dictionary definition
```

4. Deep Dive: The Self-Correction Loop

The "Self-Healing" mechanism is the most critical part of ASET. It ensures that code generated by the **Logic Coder** actually runs before it is finalized.

The 4-Step Correction Flow

This loop happens **inside** the `agents/coder.py` workflow, specifically for **Logic Fragments**.

Step 1: Isolation Wrapping (The "Test Harness")

The agent generates a function, e.g., `calculate_tax(salary)`. You cannot just "run" this function; you need to call it. The system automatically generates a temporary **Test Harness** script:

Step 2: Safe Execution

The system runs `temp_test_harness.py`.

- **Production:** Runs inside an **E2B Sandbox** (secure).
- **Dev/Local:** Runs in a distinct `subprocess` with a timeout (5 seconds).

Step 3: Error Capture

If the subprocess returns `exit_code != 0`, or prints `TEST_FAILED`, the system captures the **Traceback**.

- *Example Error:* `TypeError: unsupported operand type(s) for *: 'str' and 'float'`
- *Diagnosis:* The agent forgot to cast the input string to a float.

Step 4: The Loop (Refinement)

The **Debugger Agent** receives a specific payload:

1. **The Broken Code:** `def calculate_tax...`
2. **The Error Message:** `TypeError...`
3. **The Prompt:** "You wrote this code. It threw this error. Fix it. Return only the fixed code."