Multi-Agentic Coding Framework

Design, Architecture, and Workflow Explanation

-Saksham Jain

---

1. Introduction

This document explains the complete design, architecture, and decision-making behind the **Multi-Agentic Coding Framework**. The system is a production-grade, deterministic multi-agent orchestration platform built using **AutoGen**, **Streamlit**, and multiple LLM backends (Groq / OpenRouter / Ollama).

The core objective of this framework is:

To convert a high-level natural language software request into a fully working, tested, documented, and deployable Python application — automatically — using a pipeline of specialized agents.

This is not a toy demo. This system enforces:

- Strict output contracts
- Deterministic execution order
- Hard termination boundaries
- Iterative improvement loops
- Automated test execution
- Human-visible execution logs and artifacts

---

2. System Overview

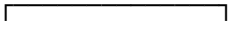
At a high level, the system consists of three major layers:

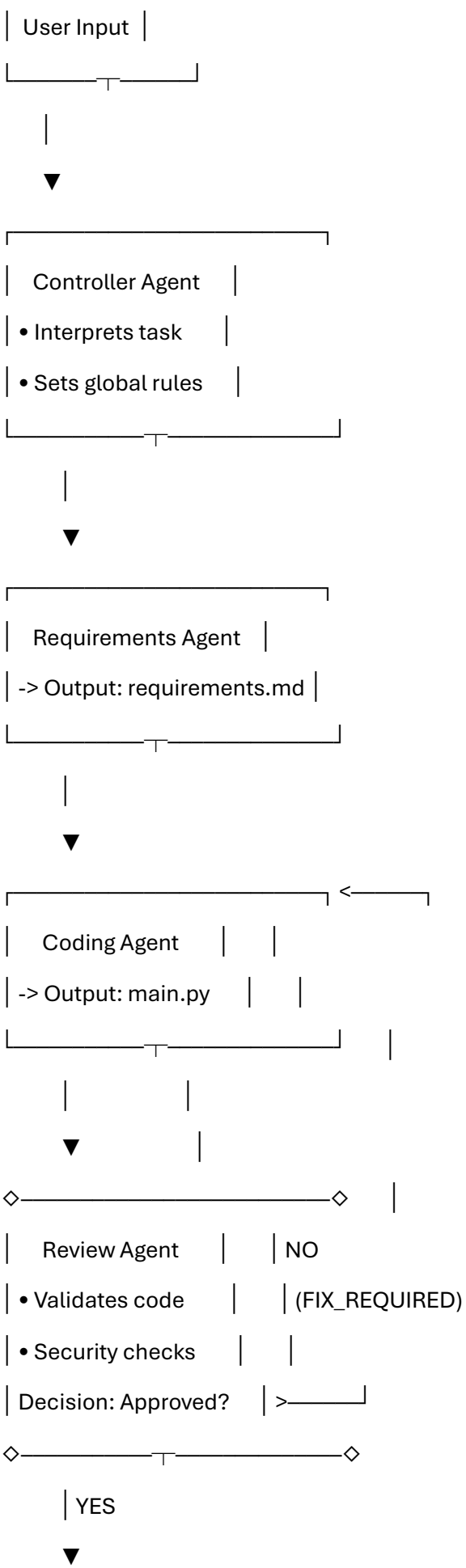
Layer	Responsibility
UI Layer (Streamlit)	Accepts user request, visualizes progress, displays generated files and test results
Orchestration Layer (WorkflowOrchestrator)	Controls agent execution order, iteration logic, file extraction, and test execution
Agent Layer (AutoGen Agents)	Individual specialists that generate requirements, code, reviews, docs, tests, deployment config, and UI

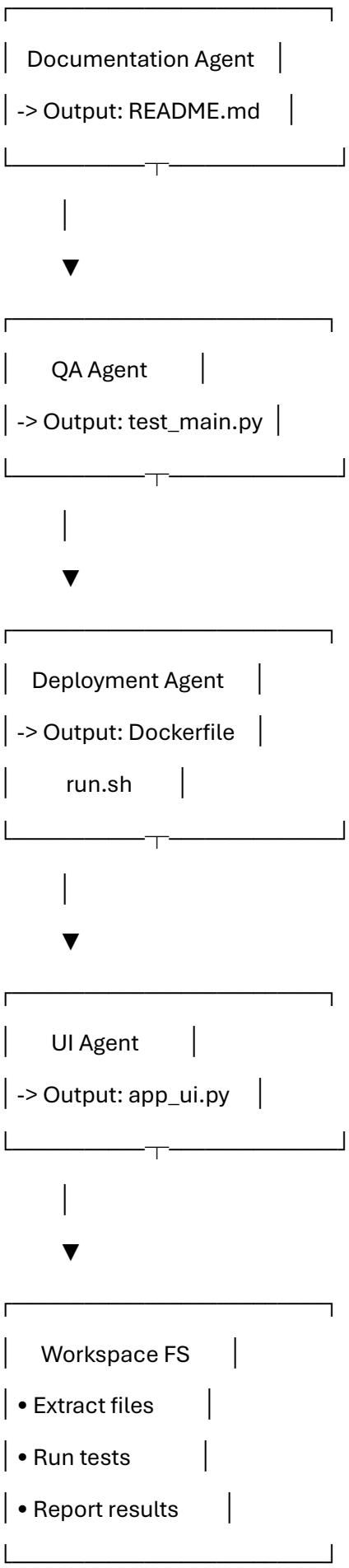
---

3. Workflow Diagram

The entire system executes using the following pipeline:




User Input
Controller Agent
• Interprets task
• Sets global rules
Requirements Agent
-> Output: requirements.md
Coding Agent
-> Output: main.py
Review Agent
• Validates code
• Security checks
Decision: Approved?
NO
(FIX_REQUIRED)
YES


Documentation Agent
-> Output: README.md
QA Agent
-> Output: test_main.py
Deployment Agent
-> Output: Dockerfile
run.sh
UI Agent
-> Output: app_ui.py
Workspace FS
• Extract files
• Run tests
• Report results

This is the **single source of truth** for execution.

No agent runs out of order. No agent skips responsibility. No agent writes outside its contract.

---

4. Agent Roles and Responsibilities

4.1 Controller Agent

- Reads the original user request
- Sets system-wide constraints
- Enforces file output format
- Controls task delegation

This agent **does not generate code**. It governs the system.

4.2 Requirements Agent

Input: User task
Output: requirements.md

- Converts vague ideas into structured functional and non-functional requirements
- Defines edge cases, assumptions, inputs, outputs
- Creates the canonical specification that all downstream agents must follow

---

4.3 Coding Agent

Input: requirements.md
Output: main.py

- Generates real production Python code
- Must use **only built-in libraries**
- Must be fully runnable
- Must exceed 100 lines
- Must contain full logic — no placeholders

4.4 Review Agent

Input: main.py
Output: APPROVED or FIX_REQUIRED

This is the gatekeeper.

- Detects logic bugs
- Detects security issues
- Enforces dependency constraints
- Forces regeneration when required

4.5 Documentation Agent

Output: README.md

- Explains purpose, architecture, features
- Provides installation, usage, testing, docker deployment
- Converts the system into something any engineer can understand

---

4.6 QA Agent

Output: test_main.py

- Generates real unit tests
- Mocks input/output
- Handles file I/O safely
- Prevents test timeouts on Windows

4.7 Deployment Agent

Output: Dockerfile, run.sh

- Makes project deployable instantly
- Keeps dependencies minimal
- Produces reproducible environment

4.8 UI Agent

Output: app_ui.py

- Creates a Streamlit interface for the generated application
- Allows interactive testing of produced code

5. File Structure & Workspace Lifecycle

The system operates around a dedicated directory named:

workspace/

This folder is the **single source of truth** for all generated artifacts. No agent writes directly to disk. Instead, all agents emit structured responses that are parsed and extracted by the orchestrator.

Workspace Layout After Successful Run

workspace/

├— main.py # Generated application code

├— requirements.md # Structured software requirements

├— README.md # Documentation

├— test_main.py # Unit tests

├— Dockerfile # Deployment config

├— run.sh # Local execution script

└— app_ui.py # Streamlit UI for generated app

Lifecycle of Workspace

Stage	Action
User clicks “Launch Team”	Workspace is cleaned
Agents generate outputs	Responses stored in GroupChat memory
Workflow completion	Orchestrator extracts files to workspace
QA Phase	TestExecutor runs tests inside workspace
UI Rendering	Streamlit reads workspace and displays files

No agent ever touches the filesystem directly.
This ensures deterministic behavior and eliminates file race conditions.

---

6. Workflow Orchestration Architecture

The orchestration engine is implemented in:

workflow.py → class WorkflowOrchestrator

This component enforces **sequential deterministic execution** using:

```
pipeline = [
    "Controller_agent",
    "Requirements_Agent",
    "coding_agent",
    "review_agent",
```

"Documentation_Agent",

"QA_Agent",

"Deployment_agent",

"UI_agent",

]

Execution Rules

1. Every agent runs **only when its turn arrives**.
2. Every agent sees the full conversation history.
3. If review_agent outputs FIX_REQUIRED, the pipeline is dynamically modified:

review_agent → coding_agent → review_agent

This loop is allowed only max_review_iterations times to prevent infinite recursion.

4. If the review never approves after N retries, the system force-terminates safely.

---

7. Why We Chose Sequential Execution

This decision was not arbitrary.

Alternative Architectures We Tried

Architecture	Failure Mode
Fully autonomous GroupChatManager selection	Agents produced inconsistent outputs
Parallel agent execution	Context collision, broken contracts
Recursive agent triggering	Infinite loops, memory explosion
HumanProxy-based interaction	Non-deterministic UX

Problems Observed

- Agents responding out of order
- Test agent running before code existed
- Review feedback not reaching coding agent
- OpenAI-compatible APIs rejecting malformed messages

Final Design Decision

We abandoned emergent multi-agent behavior in favor of a **controlled deterministic pipeline**.

This gives us:

Benefit	Why it Matters
Predictability	Every run behaves the same
Debuggability	You can pinpoint which agent failed
Contract enforcement	Each agent must follow strict formats
API stability	Works across Groq, OpenRouter, Ollama
Safe iteration	Controlled FIX_REQUIRED loops

This is the difference between a research demo and a production system.

---

8. LLM Backend Strategy

The system supports three backends via a single switch in agents.py.

Backend	Use Case
Groq (Llama 3.3 70B)	High quality cloud inference
OpenRouter	Model experimentation
Ollama (Local 7B)	Offline fallback / demo mode

We enforce OpenAI-compatible schema to avoid vendor lock-in.

---

9. Streamlit Frontend Architecture

The UI is defined in app.py.

Core responsibilities:

- Accept user request
- Trigger orchestration
- Display:
  - Generated artifacts
  - Test execution results
  - System logs

Test output visualization:

Metric	Displayed
Total tests	Yes
Passed	Yes

Metric	Displayed
Failed	Yes
Errors	Yes
Per-test output	Expandable blocks

---

10. How to Run the System

```
git clone <repo>
cd DataEconomy-Coding-Agent
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install -r requirements.txt
streamlit run app.py
```

---

11. Major Engineering Decisions Summary

Decision	Rationale
Deterministic pipeline	Eliminates hallucinated agent behavior
Hard file contracts	Guarantees artifact extraction
Workspace isolation	Prevents filesystem corruption
Strict review gate	Enforces code quality
Test execution sandbox	Provides real validation
API abstraction	Allows backend switching

---

12. Authorship

This Multi-Agentic Coding Framework was designed and implemented by:

Saksham Jain
Generative AI Engineer – Multi-Agent Systems