AutoGen Multi-Agent Coding Framework - System Documentation

Table of Contents

Overview

AutoGen Multi-Agent Coding Framework is an AI-powered software development system that uses **7 specialized AI agents** working collaboratively to transform natural language requirements into fully-functional Python applications. The system implements a sophisticated state transition graph to orchestrate agent interactions and ensure quality through automated code review and testing.

Key Capabilities

- 📊 **Automated Requirements Analysis** - Converts user requests into detailed specifications
- 💻 **Complete Code Generation** - Produces production-ready Python applications (100+ lines)
- 🔍 **Quality Assurance** - Automated code review with iterative improvements
- 📝 **Comprehensive Documentation** - Generates detailed README files
- 🧪 **Automated Testing** - Creates and executes unit tests
- 🚀 **Deployment Ready** - Provides Docker and deployment configurations
- 🎨 **User Interface** - Generates Streamlit-based UIs

Technology Stack

- **Framework**: Microsoft AutoGen 0.2.35
- **LLM Provider**: OpenAI GPT-4
- **Frontend**: Streamlit (web interface)
- **Language**: Python 3.8+
- **Architecture**: Multi-agent orchestration with custom state machine

Implemented Agents

The system implements **7 specialized AI agents** plus **1 User Proxy agent** for orchestration.

1. 📊 Product Manager

Role: Requirements Analysis and Specification

Responsibilities:

- Analyzes user requests and transforms them into detailed software requirements
- Creates comprehensive requirements documents with functional and non-functional requirements
- Defines project scope, inputs, outputs, and edge cases
- Ensures all requirements are specific, actionable, and testable

Output Format:

- Generates `requirements.txt` with structured requirements documentation
- Includes project overview, functional requirements (FR1, FR2, etc.), non-functional requirements, and assumptions

Configuration:

- Temperature: 0.7 (balanced creativity and precision)
- Model: GPT-4 (default)

2. 💻 Python Engineer

Role: Software Development and Implementation

Responsibilities:

- Writes complete, production-quality Python code (minimum 100 lines)
- Implements all functional requirements from the Product Manager
- Includes proper error handling, docstrings, and comments
- Creates executable, production-ready applications
- Iteratively fixes code based on Code Reviewer feedback

Output Format:

- Generates `main.py` with complete Python implementation
- Must include actual working code, not placeholders or TODO comments
- Follows PEP 8 style guidelines

Configuration:

- Temperature: 0.7 (balanced approach)
- Iterative improvement capability through review loop

3. 🔍 Code Reviewer

Role: Quality Assurance and Security Analysis

Responsibilities:

- Reviews Python code for correctness, efficiency, and security
- Checks for logic errors, bugs, and performance issues
- Identifies security vulnerabilities (SQL injection, XSS, etc.)
- Validates PEP 8 compliance and code style
- Ensures proper error handling and documentation

Output Format:

- Responds with `APPROVED` or `FIX_REQUIRED`
- For `FIX_REQUIRED` : Provides numbered list of specific issues
- For `APPROVED` : Brief positive summary

Decision Logic:

- **Approves** code that runs, satisfies requirements, and has no major bugs
- **Requires fixes** for logic errors, missing requirements, security issues, or critical bugs
- Balances strictness (focuses on real issues, not minor style nitpicks)

Configuration:

- Temperature: 0.3 (analytical, consistent)
- Maximum review iterations: 5 (prevents infinite loops)

4. 📝 Tech Writer

Role: Documentation Generation

Responsibilities:

- Creates comprehensive, professional README.md documentation
- Documents application purpose, features, and architecture

- Provides clear installation and usage instructions
- Includes detailed function references with parameters and examples
- Adds troubleshooting guides and performance considerations

Output Format:

- Generates `README.md` with complete project documentation
- Includes: Overview, Features, Requirements, Installation, Usage, Code Structure, Function Reference, Examples, Testing, Docker Deployment, Troubleshooting

Key Sections:

- Table of Contents with hyperlinks
- Step-by-step installation guide
- Comprehensive function/class references
- Multiple usage examples with expected outputs
- Docker deployment instructions

Configuration:

- Temperature: 0.7 (balanced)
- Focus on clarity and completeness

5. 🧪 QA Engineer

Role: Test Case Generation and Validation

Responsibilities:

- Writes comprehensive unit tests for the generated application
- Creates 5-10 meaningful test cases covering different scenarios
- Tests edge cases and error handling
- Uses Python's `unittest` framework
- Ensures tests actually call real functions (not placeholder tests)

Output Format:

- Generates `test_main.py` with complete unit tests
- Includes test classes, setup/teardown methods, and assertions
- Tests normal cases, edge cases, and error conditions

Test Requirements:

- Must import and test functions from `main.py`

- Must include actual assertions, not dummy tests like `assertEqual(1, 1)`
- Must be runnable with `python -m unittest test_main.py`

Configuration:

- Temperature: 0.7 (balanced)
- Focuses on practical, executable tests

6. 🚀 DevOps Engineer

Role: Deployment Configuration

Responsibilities:

- Creates Dockerfile for containerized deployment
- Generates run scripts for easy local execution
- Ensures production-ready deployment configurations
- Optimizes for simplicity and functionality

Output Format:

- Generates `Dockerfile` with Python 3.10-slim base image
- Generates `run.sh` bash script for quick execution
- Both files follow best practices for Docker and shell scripting

Docker Configuration:

- Uses lightweight Python base image
- Copies application files to `/app` workdir
- Installs dependencies if `requirements.txt` exists
- Runs `main.py` as the entry point

Configuration:

- Temperature: 0.7 (balanced)

7. 🎨 UX Designer

Role: User Interface Generation

Responsibilities:

- Creates beautiful, user-friendly Streamlit interfaces
- Provides clean input widgets (text inputs, buttons, sliders, etc.)
- Displays outputs in a clear, organized manner
- Adds helpful instructions and descriptions
- Implements proper layout with headers, sidebars, and sections

Output Format:

- Generates `app_ui.py` with complete Streamlit UI
- Imports and uses functions from `main.py`
- Includes page configuration, title, input widgets, processing logic, and output display

UI Features:

- Page configuration with title and icon
- Sidebar for settings (if applicable)
- Main content area with clear sections
- Input validation and error handling
- Loading spinners and success messages
- Responsive layout using Streamlit components

Configuration:

- Temperature: 0.8 (creative, for better UX design)

8. 🤵 Admin Proxy

Role: Workflow Orchestration and User Interface Bridge

Responsibilities:

- Coordinates the workflow and initiates agent conversations
- Bridges user input to the agent system
- Reviews agent outputs (does not execute code)
- Terminates workflow when complete

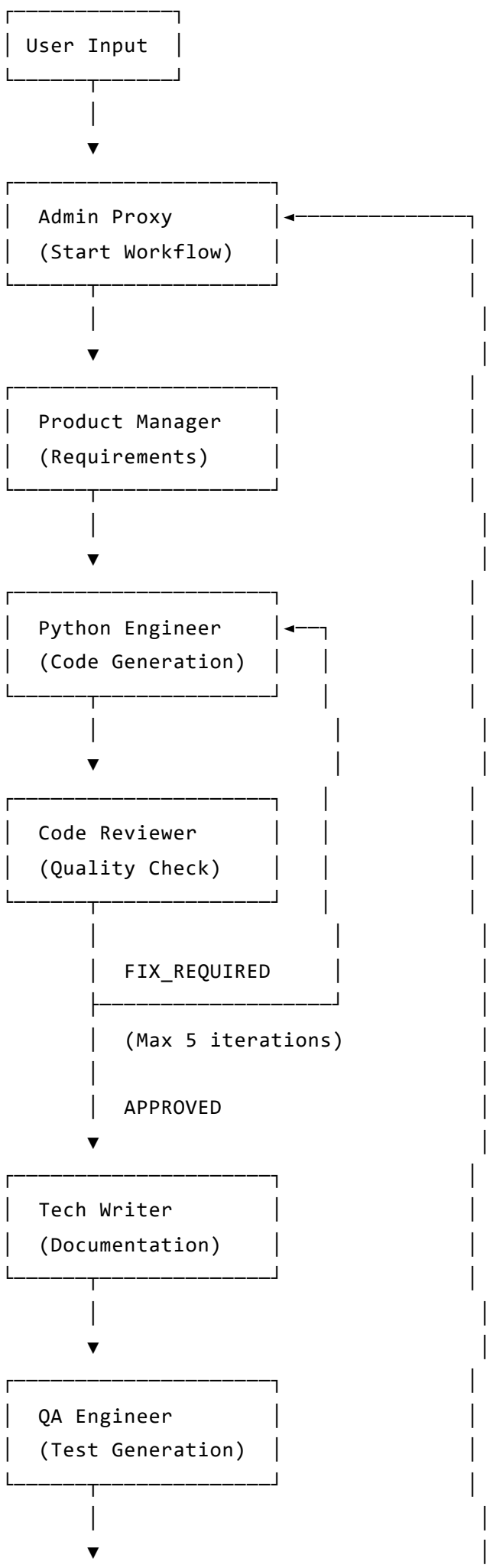
Characteristics:

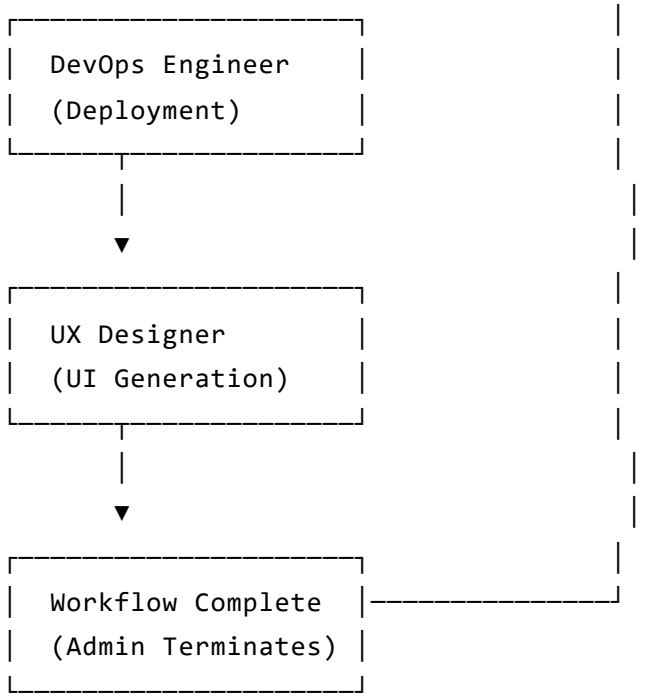
- `human_input_mode` : NEVER (fully automated)
- `max_consecutive_auto_reply` : 50
- `code_execution_config` : False (files are displayed in UI only, not executed)
- Terminates on `TERMINATE` message

Workflow and Architecture

State Transition Graph

The system implements a **directed state machine** for agent orchestration:


User Input
Admin Proxy
(Start Workflow)
Product Manager
(Requirements)
Python Engineer
(Code Generation)
Code Reviewer
(Quality Check)
FIX_REQUIRED
(Max 5 iterations)
APPROVED
Tech Writer
(Documentation)
QA Engineer
(Test Generation)



Critical Decision Points

1. **Code Review Loop** (Python Engineer ↔ Code Reviewer)

- **Trigger**: Code Reviewer responds with `FIX_REQUIRED`
- **Action**: Python Engineer iteratively fixes code
- **Max Iterations**: 5 (prevents infinite loops)
- **Force Approval**: After 5 iterations, workflow proceeds regardless to ensure completion

2. **Workflow Termination**

- **Trigger**: UX Designer completes UI generation
- **Action**: Workflow ends, all files are extracted and displayed

Architecture Components

1. **WorkflowOrchestrator** (`workflow.py`)

Purpose: Manages state transitions and agent coordination

Key Methods:

- `custom_speaker_selector()` : Implements the state transition logic
- `create_group_chat()` : Configures the multi-agent group chat
- `initiate_workflow()` : Starts the workflow with user request

Features:

- Custom speaker selection based on agent state
- Review iteration tracking and enforcement
- Progress callback support for UI updates
- File extraction from agent responses using marker format

2. Agent Factory (`agents.py`)

Purpose: Creates and configures all agent instances

Key Functions:

- `get_llm_config()` : Retrieves and validates LLM configuration from environment
- `create_agent()` : Factory function for creating AssistantAgent instances
- `create_all_agents()` : Instantiates all 8 agents with appropriate configurations

Agent Configurations:

- **Base Config**: Temperature 0.7, 120s timeout
- **Creative Config**: Temperature 0.8 (for UX Designer)
- **Analytical Config**: Temperature 0.3 (for Code Reviewer)

3. Streamlit UI (`app.py`)

Purpose: Web interface for user interaction

Key Features:

- User request input with validation
- Real-time workflow progress display
- Generated artifacts viewer (tabbed interface)
- Test execution results display
- Workspace management (clear, download as ZIP)
- Error handling and user feedback

UI Tabs:

1. **All Files**: Complete file listing with download buttons
2. **Python Code**: Syntax-highlighted Python files
3. **Documentation**: Rendered markdown and text files
4. **Deployment**: Docker and shell scripts
5. **Test Results**: Test execution summary and details

4. Test Executor (`utils/test_executor.py`)

Purpose: Runs generated tests and captures results

Key Methods:

- `find_test_files()` : Locates test files in workspace
- `execute_unittest_file()` : Runs tests using Python unittest
- `execute_all_tests()` : Aggregates results from all test files

Test Execution:

- Runs tests in isolated subprocess with 30-second timeout
- Prevents `__pycache__` creation (`PYTHONDONTWRITEBYTECODE=1`)
- Parses test output for pass/fail counts
- Returns structured test results for UI display

5. Logger (`utils/logger.py`)

Purpose: Structured logging for debugging and monitoring

Features:

- Console and file logging (dual output)
- Structured log format with timestamps
- Agent action tracking
- Error context capture

File Extraction System

Marker Format:

```
===BEGIN_FILE:filename.ext===
[file contents]
===END_FILE===
```

Process:

1. Agents wrap their file outputs in marker tags
2. WorkflowOrchestrator extracts files using regex pattern matching
3. Files are saved to `workspace/` directory
4. UI displays extracted files in categorized tabs

Supported File Types:

- Python code (`.py`)
- Documentation (`.md` , `.txt`)
- Deployment configs (`Dockerfile` , `.sh` , `.bat`)
- Test files (`test_*.py`)

System Components

Project Structure

```
DATAECONOMY.AI/
├── agents.py                    # Agent definitions and configuration
├── workflow.py                  # Workflow orchestration and state machine
├── app.py                       # Streamlit web interface
├── requirements.txt             # Python dependencies
├── .env                         # Environment configuration (user-created)
├── env_template.txt             # Environment template (if exists)
├── system_logs.log              # Application logs
├── utils/
│   ├── __init__.py
│   ├── logger.py                # Logging utilities
│   └── test_executor.py         # Test execution module
├── workspace/                   # Generated artifacts (created at runtime)
│   ├── main.py                  # Generated application code
│   ├── test_main.py             # Generated tests
│   ├── README.md                # Generated documentation
│   ├── requirements.txt         # Generated requirements
│   ├── Dockerfile               # Generated Docker config
│   ├── run.sh                   # Generated run script
│   └── app_ui.py                # Generated Streamlit UI
└── venv/                        # Virtual environment (user-created)
```

Key Files

`agents.py`

- Defines all 7 AI agents + Admin Proxy
- Configures LLM settings (API key, model, temperature)
- Implements system prompts for each agent role
- Provides agent factory functions

`workflow.py`

- Implements `WorkflowOrchestrator` class
- Custom speaker selection function
- State transition logic
- File extraction and test execution coordination

`app.py`

- Streamlit web application
- User input validation
- Progress tracking and status display
- Workspace artifact viewer
- Error handling and user feedback

`utils/test_executor.py`

- Test discovery and execution
- Result parsing and aggregation
- Timeout and error handling