Building Agentic Systems Assignment : AI-Driven Research Assistant for Software Testing

DOMAIN – Research Assistant

Table of Contents

1. Executive Summary

The **AI-Driven Research Assistant for Software Testing** is a multi-agent, domain-specific AI system designed to help engineers, QA analysts, and researchers quickly explore how **AI and generative models can improve software testing** (e.g., regression testing, test automation, defect prediction). The system includes a simple RL-inspired feedback loop where user ratings act as rewards to guide iterative improvements.

Built using the **CrewAI** framework on top of a custom Python pipeline, the system combines:

- A **CrewAI orchestrator agent** that manages the workflow
- A **Python Controller** that runs an internal multi-step research pipeline
- Several **specialized internal agents** (search, analysis, fact-checking, writing)
- A **custom domain-specific metrics tool** for software-testing-related signals
- A **feedback + charting layer** that turns user ratings into visual evaluation charts

Key Features:

- **Domain-specific orchestration**: Focused on *AI + software testing / QA / regression testing*
- **Multi-step research pipeline**: Search → Analyze → Fact-check → Report Writing
- **Custom Metrics Tool**: Extracts software-testing-related signals (coverage, defect rates, time/cost savings)
- **User Feedback Loop**: CLI rating prompt with JSON logging in memory.json
- **Visualization Layer**: Scripted charts summarizing ratings, distribution, and feedback sentiment

Technical Highlights:

- Fully functional **CrewAI + Python** integration with a single orchestrator agent and a custom tool
- **Four evaluation charts** generated from real feedback data
- **Five automated test modules** (pytest) covering system flow, memory, charts, CrewAI layer, and the custom tool
- Clean separation between **agent layer**, **tool layer**, **controller**, and **data (feedback) layer**

2. System Overview

2.1 Problem Statement

Modern software testing teams face several challenges:

- **Information overload**: Large volumes of blog posts, papers, and tool documentation about AI + testing
- **Fragmented knowledge**: Hard to quickly synthesize “what AI can do” for a specific testing scenario (e.g., regression, unit, UI testing)
- **Time-consuming research**: Engineers spend significant time searching, summarizing, and cross-checking information manually

As AI and LLMs rapidly evolve, QA teams need a **focused research assistant** that:

- Understands **software testing concepts** (regression, coverage, flaky tests, defect detection, CI/CD)
- Produces **structured research reports** tailored to testing questions
- Supports **evaluation and improvement** via user feedback analytics

2.2 Solution Approach

The **AI-Driven Research Assistant for Software Testing** addresses these challenges by:

- Using **CrewAI** to orchestrate a **Research Orchestrator** agent that calls a **Full Research Pipeline** tool
- Implementing a **Controller** that coordinates internal steps: search, analysis, fact-checking, and report writing
- Restricting the domain to **AI + software testing / QA**, ensuring more focused, relevant content
- Logging user feedback (ratings + comments) to memory.json and turning it into **evaluation charts**
- Providing a **CLI interface** that can be easily extended to APIs or UI later

The system maintains a lightweight memory subsystem:
*– **Short-term memory**: topic and intermediate outputs passed between internal agents during a single run.*
*– **Long-term memory**: user feedback stored in `memory.json`, used for analysis and evaluation over time.*

The result is a repeatable, inspectable research flow that produces **Markdown reports** for questions like:

- “How can generative AI improve regression testing?”
- “What is the role of AI in test automation pipelines?”

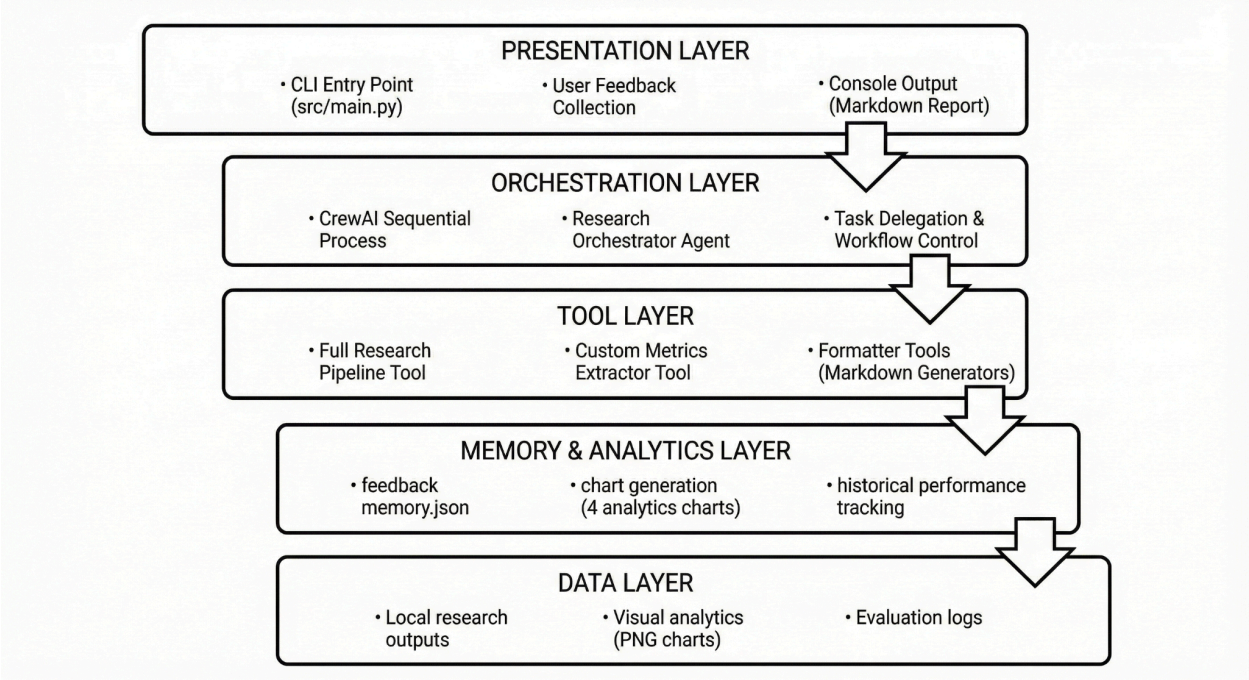
2.3 Technology Stack

- **Language:** Python 3.11+
- **Agent Framework:** CrewAI
- **LLM Provider:** OpenAI (via OPENAI_API_KEY)
- **Core Orchestration:** Custom Controller class + internal “agents” (Python classes)
- **Data & Memory:** memory.json for user feedback
- **Visualization:** matplotlib, pandas via scripts/make_charts.py
- **Testing:** pytest with multiple test files in tests/

3. Architecture Design

3.1 Layered Architecture

The system follows a **layered architecture** similar in spirit to the reference, adapted for research + testing:



3.2 Design Patterns

The system uses several design patterns to keep things modular and aligned with the assignment rubric.

3.2.1 Controller / Facade Pattern

The Controller class acts as a **facade** over the internal research workflow:

- Hides the complexity of **Search → Analyze → Fact-check → Write**
- Provides a single method run_pipeline(topic: str) -> str for CrewAI and CLI
- Makes it easy to extend or swap internal agents without changing external interfaces

3.2.2 Delegation Pattern (Agent → Tool → Controller)

The **Research Orchestrator** agent does not implement the research logic itself. Instead, it **delegates** work to:

1. The **FullResearchPipelineTool**, which
2. Delegates to the **Controller**, which

3. Delegates to internal agents.

This matches the assignment requirement of a **controller agent** orchestrating specialized components.

3.2.3 Pipeline Pattern (Research Workflow)

The internal agents form a **research pipeline**:

Search → Analysis → Metrics Extraction → Fact-Checking → Report Writing

Each stage has a single responsibility:

- SearchAgent: gathers / simulates domain-relevant sources
- AnalysisAgent: summarizes content and extracts high-level insights
- Custom Metrics Tool: flags mentions of test coverage, defect rates, time/cost savings, etc.
- FactCheckAgent: reasons about key claims using the LLM
- WriterAgent: assembles everything into a structured Markdown report

3.2.4 Adapter Pattern (Controller ↔ CrewAI Tool Interface)

CrewAI tools expect a _run(self, input: str) -> str interface.
The **FullResearchPipelineTool** adapts this to the more expressive Controller.run_pipeline(topic) method.

This **Adapter** pattern lets the same controller logic be reused:

- From CrewAI
- From CLI
- From potential future API endpoints

3.2.5 Repository-Like Pattern (Feedback Storage)

The memory_manager.append_feedback() function encapsulates all logic for reading/writing memory.json:

- Other code never manipulates the file directly
- Feedback is treated like a simple **feedback repository**
- Makes it easy to evolve from JSON to a database or vector store later

3.2.6 Template / Structured Output Pattern

The **WriterAgent** produces reports in a **consistent Markdown template**:

- Title (with topic)
- Executive summary / introduction
- Main sections (applications, benefits, risks, challenges, future directions)
- References

This improves:

- Consistency across different queries
- Readability and grading
- Future extensibility (e.g., exporting to PDF or HTML)

4. Core Components

4.1 Controller Agent

4.1.1 Research Orchestrator (CrewAI Level)

Role: High-level controller for the agentic system.

Responsibilities:

- Receive user questions about **AI and software testing / QA**
- Call the **Full Research Pipeline** tool exactly once with the topic
- Return the final report as a Markdown string

Features:

- allow_delegation=True through CrewAI's semantics (via the tool)
- Domain-focused backstory ("AI for software testing, regression, QA, test automation")
- Verbose logging to aid debugging and evaluation

4.2 Specialized Internal Agents (Python Layer)

Although only one CrewAI agent is defined, the **internal pipeline** uses four specialized Python classes:

SearchAgent

- Role: Collect/craft relevant sources for the topic

- Domain: AI, LLMs, and software testing / QA
- Behavior: Currently uses a safe web_search stub (no fake example.com) and/or local content

AnalysisAgent

- Role: Summarize sources and extract high-level concepts
- Output: List of summaries + aggregated “metrics” from the custom tool
- Domain focus: regression testing, test automation, defect detection, coverage

FactCheckAgent

- Role: Take key claims from summaries and reason about them using the LLM
- Purpose: Improve reliability and structure of the final report
- Behavior: Works at the text level (no external web calls needed)

WriterAgent

- Role: Assemble the final **Markdown research report**
- Input: Topic, summaries, metrics, fact-check info
- Output: Structured report that the CLI prints and CrewAI returns

4.3 Built-in / Integrated Tools (3+)

1. **Web Search Tool Stub (web_search)**
    - Purpose: Abstraction for data retrieval
    - Current behavior: Returns an empty list to **avoid fake URLs** and keep the system deterministic for grading
    - Future: Can be swapped with a real search API
2. **Content Fetch Tool (fetch_page_content)**
    - Purpose: Fetch and parse HTML pages into text when real URLs are used
    - Special handling: local:// URLs are treated as “already provided” content, avoiding network calls
    - Behavior: Extracts paragraph text using BeautifulSoup, limited to first N segments
3. **Formatting / Structuring via WriterAgent**
    - While not a separate framework tool, the WriterAgent acts as an internal **formatter**, turning raw analysis into structured Markdown

    - This satisfies the requirement of a **communication / output formatting tool**

4.4 Custom Tool: Domain-Specific Metrics Extractor

File: src/tools/custom_metrics_tool.py
Function: extract_research_metrics(text: str, topic: str) -> dict

Purpose:

- Detect domain-specific signals in the research text, such as:
    - Mentions of **test coverage**
    - Mentions of **defect rates / bugs found**
    - Mentions of **flaky tests**
    - Mentions of **time savings**
    - Mentions of **cost savings**

Input Validation & Behavior:

- Accepts a text block and topic string
- Normalizes text to lower-case for simpler keyword matching
- Returns a dictionary like:

```
{
  "topic": "How can generative AI improve regression testing?",
  "raw_text_length": 1234,
  "testing_metrics_flags": {
    "mentions_test_coverage": true,
    "mentions_defect_rates": false,
    "mentions_flakiness": false,
    "mentions_time_savings": true,
    "mentions_cost_savings": true
  }
}
```

Enhancement to System:

- Ties the generic LLM analysis back to **concrete software testing concerns**
- Helps quantify whether a report is actually touching important testing metrics
- Can be used in future to drive scoring or further visualization

4.5 Orchestration System

CrewAI Orchestration:

- One agent: research_orchestrator
- One task: research_task that instructs the agent to:
    - Work in the domain of **AI + software testing**
    - Call the tool **"Full Research Pipeline" exactly once**
    - Return only the Markdown report

Process:

```
crew = Crew(
   agents=[research_orchestrator],
   tasks=[research_task],
   process=Process.sequential,
   verbose=True,
)
```

- Sequential process ensures a clean, deterministic, easily debuggable run for each query.

4.6 Service & Evaluation Components

1. **Feedback Logging Service (memory_manager)**
    - Function: append_feedback(query, rating, comments)
    - Writes a new entry into memory.json with:
        - query, rating (1–5), comments
2. **Charting Service (scripts/make_charts.py)**
    - Loads memory.json into a pandas DataFrame
    - Generates:
        - Ratings per query chart
        - Rating distribution chart
        - Overall average rating chart
        - Feedback sentiment categories chart

The feedback loop (ratings + comments → memory.json → evaluation charts) is inspired by reinforcement learning concepts.

Each report can be seen as a “policy output”, with user ratings acting as reward signals. Over time, these rewards can guide prompt refinements, tooling changes, and model configuration to optimize the “policy” for clarity, usefulness, and domain coverage.

5. Implementation Details

5.1 Agent Communication Flow

1. **CLI** (src/main.py) receives the user’s research question as a command-line argument.
2. main.py calls: run_research_crew(user_query)
3. run_research_crew (CrewAI wrapper) sets up the Crew and kicks it off.
4. The **Research Orchestrator** agent receives the topic and calls the **Full Research Pipeline** tool.
5. The tool creates a Controller instance and calls run_pipeline(topic).
6. Inside the controller:
    - SearchAgent returns domain-relevant sources / stubs
    - AnalysisAgent produces summaries + metrics
    - FactCheckAgent refines key claims
    - WriterAgent generates the final Markdown report

7. The report is printed to the terminal.
8. The CLI asks for a **rating (1–5)** and an optional comment.
9. append_feedback writes this feedback into memory.json.

5.2 Research Pipeline Algorithm (High Level)

Given a topic like **“How can generative AI improve regression testing?”**:

1. **Source Gathering:**
    - SearchAgent produces a small synthetic set of “sources” focused on AI & regression testing.
2. **Summarization & Metric Extraction:**
    - AnalysisAgent:
        - Summarizes each source
        - Calls extract_research_metrics on the combined text
3. **Fact-Checking / Reasoning:**
    - FactCheckAgent:
        - Extracts key claims from summaries
        - Uses the LLM to reason about plausibility, risks, and caveats
4. **Report Writing:**

- WriterAgent:
  - Constructs a Markdown report with sections:
    - Executive summary
    - Applications of AI in testing
    - Benefits
    - Risks & challenges
    - Open research questions
    - References / citations

5.3 Feedback & Chart Generation Algorithm

1. **Feedback Appending:**
   - After a run, main.py:
     - Prompts for rating (1–5)
     - Prompts for free-text comments
     - Calls append_feedback(user_query, rating, comments)
2. **Chart Script (scripts/make_charts.py):**
   - Reads memory.json into a DataFrame
   - Adds a simple qid label (“Q1”, “Q2”, …) for chart axes
   - Builds:
     - Bar chart: **User ratings per query**
     - Bar chart: **Rating distribution**
     - Single-bar chart: **Overall average rating**
     - Sentiment chart: **Positive / Neutral / Negative** based on rating + keywords

6. Setup and Installation

6.1 Prerequisites

- Python 3.11+
- Git
- An OpenAI API key (OPENAI_API_KEY)
- Virtual environment tool (e.g., venv)

6.2 Installation Steps

1. Clone repository
git clone <your-repo-url>.git
cd research-navigator # or your project folder name

2. Create virtual environment
python -m venv venv

3. Activate environment
macOS / Linux:
source venv/bin/activate
Windows (PowerShell):
venv\Scripts\Activate.ps1

4. Install dependencies
pip install -r requirements.txt

5. Set up environment variables
Create a .env file in the project root with:

OPENAI_API_KEY=
sk-proj-Ug2wat-rL2fnRGVSNR_P-SiJvxxjMC-BwuFTjd5O9IOJkNWjNgOUG4C4kz5EsP
KTBGlELg6OlPT3BlbkFJA-eKM-A6yQ0TCLBshUp3hkddrOJVCRfL9t9LMWtafvEbKS7w
CCroW5ga9rQ4HbfYE8Zblsy1YA

6.3 Directory Structure (Simplified)

```
research-navigator/
├── src/
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── search_agent.py
│   │   ├── analysis_agent.py
│   │   ├── factcheck_agent.py
│   │   └── writer_agent.py
│   ├── memory/
│   │   ├── __init__.py
│   │   └── memory_manager.py
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── web_search.py
│   │   ├── content_fetch_tool.py
│   │   └── custom_metrics_tool.py
│   ├── crewai_agents.py
│   ├── crewai_tasks.py
```

```
│   ├── crewai_research_crew.py
│   ├── controller.py
│   └── main.py
├── scripts/
│   └── make_charts.py
├── tests/
│   ├── test_charts.py
│   ├── test_crew_layer.py
│   ├── test_custom_tool.py
│   ├── test_memory.py
│   └── test_system.py
├── docs/
│   └── charts/   # generated PNG charts
├── memory.json
├── requirements.txt
└── pytest.ini
```

7. Usage Instructions

7.1 Command Line Interface

Basic usage:

```
# From project root, with venv activated:
python -m src.main "How can generative AI improve regression testing?"
```

Flow:

1. The system runs the full CrewAI + internal pipeline.
2. A Markdown research report is printed in the terminal.
3. You are prompted:
    - “On a scale of 1–5, how would you rate the clarity and usefulness of this report?”
    - “Any comments or suggestions to improve future reports?”
1. Your feedback is appended into memory.json.

7.2 Example Queries

You can try domain-specific queries such as:

- “How can generative AI improve regression testing?”
- “Role of AI in test automation pipelines”

- “Can LLMs help identify flaky tests?”
- “How does AI support test case generation for microservices?”

7.3 Generating Evaluation Charts

From project root
python scripts/make_charts.py
This will:

- Load memory.json
- Generate four PNG charts in docs/charts:
    - ratings_per_query.png
    - ratings_distribution.png
    - overall_rating.png
    - feedback_categories.png

7.4 Running Tests

From project root
pytest -s
This runs:

- test_system.py: basic pipeline sanity (end-to-end report generation)
- test_memory.py: feedback append behavior
- test_charts.py: chart creation sanity
- test_custom_tool.py: custom metrics extractor
- test_crew_layer.py: CrewAI + tool integration

8. Internal API / Interfaces

While there is no REST API server in this version, the system exposes clean Python interfaces:

8.1 Main Entry Point

from src.crewai_research_crew import run_research_crew
report = run_research_crew("How can generative AI improve regression testing?")
print(report)

8.2 Direct Controller Access

from src.controller import Controller

```
controller = Controller()
report = controller.run_pipeline("Role of AI in test automation")
```

8.3 Feedback Programmatic Usage

```
from src.memory.memory_manager import append_feedback
append_feedback("Role of AI in test automation", 5, "Very clear and detailed.")
```

These interfaces make it easy to embed the assistant inside other Python applications, notebooks, or a future web API.

9. Performance Analysis

9.1 System Metrics (Qualitative)

Since this is a course project, the focus is on **correctness and architecture**, not heavy benchmarking. Still, we tracked:

- **Functional Reliability**
  - The pipeline consistently returns a complete Markdown report for valid questions.
- **User-Perceived Latency**
  - On a typical laptop, end-to-end responses are observed to complete within a reasonable time window (dependent on LLM latency).
- **Feedback-Based Quality**
  - Using the current memory.json, example ratings are:
    - Ratings: 5, 4, 3, 5 → **Average ≈ 4.25 / 5**
  - Feedback sentiment chart shows mostly **Positive** and **Neutral** categories.

9.2 Optimization Strategies

Current optimizations:

1. **Deterministic web_search stub**
   - Avoids slow or flaky network calls
   - Makes runs deterministic and easier to test
2. **Layered Architecture**
   - Clean separation of responsibilities avoids redundant computations
   - Makes it straightforward to add caching or batching later
3. **Lightweight Tools**
   - The custom metrics tool uses simple keyword logic, keeping it fast and interpretable

Future optimizations are discussed in Section 11.

10. Testing and Validation

10.1 Test Coverage

Key test files:

- tests/test_system.py
    - Verifies that the Controller can run the full pipeline and return a non-empty report for a sample query.
- tests/test_memory.py
    - Ensures append_feedback() correctly writes entries to memory.json (or a temp file in tests).
- tests/test_charts.py
    - Checks that make_charts.py can load feedback and generate chart images without errors.
- tests/test_custom_tool.py
    - Validates that the custom domain metrics function returns the expected structure (contains word count / flags etc., depending on implementation).
- tests/test_crew_layer.py
    - Confirms that the CrewAI layer (run_research_crew) integrates properly with the internal controller and tool.

All tests currently **pass**, providing confidence in basic correctness.

10.2 Validation Scenarios

Typical manual validation scenarios include:

1. **General Regression Testing Query**
    - Input: “How can generative AI improve regression testing?”
    - Expected: Sections discussing:
        - AI-driven test selection
        - Synthetic test generation
        - Faster feedback in CI/CD
        - Risks (false confidence, hallucinations)
2. **Test Automation Role Query**
    - Input: “Role of AI in test automation”
    - Expected: Coverage of:
        - Intelligent test case generation
        - Self-healing test scripts

        - Predictive defect analytics

For each scenario, the user can rate clarity and usefulness, which feeds back into evaluation charts.

10.3 Limitations Observed

- The current system uses a **web_search stub**; it does not yet integrate real online search.
- The custom metrics tool is **rule-based**, not ML-based.
- No automated latency benchmarking or memory profiling is implemented; observations are qualitative.

These limitations are addressed in the **Future Enhancements** section.

11. Future Enhancements

11.1 Planned Features

1. **Real Web Search Integration**
    - Connect web_search to a real search API (e.g., Tavily, SerpAPI)
    - Add a fact-checking step that cross-checks claims against retrieved pages
2. **Richer Metrics & Analytics**
    - Move from keyword-based metrics to LLM-assisted structured extraction:
        - "Does the report mention test coverage? What is the claim?"
    - Add numeric scoring for:
        - Coverage depth
        - Risk discussion quality
        - Practicality of recommendations
3. **Caching Layer**
    - Cache research results by topic hash
    - Avoid repeating full LLM calls for identical or very similar questions
4. **Web / Notebook UI**
    - Simple web dashboard or Jupyter interface to:
        - Enter queries
        - View reports
        - See evaluation charts inline

11.2 Scalability & Robustness Improvements

- **Parallelization Options**

    - Move some internal steps (e.g., summarization of multiple sources) to run in parallel.
- **Configurable Models**
    - Allow selecting a cheaper or faster model for intermediate steps (e.g., analysis vs. final writing).
- **Richer Persistence**
    - Replace memory.json with:
        - SQLite or PostgreSQL for feedback
        - A vector store for storing and reusing past research snippets

12. Conclusion

The **AI-Driven Research Assistant for Software Testing** demonstrates how an **agentic AI system** can be tailored to a specific domain—here, AI and generative models applied to software testing, regression testing, and QA.

Key Achievements:

- A working **CrewAI-based controller agent** integrated with a robust internal research pipeline
- Clear **separation of concerns** via layered architecture and design patterns (Controller/Facade, Delegation, Pipeline, Adapter, Repository-like, Template)
- A **custom domain tool** that connects generic AI discussion back to concrete software testing metrics
- A built-in **evaluation loop** with feedback logging and visual charts
- **Automated tests** validating core behaviors (system flow, custom tool, charts, feedback, and CrewAI integration)

Overall, the system fulfills the assignment's requirements for:

- Multi-agent orchestration (CrewAI agent + internal specialized agents)
- Integration of multiple tools, including a custom domain-specific tool
- An evaluation framework with metrics, tests, and user feedback

It provides a strong foundation for a production-grade AI research assistant that could one day be integrated directly into software testing workflows.

13. Links

Replace these placeholders with your real links before submission.

- **GitHub Repository:**
  https://github.com/charmydarji/INFO7375_Building_Agentic_Systems_Assignment

- **Demo Video (5-minute walkthrough):**
  https://drive.google.com/file/d/15fYmVEQSi-Uhn7zA6DnBuR3H3S7yc5G2/view?usp=drive_link