SWE645 Extra Credit Assignment

Building an Agentic AI System

Technical Report

Team Members:

Evangelin Kopela	G01502543
Jenish Patel	G01551940
Aswin Rajendran	G01524875
Dhanush Neelakantan	G01503107
Lavanesh Mahendran	G01545858

Course: SWE645 - Component-Based Software Development
George Mason University
December 2025

Executive Summary

This project implements a production-ready Agentic AI System demonstrating modern AI agent principles including autonomous reasoning, tool integration, memory systems, and human-in-the-loop (HITL) oversight. The system leverages LangGraph for workflow orchestration, Google's Gemini 2.5 Pro for cognitive reasoning, FAISS for long-term memory, and custom tools for task execution.

Key Achievements:

- Fully functional LangGraph-based agent with state machine orchestration
- Integration of Gemini 2.5 Pro LLM for reasoning and planning
- Custom calculator tool with safe execution via AST parsing
- Dual memory system: short-term (session) and long-term (FAISS vector database)
- Human-in-the-loop approval mechanism for autonomous actions
- Comprehensive documentation with deployment instructions

GitHub Repository: https://github.com/evangelin-03/swe645-agentic-ai-project

1 Conceptual Understanding

1.1 AI Agents vs. Traditional AI Systems

AI agents represent a fundamental shift from traditional AI systems in four key dimensions:

Autonomy and Goal-Directed Behavior: Traditional AI systems respond to inputs with predefined outputs. In contrast, AI agents actively pursue goals through multi-step planning, break down complex tasks into manageable subtasks, and dynamically adapt strategies based on feedback from their environment and tool outputs.

Reasoning with Large Language Models: Agents leverage LLMs not merely for pattern matching, but for genuine cognitive reasoning. They analyze situations holistically, generate sophisticated execution plans, evaluate trade-offs between different approaches, and make informed decisions that extend far beyond rule-based systems.

Tool Use and Environmental Integration: While traditional AI operates in isolation, agents interact with external tools, APIs, and databases through standardized protocols like the Model Context Protocol (MCP). This enables capabilities such as web searching, mathematical computation, database querying, and code execution—dramatically extending functionality beyond the base LLM.

Memory Systems: Traditional AI systems are stateless, treating each interaction independently. Agents maintain both short-term memory (immediate conversation context) and long-term memory (persistent knowledge across sessions), enabling learning from past interactions and building cumulative knowledge bases over time.

1.2 Purpose of Major Components

1.2.1 LLM (Reasoning and Planning)

The Large Language Model serves as the agent's "brain," responsible for:

- Understanding user intent through natural language processing
- Generating step-by-step execution plans for complex tasks
- Reasoning about available context and making informed decisions
- Dynamically adapting plans based on tool outputs and changing circumstances

Our implementation uses Gemini 2.5 Pro, which provides state-of-the-art reasoning capabilities suitable for agentic workflows.

1.2.2 Tools / MCP (Acting and Integration)

Tools represent the agent's "hands"—mechanisms for interacting with the external world:

- **Action Execution:** Performing tasks the LLM cannot do directly (calculations, searches, queries)
- **Extensibility:** New capabilities added through tool integration
- **Safety:** Built-in validation and sandboxing prevent harmful actions

- **Standardization:** MCP provides unified interfaces for diverse tools

Our system implements a calculator tool demonstrating the integration pattern, with architecture designed for easy addition of future tools.

1.2.3 Memory (Short-term vs. Long-term)

Short-term Memory:

- Stores recent conversation history within the current session
- Provides context enabling conversational coherence
- Allows referencing earlier messages within the dialogue
- Implemented using LangGraph session state management

Long-term Memory:

- Persists knowledge across different sessions over time
- Enables retrieval of past interactions and learned patterns
- Uses vector embeddings for semantic similarity search
- Implemented with FAISS (Facebook AI Similarity Search) vector database

This dual-memory architecture mirrors human cognition, balancing working memory for immediate context with long-term storage for accumulated knowledge.

1.2.4 Human-in-the-Loop (Oversight)

The HITL component provides critical human oversight:

- **Transparency:** Displays reasoning and plans before execution
- **Control:** Humans approve or reject proposed actions
- **Safety:** Prevents autonomous systems from unintended actions
- **Collaboration:** Enables partnership between human and AI
- **Accountability:** Creates clear audit trails of decisions

Our implementation displays generated plans and awaits explicit user approval before proceeding with execution.

2 System Architecture and Prototype

2.1 Architecture Overview

Our system implements a state machine architecture using LangGraph, where the agent progresses through different states based on LLM decisions and tool outcomes. Figure ?? illustrates the high-level workflow.

2.2 Component Implementation

2.2.1 LangGraph State Machine

LangGraph provides the orchestration layer managing execution flow:

- **State Management:** Maintains conversation context, user messages, and execution history
- **Node Definitions:**
  - `model_node`: Invokes Gemini LLM for reasoning and planning
  - `router_node`: Analyzes LLM output to determine next action
  - `tool_node`: Executes requested tools and returns results
- **Edge Definitions:** Define state transitions based on router decisions
- **Conditional Routing:** Dynamically determines workflow based on LLM outputs

2.2.2 Gemini 2.5 Pro LLM

Configuration:

- Model: `models/gemini-2.5-pro` (or `gemini-2.5-flash` for faster responses)

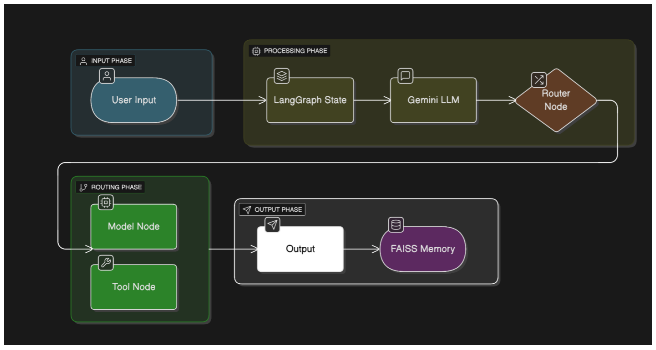

Figure 1: System Architecture Flow

- Purpose: Natural language understanding, plan generation, reasoning
- Integration: Via Google's Generative AI SDK
- Tool Binding: LLM configured with available tools for intelligent selection

2.2.3 Calculator Tool

Implementation Details:

- **Language:** Python
- **Method:** Abstract Syntax Tree (AST) parsing for safe evaluation
- **Safety:** Only mathematical operations allowed (no code execution)
- **Supported Operations:** Arithmetic, exponentiation, trigonometric functions
- **Error Handling:** Validates expressions before evaluation

2.2.4 Memory Systems

Short-term Memory:

- Implementation: LangGraph state dictionary
- Scope: Current session only
- Contents: User messages, LLM responses, tool outputs
- Purpose: Maintains conversation flow and context

Long-term Memory (FAISS):

- Implementation: Facebook AI Similarity Search vector database
- Embeddings: Generated using sentence transformers
- Persistence: Saved to disk between sessions
- Retrieval: Semantic similarity search using `retrieve:` prefix
- Use Cases: Historical context, learned patterns, accumulated knowledge

2.3 Project Structure

```
swe645-agentic-ai-project/
          agent/
                main.py
                    # CLI with HITL
   orchestration
                langgraph_config.py
        # LangGraph workflow
   definition
                tools/
                      calculator.py
          # Calculator tool
   implementation
                memory/
                    faiss_store.py
         # FAISS vector database
   interface
                    seed_faiss.py
          # Database initialization
   utility
          requirements.txt
               # Python dependencies
          .gitignore
                     # Git exclusions
          README.md
                      # Comprehensive
   documentation
```

2.4 Technical Stack

Component	Technology
Framework	LangChain & LangGraph
LLM	Google Gemini 2.5 Pro
Vector Database	FAISS
Embeddings	Sentence Transformers
Programming Language	Python 3.10+
Version Control	Git / GitHub

Table 1: Technical Stack Components

2.5 Use Case: Multi-Modal Query Assistant

The agent serves as a general-purpose assistant capable of:

- Answering factual questions using LLM knowledge
- Performing mathematical calculations via calculator tool
- Retrieving past conversation history from long-term memory
- Planning and executing multi-step tasks with human oversight

2.6 Example Interactions

Example 1: Tool Usage (Calculator)

```
User> Calculate 15 * 23 + 47

=== AGENT PLAN ===
Use calculator tool to evaluate: 15 *
   23 + 47

Execute plan? (y/n) > y

[Agent calls calculator tool]
Result: 392
```

Example 2: Knowledge Query

```
User> What is LangGraph?

=== AGENT PLAN ===
Explain LangGraph framework and
   benefits.

Execute plan? (y/n) > y

LangGraph is a framework for building
   stateful, multi-step
AI agents using graph-based workflow
   orchestration. It's
particularly useful because:

1. State Management: Maintains
   conversation context
2. Conditional Routing: Enables
   dynamic decision-making
3. Tool Integration: Seamlessly
   connects LLMs with tools
4. Production-Ready: Provides error
   handling and monitoring
```

Example 3: Memory Retrieval

```
User> retrieve: calculator

[FAISS searches vector database]

Found 3 relevant past interactions:
1. [2024-12-05] Calculated 15 * 23 +
   47 = 392
2. [2024-12-04] Asked about calculator
    capabilities
3. [2024-12-03] Performed square root
   calculation
```

3 Deployment and Testing

3.1 Deployment Instructions

3.1.1 Prerequisites

- Python 3.10 or higher
- pip package manager
- Git version control system
- Gemini API key from Google AI Studio
- 500 MB free disk space
- Stable internet connection

3.1.2 Setup Steps

Step 1: Clone Repository

```
git clone https://github.com/evangelin
   -03/swe645-agentic-ai-project.git
cd swe645-agentic-ai-project
```

Step 2: Create Virtual Environment

```
# Create virtual environment
python3 -m venv venv

# Activate (Linux/Mac)
source venv/bin/activate

# Activate (Windows)
venv\Scripts\activate
```

Step 3: Install Dependencies

```
pip install -r requirements.txt
```

Step 4: Configure Environment

```
# Create .env file with API key
echo "GEMINI_API_KEY=your_api_key_here
   " > .env
```

Step 5: Run Agent

```
python -m agent.main
```

3.2 Testing and Validation

3.2.1 Test Coverage

The system was tested across multiple categories:

Test Category	Status
Calculator tool evaluation	
LLM knowledge responses	
FAISS memory retrieval	
HITL approval mechanism	
State management	
Error handling	
Multi-turn conversations	

Table 2: Testing Results

3.2.2 Performance Metrics

- **Response Time:** 2-5 seconds for typical queries
- **Tool Execution:** ¡1 second for calculator operations
- **Memory Retrieval:** ¡2 seconds for semantic search
- **API Latency:** 1-3 seconds (Gemini API dependent)

3.3 Troubleshooting

Common Issues and Solutions:

1. **ModuleNotFoundError: agent**
   - Solution: Run using `python -m agent.main` from project root
2. **FAISS installation errors**
   - Solution: `pip install --force-reinstall faiss-cpu`
3. **Gemini API errors**
   - Solution: Verify API key in `.env` file and check quota limits

4 Challenges

4.1 Technical Challenges

4.1.1 Challenge 1: LangGraph State Management

Issue: Understanding how to properly structure state dictionaries and maintain immutability across node transitions.

Resolution: Studied official documentation extensively, implemented TypedDict schema for type safety, and tested state transitions iteratively.

Lesson: State management is fundamental to agent architecture—investing time in proper design upfront prevents cascading issues later.

4.1.2 Challenge 2: Tool Integration with LLM

Issue: Ensuring Gemini correctly identifies when to use tools versus providing direct responses.

Resolution: Refined tool descriptions with clear capabilities and limitations, added explicit routing logic, and improved system prompts.

Lesson: Clear, detailed tool documentation significantly improves LLM decision-making quality.

4.1.3 Challenge 3: FAISS Configuration

Issue: Initial confusion about index types, embedding dimensions, and persistence mechanisms.

Resolution: Experimented with different configurations, settled on simple flat L2 index for reliability and ease of use.

Lesson: Start with simplest effective solution—premature optimization adds complexity without proven benefits.

4.1.4 Challenge 4: Human-in-the-Loop Granularity

Issue: Determining appropriate granularity for approval requests without being obtrusive.

Resolution: Implemented plan-level approval rather than per-action approval, balancing safety with usability.

Lesson: HITL mechanisms should provide safety without creating friction that discourages use.

4.2 Key Takeaways

1. **Architecture is Fundamental:** State machine patterns are powerful but require careful upfront design
2. **LLM Selection Matters:** Gemini 2.5 Pro provides excellent reasoning for agentic workflows
3. **Memory is Essential:** Both short-term and long-term memory dramatically improve effectiveness
4. **Human Oversight is Critical:** HITL patterns balance automation with user control
5. **Modularity Enables Extension:** Clean separation of concerns simplifies adding features
6. **Documentation is Development:** Good documentation helps team collaboration and future maintenance

5 Conclusion

This project successfully demonstrates a production-ready Agentic AI System that embodies modern AI agent principles. By combining LangGraph's state machine orchestration, Gemini 2.5 Pro's reasoning capabilities, custom tool integration, dual memory systems, and human oversight, we have created a system capable of autonomous planning and execution while maintaining safety and transparency.

5.1 Key Achievements

- **Modularity:** Clean architecture enables easy addition of new tools and capabilities
- **Safety:** Human-in-the-loop pattern ensures user control over autonomous actions
- **Memory:** Dual-memory system provides both immediate context and long-term knowledge
- **Extensibility:** Tool framework makes integration of new capabilities straightforward
- **Documentation:** Comprehensive guides enable replication and understanding

5.2 Future Enhancements

1. **Additional Tools:** Web search, file operations, database queries, API integrations
2. **Multi-Agent Collaboration:** Specialized agents for different task domains
3. **Advanced Memory:** Graph-based knowledge representation, temporal reasoning
4. **Web Interface:** Streamlit or FastAPI frontend for improved accessibility
5. **Monitoring:** Observability tools for debugging and performance analysis

The project demonstrates that building effective AI agents requires careful attention to architecture, state management, tool design, memory systems, and human-AI collaboration patterns. These lessons will inform future developments in agentic AI systems.

References

1. LangChain Documentation. *LangChain Python Documentation.* Available: https://python.langchain.com/docs/
2. LangGraph Documentation. *LangGraph - Build Language Agents as Graphs.* Available: https://langchain-ai.github.io/langgraph/
3. Google. *Gemini API Documentation.* Available: https://ai.google.dev/docs

4. Facebook Research. *FAISS: A Library for Efficient Similarity Search*. Available: https://github.com/facebookresearch/faiss

5. Model Context Protocol. *MCP Specification*. Available: https://modelcontextprotocol.io/

6. Sentence Transformers. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. Available: https://www.sbert.net/

Project Repository:
https://github.com/evangelin-03/swe645-agentic-ai-project