AI Agent Architecture for Domain-Specific Intelligence

Table of Contents

1. Objective

This document outlines the architecture and technical implementation of AI Agents tailored for domain-specific intelligence. The objective is to design a modular, extensible agent framework capable of mirroring a user?s skill set across fields like finance, psychology, technical writing, and automation. The system should support rapid prototyping, scalable execution, and seamless integration with real-world workflows.

2. Overview of Agent-Based Systems

AI Agents are self-contained software modules that perceive their environment, reason based on context and predefined instructions, and act autonomously or semi-autonomously. Unlike traditional chatbots, these agents carry embedded workflows, behavioral schemas, and operational scope derived from subject matter expertise.

3. Technical Architecture

Each AI Agent is structured as a composite of the following layers:

- Memory Layer: Long-term and short-term memory management using vector databases.
- Skill Layer: Injected via prompt engineering, APIs, datasets, or YAML-defined task flows.
- Execution Layer: Integrated with external systems (REST APIs, command-line tools).
- Interface Layer: CLI, GUI, or embedded dashboard (e.g., FastAPI + React).

Agents are deployed as isolated services using containerization (e.g., Docker), orchestrated via Kubernetes for scalability.

4. Skill Injection and Personalization

Agents are designed to replicate human expertise by injecting domain-specific data and tasks. For example:

- A ?Finance Agent? receives prompt history about trading patterns, API keys to market data, and YAML workflows for alert generation.
- A ?Tech Writing Agent? is injected with semantic structures, doc templates, glossary terms, and Markdown/DITA schemas.

Personalization is achieved by encoding the user?s behavior, decision trees, and tool preferences into the memory and skill layers.

5. Workflow Automation with YAML and APIs

YAML files serve as the declarative backbone for task execution:

```yaml
agent:
name: "OptionsBot"
description: "Executes daily option sweeps and alerting"
schedule: "0 8 * * *"
actions:
- fetch_market_data: { source: "TD Ameritrade", symbols: ["TSLA", "NVDA"] }
- analyze_patterns: { strategy: "bull_call_spread" }
- notify_user: { channel: "email", template: "daily_summary.html" }
```

Agents read these configurations and execute corresponding API calls, either through cron-scheduled jobs or reactive triggers.

6. Use Case Scenarios

- Trading Agent: Executes structured trading strategies based on real-time data.
- Research Assistant: Fetches, summarizes, and clusters academic research.
- Technical Writing Agent: Converts raw developer notes into DITA/Markdown documents.
- Fitness/Nutrition Coach: Tracks physical data and provides real-time adjustments.

Each agent is modular and independently executable, while sharing a common schema for inter-agent communication.

7. System Requirements and Deployment

Minimum Specs:

- Python 3.11+, FastAPI, LangChain
- Vector DB (e.g., Chroma, Weaviate)
- Redis or SQLite for caching/state
- Docker + Compose (or Kubernetes for production)
- GitHub Actions or Airflow for orchestration

Deployment can be local (MacOS/Linux), containerized, or cloud-based (GCP/AWS).

8. Future Directions and Scalability

- Memory Architecture: Move toward neuro-symbolic memory graphs
- Agent-to-Agent Messaging: Enable agent orchestration using pub/sub
- Monetization: Offer agents as microservices on marketplace (e.g., HuggingFace Agents)
- DSL for Agents: A simple domain-specific language for declarative agent design

9. Appendix

- Author: Arsalan Khan
- Project: Sophie.AI Agent Layer