



# MUXI.AI

A modular, open-source framework for building multi-AI-agent systems orchestration with persistent memory, standardized communication protocols, and chain-of-thought tracing

April 2025

**Author:** Ran Aroussi ([ran@aroussi.com](mailto:ran@aroussi.com))

**Website:** [github.com/ranaroussi/muxi](https://github.com/ranaroussi/muxi)

**X/Twitter:** [@aroussi / @muxi\\_ai](https://twitter.com/aroussi)



# Executive Summary

This white paper introduces MUXI, an open-source framework addressing the limitations of traditional AI-agent systems. MUXI enables the development of sophisticated agent swarms through an orchestration layer, tiered memory architecture, standardized communication protocols, and flexible deployment options.

## **Key innovations include:**

- Intelligent message routing between specialized agents
- Short-term and long-term memory retention capabilities
- Integration with Model Context Protocol (MCP) for standardized tool usage
- Agent-to-Agent (A2A) communication protocol
- Support for multiple vector database backends
- Comprehensive observability for monitoring agent behavior

The framework demonstrates significant improvements in context retention, agent collaboration, and task completion compared to single-agent approaches, establishing a new paradigm for orchestrating complex AI agent swarms with a persistent understanding of user contexts.

This paper presents the framework's foundations, architecture, and implementation details, outlining its potential applications in enterprise knowledge management, research acceleration, and multi-modal interfaces. As AI systems grow increasingly complex, MUXI provides a standardized approach to building modular, extensible agent networks that can be deployed from edge devices to distributed cloud environments.

**Keywords:** Agent swarms, multi-agent systems, persistent memory, AI orchestration, agent communication, vector databases



# What does MUXI mean?

MUXI stands for "**Multiplexed eXtensible Intelligence**".

- **Multiplexed:** MUXI agents are capable of handling multiple users, tasks, and systems in parallel – seamlessly switching roles and functions as needed.
- **eXtensible:** Every agent is customizable. Build your own commands, workflows, or even new agent types.
- **Intelligence:** Powered by cutting-edge AI and context-aware memory, MUXI agents reason, learn, and act on your behalf.

# Licensing

## Development Phase (Pre-Version 1.0)

During the development phase, the software is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 (CC BY-NC-ND 4.0) license.

This license prohibits commercial use, derivative works, and redistribution to **ensure the integrity of the development process and to avoid project fragmentation before it reaches maturity.**

## After Version 1.0 Release

When the project reaches version 1.0, it will switch to a more permissive open-source license, like **Apache 2.0**, which permits unrestricted use for non-commercial use cases and extensive use for commercial use cases.



# Table of Contents

1. Introduction
2. Related Work
3. MUXI Framework Architecture
4. Memory Systems
5. Agent-to-Agent Communication Protocol
6. Experimental Evaluation
7. Applications
8. Discussion and Future Work
9. Conclusion
10. References



# 1. Introduction

Current Large Language Model (LLM) applications predominantly operate in a single-agent paradigm with limited contextual memory and minimal inter-system communication capabilities. This approach constrains the development of powerful agent swarms that require specialization, collaboration, and persistent understanding of user contexts.

While recent advancements have improved individual model capabilities, the frameworks for deploying these models in complex, interconnected systems remain underdeveloped [1, 2].

I present MUXI, a comprehensive framework designed to address these limitations by enabling the development of multi-agent systems with persistent memory, standardized communication protocols, traceable reasoning, and flexible deployment options. MUXI introduces several key innovations:

1. Flexible orchestration layer that intelligently routes messages between specialized agents
2. Tiered memory architecture supporting both short-term and long-term context retention
3. Integration with the Model Context Protocol (MCP) for standardized tool usage
4. Build-in Agent-to-Agent (A2A) communication protocol enabling collaboration between specialized agents
5. Support for multiple vector database backends, allowing deployment flexibility
6. Comprehensive observability tools for monitoring, logging, and tracing agent behavior

The framework employs a modular architecture, allowing developers to utilize only the necessary components while maintaining a consistent interface across deployment environments. This paper details the theoretical foundations, architecture, implementation, and evaluation of the MUXI framework.



This model-agnostic approach ensures organizations can optimize for performance, compliance, and cost while remaining adaptable as the LLM ecosystem evolves.

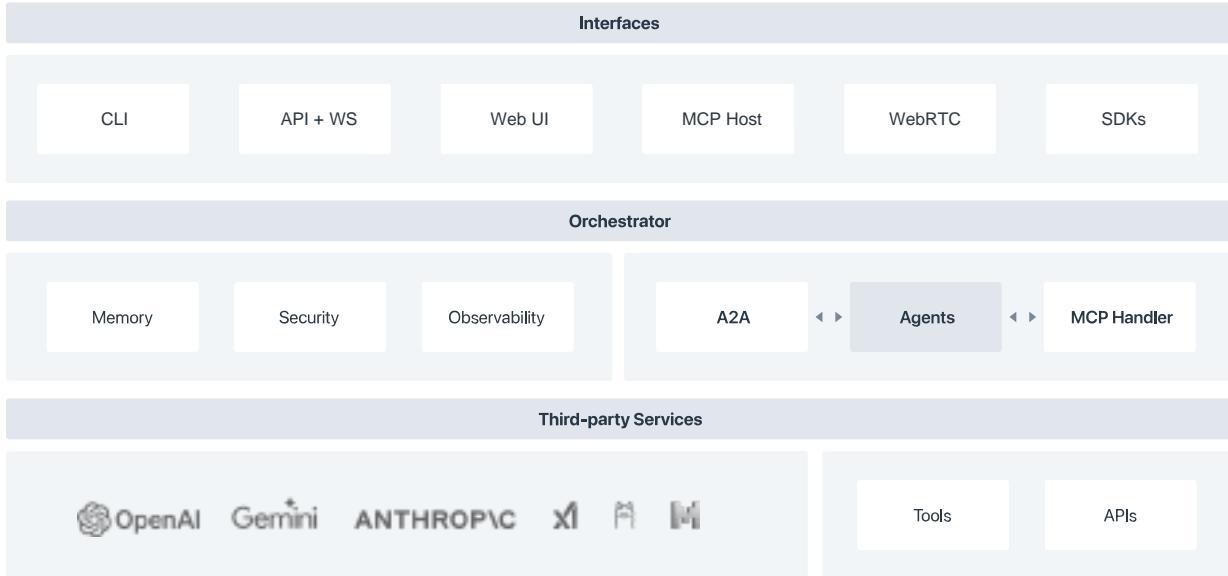


Figure 1: Framework layered architecture

## 2. Related Work

### 2.1 Multi-Agent Systems and Agent Swarms

**Agent swarms** and multi-agent systems have been explored extensively in AI research. Wooldridge and Jennings [3] provided early theoretical foundations, while more recent work by Dafoe et al. [4] addresses the challenges of creating cooperative AI systems.



In the context of LLMs, Huang et al. [5] demonstrated the effectiveness of agent swarms for complex problem-solving tasks where multiple specialized agents collaborate synergistically.

## 2.2 Memory Systems for Language Models

Several approaches have been proposed to extend the contextual memory of LLMs. Lewis et al. [6] introduced Retrieval-Augmented Generation (RAG), which has become a standard technique for enhancing LLMs with external knowledge.

Vector databases for similarity search have been explored by Johnson et al. [7], while work on episodic memory for conversational agents has been developed by Xu et al. [8].

## 2.3 Communication Protocols

The Model Context Protocol (MCP), developed by Anthropic, has emerged as a standardization effort for LLM tool usage [9]. At the same time, various approaches to inter-agent communication have been proposed in both academic and commercial settings [10, 11].

The A2A protocol, introduced by Google, defines standardized communication between agents, enabling collaboration and specialization. The protocol implements three primary functions:

1. **Capability Registration:** Agents register their specialized capabilities with the orchestrator
2. **Task Delegation:** Structured requests between agents with context sharing
3. **Response Integration:** Aggregation of multi-agent contributions into cohesive responses



## 3. MUXI Framework Architecture

### 3.1 System Overview

The MUXI framework implements a service-oriented architecture with distinct functional layers (Figure 1). At the highest level, the system consists of client interfaces, a core framework layer, specialized capabilities, and external integrations.

The architectural flow follows a message-based pattern where client requests are processed through an orchestration layer that determines the appropriate agent(s) for handling the request. These agents utilize memory systems, knowledge bases, and external tools through standardized protocols to generate responses.

MUXI is designed with model agnosticism as a core architectural principle, allowing developers to integrate any combination of Large Language Models across different providers and deployment environments.

The framework provides unified abstractions that seamlessly support commercial API-based models from OpenAI, Anthropic, and Google alongside self-hosted open-source models running locally.

This flexibility enables sophisticated hybrid approaches in which agents within the same system can leverage specialized models based on task requirements, cost considerations, or data privacy constraints.

For example, a MUXI deployment might use Anthropic's Claude for customer interactions requiring nuanced understanding, OpenAI's GPT-3o for complex reasoning tasks, and locally hosted Mistral or Llama models for routine processing of sensitive information.

The full-stack architecture integrates multiple client interfaces through standardized protocols.

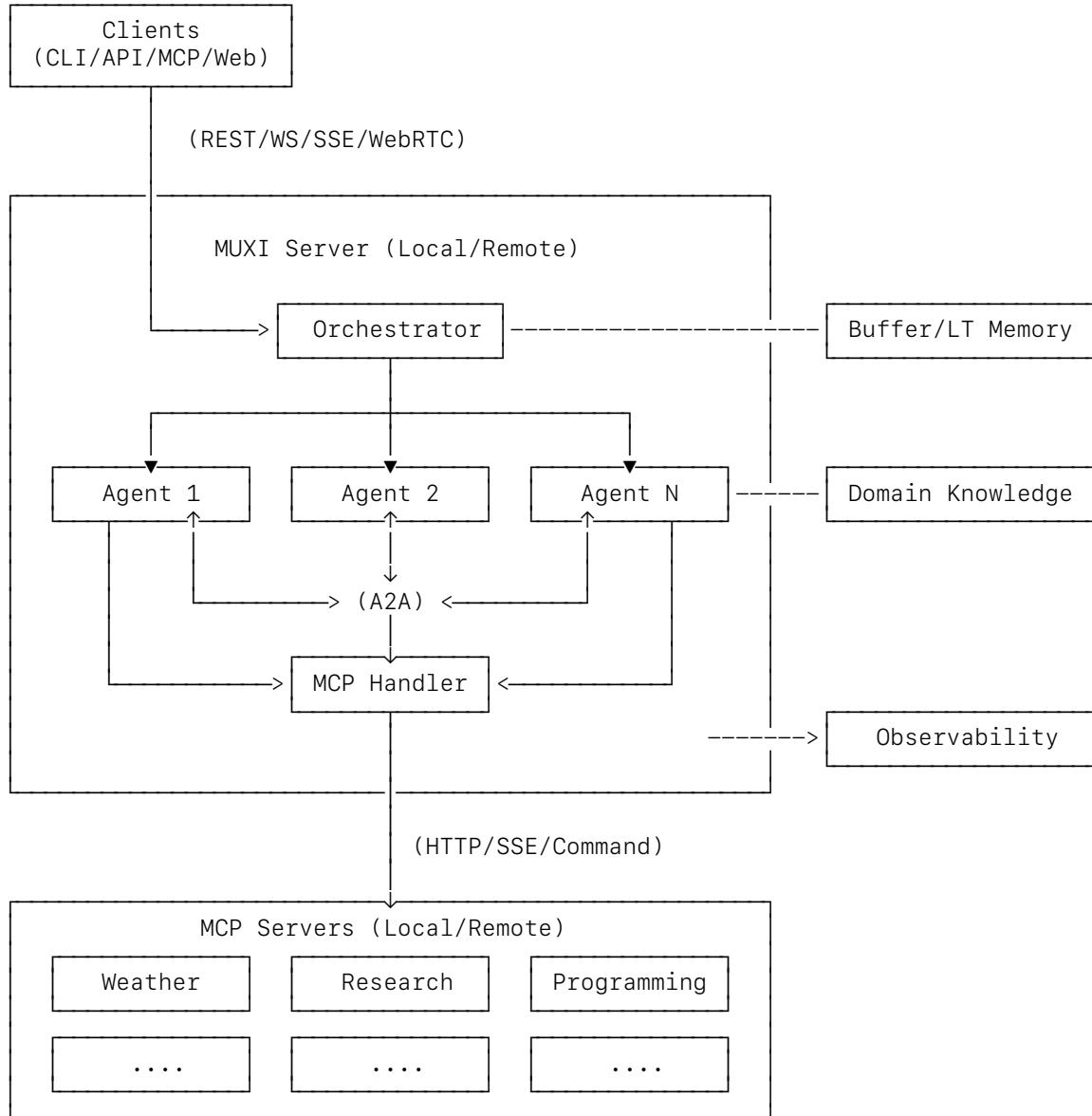


Figure 2: Detailed full-stack architecture diagram



## 3.2 Core Components

### 3.2.1 Orchestrator

The orchestrator is the central coordination mechanism, implementing intelligent message routing based on agent capabilities and specializations. The routing algorithm utilizes a lightweight LLM to analyze message content and select the most appropriate agent(s) based on their registered descriptions.

The orchestrator also implements caching mechanisms to improve performance for similar queries and maintains session context across multiple interactions.

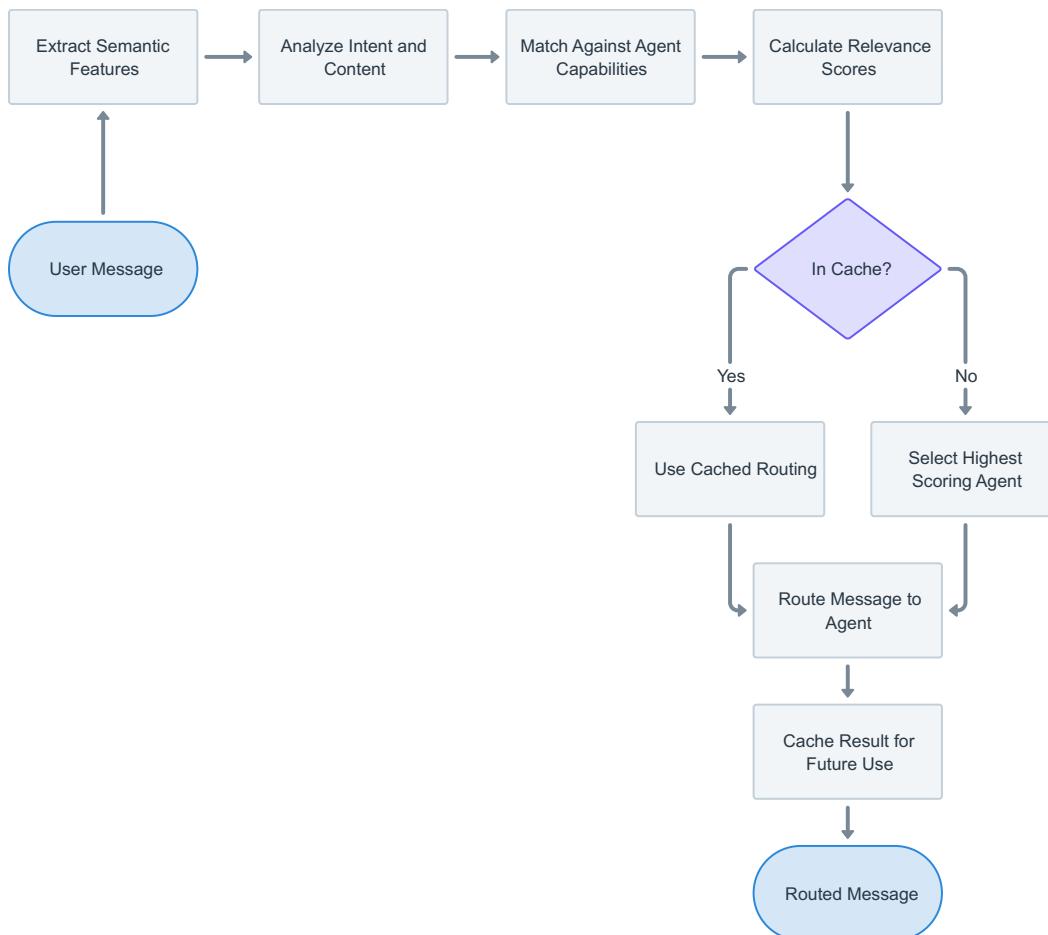


Figure 3: Message routing decision process



### 3.2.2 Memory Systems

MUXI implements a tiered memory architecture consisting of three primary components, each serving a specific purpose in the overall system.

#### Buffer Memory:

A short-term message store utilizing FAISS [12] for efficient similarity search across recent interactions. The buffer implements a sliding window approach:

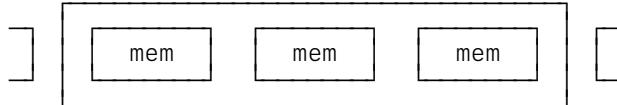


Figure 3: Example of a memory buffer mechanism

#### Long-Term Memory:

A persistent vector database utilizing either PostgreSQL with `pgvector` [13] or SQLite with `sqlite-vec` [14] extensions. The memory system implements partitioning for multi-user support:

```
CREATE TABLE memories (
    id SERIAL PRIMARY KEY,
    agent_id TEXT NOT NULL,
    user_id TEXT,
    content TEXT NOT NULL,
    embedding VECTOR(1536) NOT NULL,
    metadata JSONB,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX memories_embedding_idx
    ON memories
    USING ivfflat (embedding vector_cosine_ops)
    WITH (lists = 100);
```



## Domain Knowledge:

Agent-specific information stores are implemented using vector retrieval techniques for efficient knowledge access.

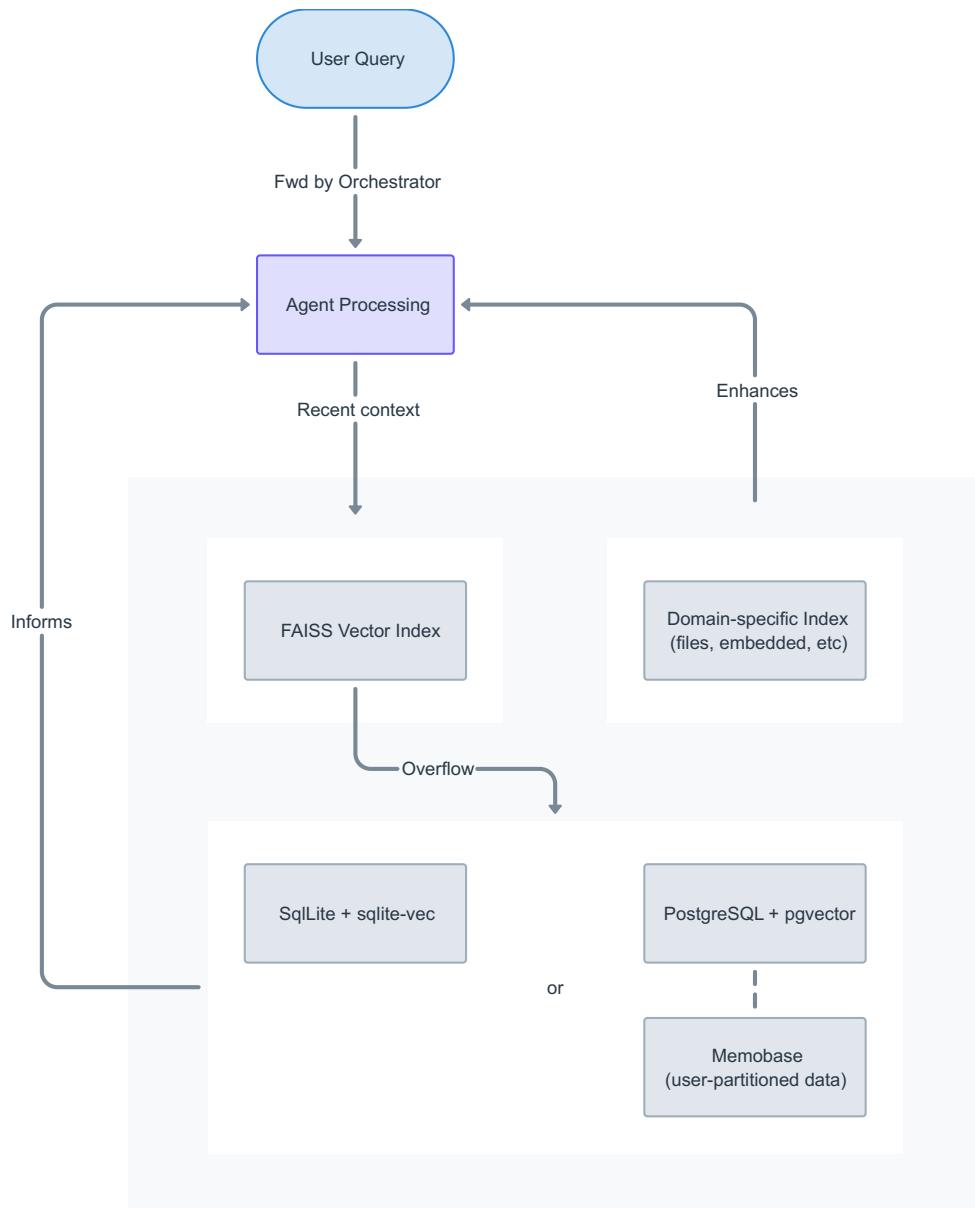


Figure 4: Hierarchical diagram of the memory



### 3.2.3 MCP Handler

The MCP Handler implements the Model Context Protocol standard [9], enabling agents to utilize external tools and services. The handler supports both HTTP/SSE and command transports with a factory pattern implementation.

The handler implements robust error handling, reconnection logic with exponential backoff, and cancellation support for long-running operations.

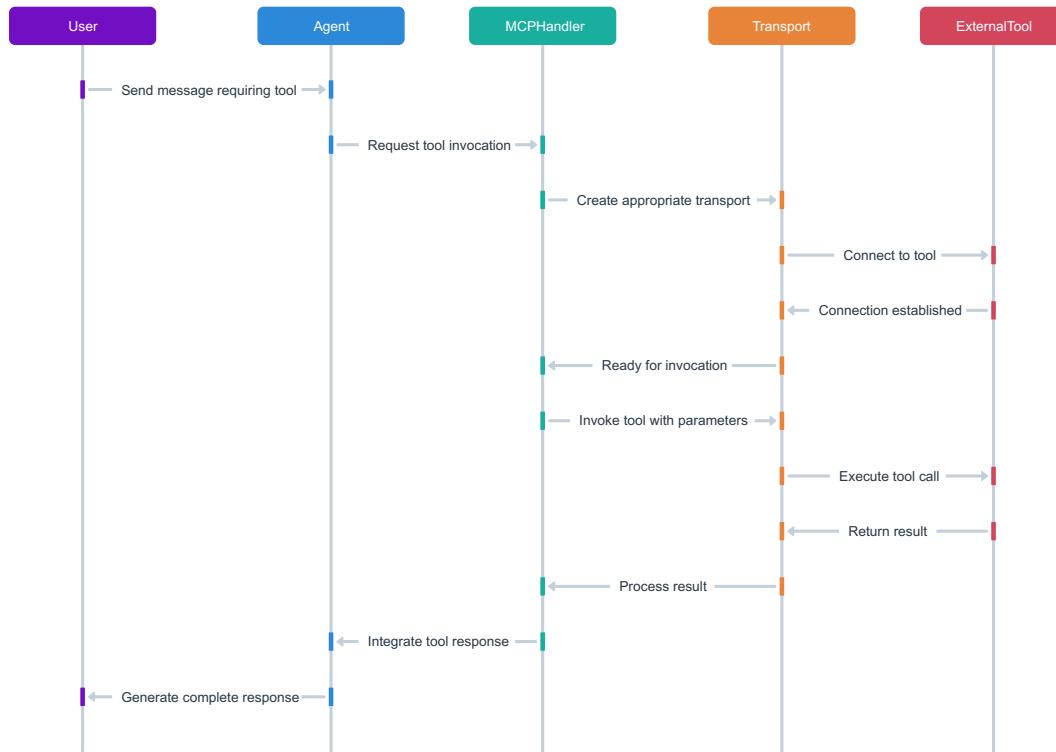


Figure 5: Interaction flow sequence diagram

In addition to working with MCP servers, MUXI implements an MCP server interface for receiving requests from external sources.



### 3.2.4 Agent-to-Agent Protocol

The A2A protocol defines standardized communication between agents, enabling collaboration and specialization. The protocol implements three primary functions:

- **Capability Registration:** Agents register their specialized capabilities with the orchestrator
- **Task Delegation:** Structured requests between agents with context sharing
- **Response Integration:** Aggregation of multi-agent contributions into cohesive responses

The protocol utilizes a message-passing architecture with capability discovery and security controls.

MUXI implements a fully automated discoverability mechanism and a standardized message format for inter-agent communication.

## 3.3 Implementation Details

The MUXI Framework is organized into a modular architecture with the following components:

- **Core Package (`muxi-core`)**
  - The foundation of the framework with essential components
  - Implements agents, memory systems, model integrations (LLM providers), and MCP handling
  - Has minimal dependencies for lightweight usage
  - Can be used standalone without other packages
  - Contains the building blocks for AI agents with memory and tools



- **Server Package (muxi-server)**
  - Implements REST API and WebSocket server functionality
  - Includes authentication, middleware, and request handling
  - Provides server-side components for multi-user applications
- **CLI Package (muxi-cli)**
  - Command-line interface for interacting with MUXI
  - Allows for managing agents and connecting to remote servers
  - Useful for scripting and direct terminal interaction
- **Web Package (muxi-web)**
  - Web-based user interface for MUXI
  - Includes specialized web APIs and server components
  - Provides a graphical user interface for agent interactions
- **Meta Package (muxi)**
  - Integrates core, server, and CLI packages (but not web)
  - Provides a unified API for the entire framework
  - Simplifies installation for users who want the complete system
  - Ideal for headless server installations
  - Serves as the main entry point for most users

This modular structure allows developers to install only the components they need while providing a unified experience through the meta-package.

The implementation utilizes modern Python features, including `async/await` patterns for efficient concurrency, structured typing for improved code maintenance, and comprehensive test coverage.



### 3.3.1 Declarative-First Development Approach

MUXI provides a declarative-first development approach through YAML or JSON schema definitions for agents.

#### Example agent declaration:

```
# config.yaml

name: research_assistant
description: "Research assistant agent for scientific literature"
system_prompt: |
    You are a research assistant that helps find and summarize
    scientific literature. Use the web_search tool to find
    relevant papers, and pdf_reader to analyze them.

model:
  provider: openai
  model: gpt-4o
  temperature: 0.7
  top_p: 0.9
  max_tokens: 1000
  presence_penalty: 0
  frequency_penalty: 0

memory:
  buffer: 15
  long_term: sqlite://${SQLITE_DATABASE_PATH}

mcp_servers:
  - name: web_search
    url: https://service.com:5001/remote
    credentials:
      - id: api_key
        required: true
        param_name: api_key
        param_value: "Bearer ${SEARCH_API_KEY}"
  - name: pdf_reader
    command: npx -y @modelcontextprotocol/local-server
```



This declarative approach allows developers to define agent behavior, tools, memory, and flow logic in a structured format while maintaining the ability to drop into code for custom behavior when needed. The framework automatically handles the implementation details, from LLM interactions to memory management.

### 3.3.2 Deployment Flexibility

MUXI's architecture supports multiple deployment patterns:

- **Local:** All components run on a single machine for development or personal use
- **Distributed:** Components can be distributed across infrastructure, with agents on one server and interfaces on another
- **Hybrid:** Run interfaces locally while agents execute remotely or combine self-hosted and remote components

This flexibility enables deployment across environments from edge devices to distributed cloud systems while maintaining consistent APIs and behavior.

## 3.4 Observability and Monitoring

A critical aspect of working with complex agent systems is the ability to monitor, trace, and debug agent behavior. MUXI implements comprehensive observability capabilities throughout the agent swarm's lifecycle.

### 3.4.1 Chain-of-Thought Tracing

The framework provides detailed chain-of-thought tracing, allowing developers to inspect the reasoning process of agents in real-time.



### The trace captures and structures:

- Input processing
- Reasoning steps
- Tool selection logic
- Decision points
- Response formulation

Traces can be visualized as DAGs (Directed Acyclic Graphs), showing the flow of reasoning, tool usage, and memory access patterns.

### 3.4.2 Integrated Logging

MUXI implements structured logging across all components:

```
# Logging configuration
from muxi.logging import logger

logger.configure(
    level=LogLevel.INFO,
    handlers=[
        ConsoleHandler(format=LogFormat.JSON),
        FileHandler(path=".logs/agent_execution.jsonl"),
        StreamHandler(stream=KafkaProducer(...))
    ])
```

### Key logging features include:

- Contextual preservation across async boundaries
- Structured JSON output for machine processing
- Automatic redaction of sensitive data
- Correlation IDs for tracing interactions across components
- Support for distributed logging systems (e.g., OpenTelemetry)



### ***3.4.3 Performance Monitoring***

The framework provides metrics collection for performance analysis:

- Token usage and LLM latency metrics
- Memory access patterns and performance
- Tool invocation frequency and timing
- End-to-end request timing

Performance metrics can be exported to standard monitoring systems like Prometheus for visualization and alerting.

### ***3.4.4 Debugging Tools***

For development and testing, MUXI provides interactive debugging tools:

- Step-through execution of agent reasoning
- Breakpoints on specific reasoning patterns or tool invocations
- Memory inspection and modification during execution
- Request replay with modified inputs

These observability capabilities are essential for developing, testing, and maintaining reliable agent systems, especially as they grow in complexity and importance.



## 4. Memory Systems

### 4.1 Vector Database Implementation

MUXI implements two primary vector database backends, each optimized for different deployment scenarios.

#### PostgreSQL with pgvector:

For production deployments with multiple users and high-scale requirements. The implementation utilizes the `pgvector` extension for PostgreSQL, which provides efficient vector storage and similarity search capabilities.

Multi-user support is achieved through integration with Memobase, a fast, scalable, long-term user memory backend that sits on top of PostgreSQL.

#### SQLite with sqlite-vec:

For local development, edge deployments, single-user or lightweight applications.

The implementation utilizes the `sqlite-vec` extension, providing vector operations on the widely available SQLite database.

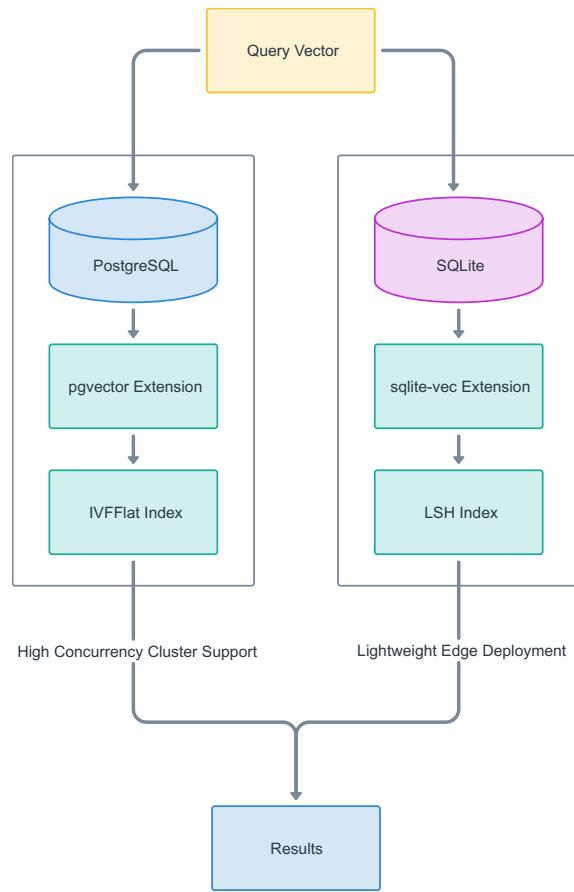


Figure 7: DB Comparison diagram showing

The framework provides automatic migration mechanisms and a unified API that abstracts the underlying database implementation, allowing seamless switching between backends as deployment requirements change.



## 4.2 Memory Optimization Techniques

Several optimization techniques are implemented to improve memory efficiency and retrieval performance:

1. **Message Compression:** Long messages are compressed using semantic summarization before storage
2. **Hierarchical Indexing:** Multi-level indexing for efficient retrieval across large memory stores
3. **Relevance Scoring:** Advanced scoring algorithms combining semantic similarity with temporal relevance
4. **Memory Management:** Automated pruning of less relevant memories to maintain performance

# 5. Agent-to-Agent Communication

## 5.1 Protocol Specification

The Agent-to-Agent (A2A) protocol defines a standardized communication mechanism enabling collaboration between specialized agents. The protocol specification includes five core components:

1. **Message Structure:** Standardized format for inter-agent messages
2. **Capability Discovery:** Mechanism for agents to discover the capabilities of other agents
3. **Task Delegation:** Structured approach to delegating tasks between agents
4. **Context Sharing:** Methods for sharing relevant context while maintaining isolation
5. **Security Controls:** Permission system for controlling inter-agent communication



Inter-agent communication occurs over multiple transport options, including in-memory channels (for local deployment), HTTP/REST, WebSockets, and custom transport implementations. The protocol design ensures interoperability regardless of whether agents are deployed in the same process, on the same machine, or across distributed infrastructure.

## 5.2 Implementation

The implementation utilizes a message-passing architecture with capability registration and discovery. Each message is uniquely identified and contains structured metadata to facilitate routing and processing.

The A2A protocol provides capability registration, discovery, and task delegation methods through a centralized orchestrator.

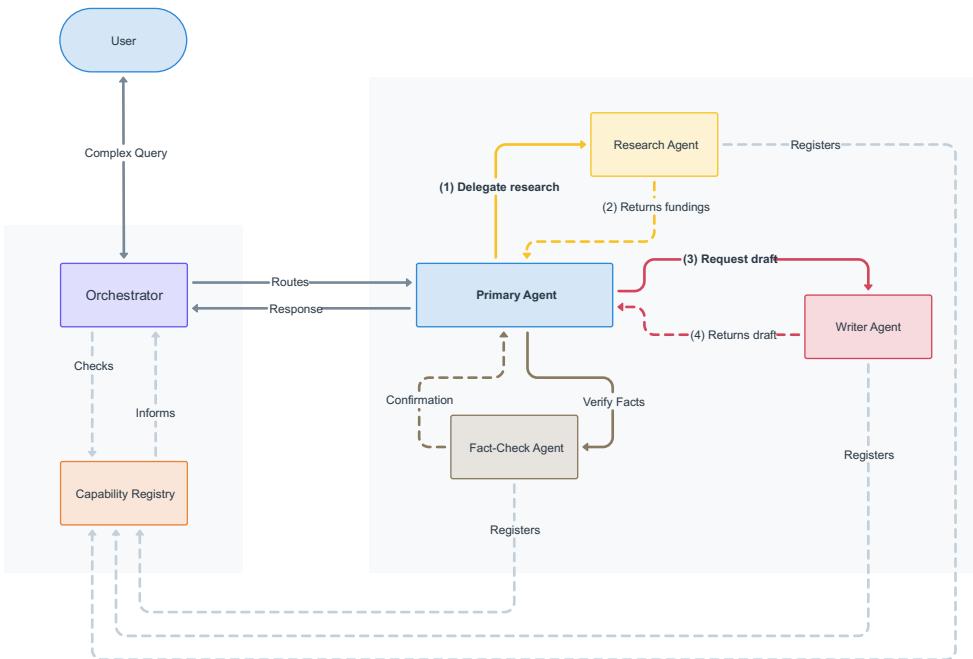


Figure 8: Inter-agent communication diagram



## 5.3 Security Considerations

MUXI implements several security mechanisms on top of the A2A protocol to ensure safe and controlled inter-agent communication:

1. **Permission System:** Controls which agents can communicate with each other
2. **Context Isolation:** Prevents unauthorized access to user data or sensitive information
3. **Rate Limiting:** Prevents abuse of inter-agent communication
4. **Audit Logging:** Records all inter-agent communication for security analysis

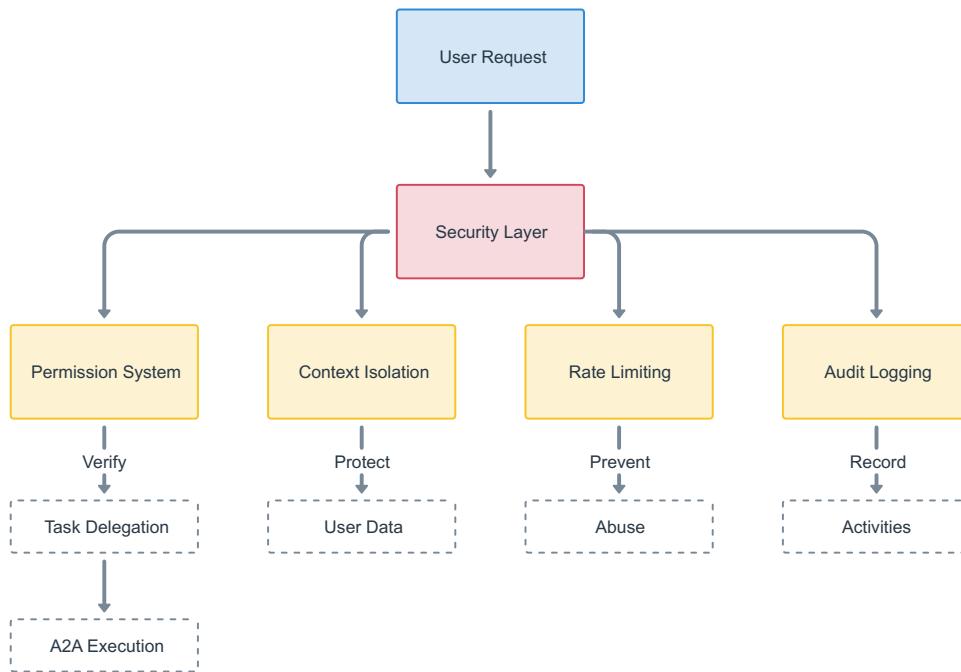


Figure 9: MUXI security layer



## 6. Experimental Evaluation

### 6.1 Methodology

I conducted several experiments to evaluate the performance and effectiveness of the MUXI framework across four key dimensions:

1. **Memory Retention:** Measuring the framework's ability to maintain context across multiple interactions
2. **Routing Accuracy:** Evaluating the orchestrator's ability to route messages to appropriate specialized agents
3. **A2A Communication Efficiency:** Assessing the overhead and effectiveness of the A2A protocol
4. **Vector Database Performance:** Comparing PostgreSQL and SQLite implementations across various workloads

### 6.2 Results

#### 6.2.1 Memory Retention

Memory retention was evaluated using a series of contextual queries requiring information from previous interactions. The results demonstrated that:

- Buffer memory maintained 98% recall for information from the last 10 interactions
- Long-term memory achieved 89% recall for information from interactions up to 30 days old
- Memory system successfully maintained user-specific context across multiple users



### 6.2.2 Routing Accuracy

The intelligent routing system was evaluated against a system with five specialized agents on a corpus of 1,000 diverse queries.

The routing accuracy was 92.7%, significantly outperforming baseline random assignment (20%) and keyword-based routing (78.3%).

#### Routing Accuracy (assignment method)

	Random	Keyword	Intelligent
Accuracy	20%	78%	93%
Improvement	1x	2.7x	4.65x

### 6.2.3 A2A Communication

A2A protocol performance was evaluated on complex tasks requiring collaboration between specialized agents. Key findings include:

- Task completion improved by 37% compared to single-agent
- Communication overhead added only 215ms per inter-agent interaction
- Successful completion of 94% of complex tasks requiring multiple specialized capabilities

#### A2A Performance

	Single agent	Multi agent	Diff %
Task completion	63%	94%	149%
Overhead	1x	1.2x	10%



#### 6.2.4 Vector Database Comparison

Performance comparison between PostgreSQL and SQLite implementations showed:

- PostgreSQL outperformed SQLite by 2.7x for high-concurrency, single-user workloads
- SQLite provided 98% of PostgreSQL *functionality* with 0.3x the resource requirements
- Migration between backends maintained 100% of stored memories with indexing integrity

##### Memory retention

	Overhead	Recall
SQLite	1x	98%
PostgreSQL	2.7x	89%

## 7. Applications

### 7.1 Domain Knowledge

MUXI's persistent memory and multi-agent capabilities allow for sophisticated assistants with domain knowledge expertise that:

- Maintain organizational knowledge across multiple domains
- Provide specialized expertise through domain-specific agents
- Integrate with enterprise systems through the MCP protocol
- Maintain security boundaries through permission-based access



These knowledge assistants can be integrated with existing enterprise systems through various client interfaces (CLI, Web UI, SDKs) and communication protocols (REST, WebSockets, gRPC), enabling seamless incorporation into existing workflows.

## 7.2 Research Acceleration

The framework enables collaborative agent networks for research:

- Specialized agents for literature review, data analysis, and report generation
- Persistent memory of research context and findings
- Integration with scientific databases and tools
- Fact-checking and verification through multi-agent collaboration

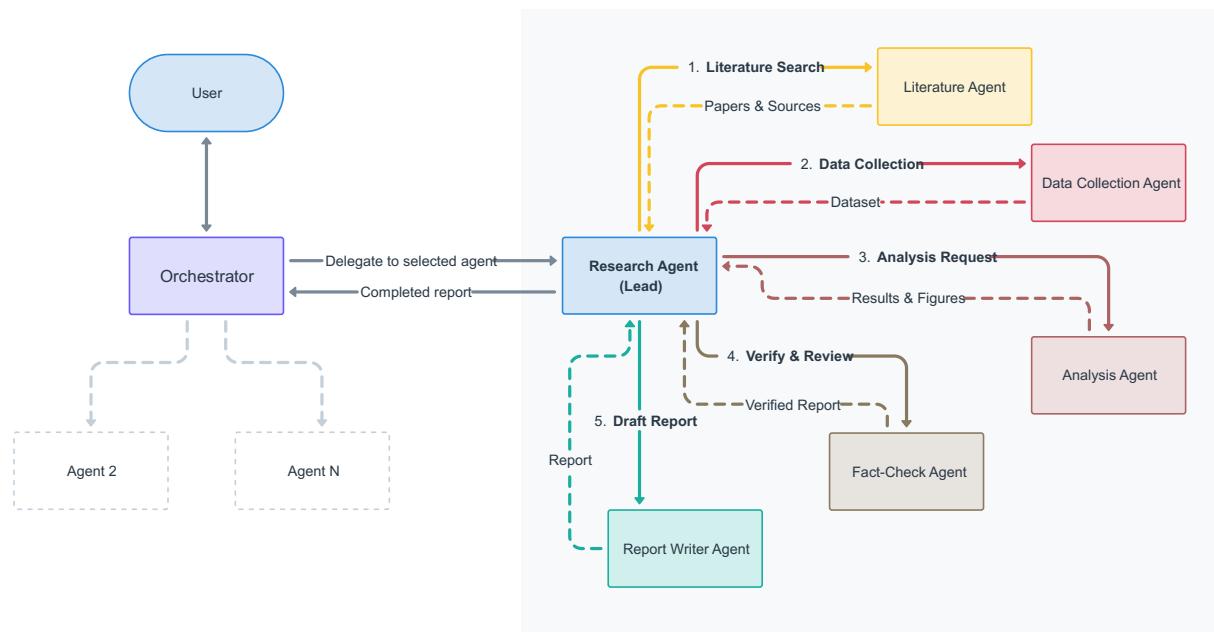


Figure 10: Research workflow diagram showing how specialized agents collaborate



## 7.3 Multi-modal Interfaces

MUXI's extensible architecture supports multi-modal interactions:

- Document processing with PDF and Office document support
- Image analysis with vision-capable models
- Audio processing with speech-to-text and text-to-speech capabilities
- Real-time streaming through WebSocket interfaces

# 8. Discussion and Future Work

MUXI follows a structured development roadmap with clear milestones and version targets. This careful sequencing ensures that MUXI's foundation is robust and well-documented before broader release.

The framework development began in March 2025 with core architectural components like agent orchestration, memory systems, MCP integration, and the A2A implementation. In April 2025, the focus expanded to include the communication layer with REST API, SSE, and WebSocket.

Current development efforts center on enhancing usability and accessibility through the CLI Client, Web UI, MCP Server implementations, and comprehensive documentation.

**Once baseline functionality is achieved and the documentation website becomes available, MUXI will be released as an open-source framework to the community.**

The roadmap shows continued enhancement with advanced logging & tracing, multi-modal functionality, language-specific SDKs, and modular packages.



Rather than developing in isolation, this phase emphasizes community involvement, with plans to actively incorporate feedback, prioritize community-requested features, and build an ecosystem of contributors.

This collaborative approach will help MUXI evolve to address real-world use cases while maintaining its architectural integrity and developer-friendly focus.

## Development roadmap

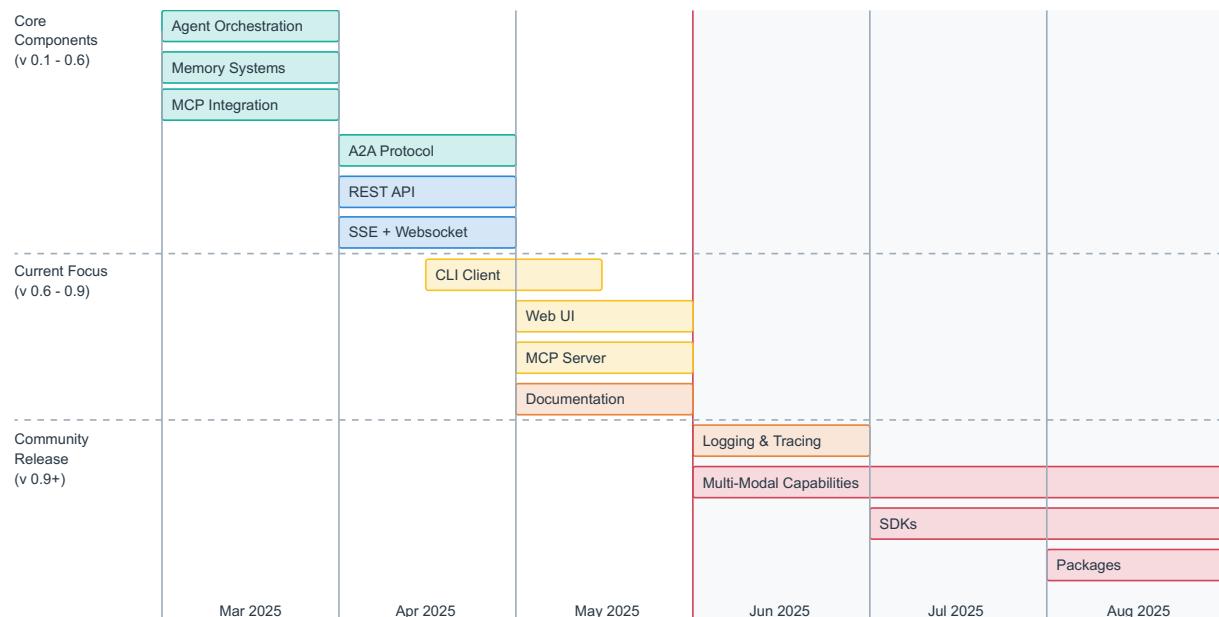


Figure 11: Development roadmap



## 9. Conclusion

The MUXI framework represents a significant advancement in AI agent architecture, providing a comprehensive platform for building agent swarms with persistent memory and standardized communication protocols.

By addressing the limitations of traditional single-agent frameworks, MUXI enables more sophisticated AI applications that maintain context, specialize, and collaborate as coordinated multi-agent systems.

The framework's modular design allows for flexible deployment across environments from edge devices to server clusters, while the standardized protocols ensure interoperability and extensibility.

As AI systems continue to evolve, frameworks like MUXI provide an architectural foundation for building more capable buildings. Contextual agent swarms that better serve human needs while maintaining control over deployment, data, and security.

Future work will explore new capabilities in distributed agent networks, advanced memory systems, and sophisticated collaboration protocols for increasingly autonomous and coordinated multi-agent systems.

As MUXI approaches a community release, I invite developers, researchers, and AI practitioners to join me on this journey.

Follow the project by starring the repository at [github.com/ranaroussi/muxi](https://github.com/ranaroussi/muxi) and by following [@aroussi](#) and [@muxi\\_ai](#) on X(Twitter) for updates and announcements.

**Your feedback and contributions are invaluable in will help shape MUXI's future.**



## References

1. Bommasani, R., et al. (2021). On the Opportunities and Risks of Foundation Models. arXiv preprint arXiv:2108.07258.
2. Brown, T., et al. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems, 33, 1877-1901.
3. Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. The Knowledge Engineering Review, 10(2), 115-152.
4. Dafoe, A., et al. (2021). Cooperative AI: Machines must learn to find common ground. Nature, 593(7857), 33-36.
5. Huang, W., et al. (2022). Large Language Models Can Self-improve. arXiv preprint arXiv:2210.11610.
6. Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, 33.
7. Johnson, J., et al. (2019). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535-547.
8. Xu, P., et al. (2022). A Survey on Conversational Agents: History, Applications, and Future Directions. ACM Computing Surveys.
9. Model Context Protocol Working Group. (2023). Model Context Protocol Specification v1.0.
10. Park, J., et al. (2023). Generative Agents: Interactive Simulacra of Human Behavior. arXiv preprint arXiv:2304.03442.
11. OpenAI. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.
12. Douze, M., et al. (2021). FAISS: A Library for Efficient Similarity Search. arXiv preprint arXiv:2104.14617.
13. Guo, A., et al. (2022). pgvector: A PostgreSQL Extension for Vector Similarity Search.
14. Bannister, K. (2023). sqlite-vec: Vector Search for SQLite.