Maiar: A Composable, Plugin-Based AI Agent Framework

Uranium Corporation

February 11, 2025

Abstract

Maiar is a powerful framework for building AI agents that introduces a novel plugin-based architecture inspired by Unix pipes. By abstracting AI agent functionality into modular, composable plugins and leveraging dynamic LLM-driven decision making, Maiar enables developers to build flexible, extensible AI systems without being constrained by rigid workflows or monolithic architectures.

1 Introduction

The field of artificial intelligence is experiencing rapid evolution, particularly in the development of AI agents that can interact with various systems and services. However, current approaches to building AI agents often suffer from rigid architectures, monolithic codebases, and inflexible workflows that make it difficult to adapt and extend functionality as requirements evolve. This challenge is particularly acute as AI capabilities expand and use cases diversify, requiring frameworks that can seamlessly incorporate new features while maintaining system coherence.

Maiar addresses these limitations by introducing a novel, plugin-based architecture inspired by Unix pipes. The framework is built around the thesis that AI agents primarily consist of three major steps: data ingestion and triggers, decision-making, and action execution. Rather than implementing these components in a tightly coupled manner, Maiar abstracts them into a modular, plugin-based system where developers can define triggers and actions as standalone components while the core runtime dynamically handles decision-making through LLM-assisted reasoning.

This approach represents a fundamental shift in how AI agents are constructed. Instead of predetermined workflows, Maiar produces emergent behavior by dynamically selecting and composing relevant plugins based on context. This enables AI agents to evolve and adapt without requiring extensive rewrites of core logic, while maintaining the reliability and predictability necessary for production systems.

2 Background and Related Work

Recent advances in AI agent frameworks have made significant strides in making AI systems more accessible and powerful. Notable among these is Eliza [1], which pioneered several key concepts in modern AI agent architectures. Eliza's provider-action-evaluator

chain introduced a structured approach to building AI agents, demonstrating how complex behaviors could emerge from well-defined architectural patterns.

However, while Eliza made groundbreaking contributions to the field, its rigid architectural constraints present limitations for certain use cases. The fixed provider-action-evaluator chain, while elegant in its simplicity, can become a constraint when developers need to implement more complex interaction patterns. For example, adding pre-action evaluators or post-evaluator providers requires fundamental changes to the core architecture, as the system wasn't designed for such flexible compositions.

Traditional approaches to building AI agents have typically followed one of several patterns, each with its own limitations:

- **Fixed Pipeline Architectures:** Systems like Eliza implement a predetermined chain of operations. While this approach provides clarity and predictability, it can limit the emergence of complex behaviors that arise from more flexible compositions.
- **Monolithic Architectures:** Many frameworks implement agent logic as a single, tightly coupled system. While this approach can be effective for simple use cases, it becomes increasingly difficult to maintain and extend as the system grows.
- **Rule-Based Systems:** Some frameworks rely heavily on predefined rules and decision trees to determine agent behavior. While these systems can be predictable and easy to debug, they lack the flexibility to handle novel situations.

Our thesis is that by making the building blocks simpler and more composable, we can enable even more complex emergent behaviors than those possible with fixed architectural patterns. This insight draws inspiration from Unix pipes [2], where simple, single-purpose tools can be combined in countless ways to create sophisticated workflows. Just as Unix pipes enable processes to communicate through a simple read-write interface without knowing the details of their communication channel, Maiar's plugins communicate through a standardized context chain that abstracts away the complexity of inter-plugin interactions.

Maiar builds upon these foundations while addressing their limitations through several key innovations:

- **Plugin-First Architecture:** By treating every capability as a plugin, Maiar achieves true modularity without sacrificing system coherence.
- **Dynamic Pipeline Construction:** Rather than enforcing a fixed chain of operations, Maiar allows dynamic construction of processing pipelines based on context and requirements.
- **Unix-Style Composition:** Drawing inspiration from Unix pipes, Maiar enables seamless composition of plugins through a standardized context chain interface.

3 Technical Overview

Maiar's architecture is built around three core principles: modularity through plugins, dynamic execution through LLM-driven decision making, and composability through context chains. This section details the key components and their interactions.

3.1 Core Architecture

The framework consists of several key components:

- **Runtime:** The central orchestrator that manages plugins, handles the event queue, and provides essential services for plugin interaction.
- **Plugin System:** A flexible architecture where each plugin can provide triggers (event listeners) and executors (actions).
- **Model Provider System:** An abstraction layer for integrating various Language Models (LLMs) with standardized interfaces.
- **Memory Provider System:** A flexible storage system for maintaining conversation history and context across interactions.

3.2 Plugin Architecture

Plugins in Maiar follow a Unix-inspired pipeline architecture where:

- Data flows through a sequence of operations
- Each plugin acts as an independent unit
- Plugins can be composed to create complex behaviors
- Context is passed and transformed along the chain

3.3 Context Chain

The context chain is central to Maiar's pipeline architecture:

```
[Trigger] → [Initial Context] → [Executor 1] → [Executor 2] → [Response]
```

Each step in the pipeline can:

- Read from the context
- Modify or enhance the context
- Pass the modified context forward

3.4 LLM Integration

Maiar's model provider system offers a simple interface for integrating any Language Model:

```
interface ModelProvider {
    init?(): Promise<void>;
    getText(prompt: string, config?: ModelRequestConfig): Promise<string>;
}
```

This simplicity enables:

- Easy integration of new LLM providers
- Custom provider implementations
- Wrapping existing providers to add functionality

4 Implementation Details

This section provides a detailed look at implementing and using Maiar in practice.

4.1 Installation and Setup

Getting started with Maiar is straightforward:

```
# Create a new project
mkdir my-maiar-agent
cd my-maiar-agent
pnpm init

# Install core dependencies
pnpm add @maiar-ai/core @maiar-ai/model-openai \
  @maiar-ai/memory-sqlite @maiar-ai/plugin-express \
  @maiar-ai/plugin-text dotenv
```

4.2 Basic Implementation

A minimal Maiar implementation requires:

```
import "dotenv/config";
import { createRuntime } from "@maiar-ai/core";
import { OpenAIProvider } from "@maiar-ai/model-openai";
import { SQLiteProvider } from "@maiar-ai/memory-sqlite";
import { PluginExpress } from "@maiar-ai/plugin-express";
import { PluginTextGeneration } from "@maiar-ai/plugin-text";
import path from "path";

const runtime = createRuntime({
  model: new OpenAIProvider({
    apiKey: process.env.OPENAI_API_KEY,
    model: "gpt-3.5-turbo"
  }),
  memory: new SQLiteProvider({
    dbPath: path.join(process.cwd(), "data", "conversations.db")
  }),
  plugins: [
    new PluginExpress({ port: 3000 }),
    new PluginTextGeneration()
  ]
});

runtime.start();
```

4.3 Creating Custom Plugins

Plugins in Maiar are highly customizable. A basic plugin structure includes:

- **Triggers:** Event listeners that determine when the agent should act
- **Executors:** Actions that the agent can perform
- **Context Handlers:** Functions for modifying the context chain

Example of a custom plugin:

```
class CustomPlugin implements Plugin {
  readonly id = "custom-plugin";
  readonly name = "Custom Plugin";
  readonly description = "Handles custom functionality";

  async init(runtime: Runtime): Promise<void> {
    // Plugin initialization logic
  }

  getTriggers(): Trigger[] {
    return [
      {
        id: "custom-trigger",
        match: (event) => event.type === "custom",
        handle: async (event) => {
          // Trigger handling logic
        }
      }
    ];
  }

  getExecutors(): Executor[] {
    return [
      {
        id: "custom-action",
        execute: async (context) => {
          // Action execution logic
        }
      }
    ];
  }
}
```

4.4 Memory Management

Maiar provides a flexible memory system for maintaining conversation state:

```
interface MemoryProvider {
  storeMessage(message: Message, conversationId: string): Promise<void>;
  getMessages(options: MemoryQueryOptions): Promise<Message[]>;
  createConversation(options?: {
```

```
    id?: string;
    metadata?: Record<string, any>;
  }): Promise<string>;
}
```

This interface can be implemented for various storage solutions:

- SQLite for local development
- MongoDB for document storage
- Redis for high-performance caching
- Custom implementations for specific needs

5 Use Cases and Applications

Maiar's flexible architecture makes it suitable for a wide range of applications and use cases. This section explores some key scenarios where Maiar provides significant value.

5.1 Chatbots and Virtual Assistants

Maiar excels in building sophisticated conversational agents:

- **Customer Service:** Handle customer inquiries across multiple platforms with consistent behavior
- **Virtual Assistants:** Create personal assistants that can learn and adapt to user preferences
- **Educational Bots:** Develop interactive learning experiences with contextual awareness

5.2 System Integration and Automation

The plugin architecture makes Maiar ideal for system integration:

- **DevOps Automation:** Create agents that can monitor systems and respond to incidents
- **Workflow Automation:** Build intelligent processes that can adapt to changing conditions
- **Data Pipeline Management:** Orchestrate complex data flows with intelligent decision-making

5.3 Research and Development

Maiar provides a powerful platform for AI research:

- **Prototype Development:** Quickly test new AI agent architectures and behaviors
- **Model Evaluation:** Compare different LLM providers and configurations
- **Behavior Analysis:** Study emergent behaviors in AI systems

5.4 Enterprise Applications

Organizations can leverage Maiar for various business needs:

- **Knowledge Management:** Create intelligent systems for organizing and accessing information
- **Process Automation:** Streamline business processes with adaptive AI agents
- **Customer Engagement:** Build personalized interaction systems across multiple channels

5.5 Platform Integration

Maiar's plugin system supports various platforms:

- **Chat Platforms:** Telegram, Discord, Slack, etc.
- **Web Services:** REST APIs, WebSocket servers
- **Custom Interfaces:** Command-line tools, desktop applications

6 Roadmap and Future Work

As Maiar continues to evolve, we are focusing on three transformative areas that will significantly enhance the framework's capabilities and developer experience.

6.1 Plugin Ecosystem Platform

We are developing a comprehensive platform to support the Maiar plugin ecosystem:

- **Plugin Registry:** A centralized marketplace for discovering, sharing, and managing plugins
- **Plugin Analytics:** Tools for tracking plugin usage, performance metrics, and community engagement
- **Collaborative Development:** Infrastructure for community contributions and plugin maintenance
- **Version Management:** Sophisticated tooling for managing plugin dependencies and compatibility
- **Quality Assurance:** Automated testing and validation systems for plugin submissions

6.2 Multi-Modal Model Integration

We are expanding Maiar's capabilities to handle multiple AI modalities with intelligent context switching:

- **Dynamic Model Selection:** Intelligent routing of requests to the most appropriate model based on context
- **Cross-Modal Reasoning:** Seamless integration of text, image, audio, and video understanding
- **Context-Aware Switching:** Automatic model switching based on task requirements and performance metrics
- **Unified Context Management:** Cohesive handling of context across different modalities
- **Hybrid Model Pipelines:** Support for combining multiple models in single processing chains

6.3 Bleeding Edge AI Agent Development Tools

We are building next-generation tools to revolutionize AI agent development:

- **Visual Plugin Builder:** Interactive tools for designing and testing plugin chains
- **Real-Time Debugging:** Advanced visualization and inspection of agent decision-making processes
- **Behavior Simulation:** Tools for testing agent behavior in controlled environments
- **Performance Profiling:** Sophisticated analytics for optimizing agent performance
- **Development IDE Integration:** Seamless integration with popular development environments

7 Conclusion

Maiar represents a significant advancement in the field of AI agent development, introducing a novel approach that combines the flexibility of plugin-based architectures with the power of LLM-driven decision making. By drawing inspiration from Unix pipes and emphasizing modularity and composability, Maiar provides a robust foundation for building the next generation of AI applications.

The framework's key innovations—plugin-first architecture, dynamic execution pipelines, and standardized context chains—address many of the limitations found in traditional agent frameworks. This enables developers to create more adaptable, maintainable, and scalable AI systems while reducing the complexity typically associated with agent development.

As the AI landscape continues to evolve, Maiar's extensible architecture positions it well to incorporate new advances in language models, memory systems, and agent architectures. The framework's growing ecosystem of plugins and tools, combined with

its strong focus on developer experience and enterprise readiness, makes it a compelling choice for organizations looking to leverage AI agents in their applications.

The future of AI agents lies in frameworks that can adapt to changing requirements while maintaining reliability and security. Maiar's approach to these challenges, along with its comprehensive roadmap for future development, suggests it will play a significant role in shaping how AI agents are built and deployed in the years to come.

References

[1] Walters, S., Gao, S. et al., *Eliza: A Web3 friendly AI Agent Operating System*, arXiv preprint arXiv:2501.06781, 2025.

[2] Ritchie, Dennis M. and Thompson, Ken, *The UNIX Time-Sharing System*, Communications of the ACM, 1974.