Architectural Redesign Report: MCP Registry and Management Platform

Author: Manus AI **Date:** December 18, 2025 **Project Goal:** To redesign the MCP Messenger architecture from scratch, transforming it into a highly scalable, maintainable, and cloud-native Microservices Architecture for the MCP Registry and Management Platform.

1. Executive Summary

The proposed architecture is a complete overhaul, moving from a likely monolithic structure to a **Cloud-Native Microservices Architecture** centered around an **Event-Driven Architecture (EDA)** using **Apache Kafka**. This design directly addresses technical debt, maximizes maintainability, and provides the necessary foundation for advanced multimodal capabilities (Voice, Vision, Document Support) and the core function of an MCP Registry.

The key features of this design are:

- **Decoupling:** Services are independent, allowing for separate development, deployment, and scaling.
- **Intelligent Routing:** A dedicated **Routing & Orchestration Service** acts as the central intelligence, selecting the best-fit MCP agent for any given task.
- **Multimodal Support:** A **Multimodal Processing Service** handles all non-textual inputs, leveraging cloud APIs and a Vector Database for advanced document processing.
- **Modern Stack:** A polyglot stack utilizing **Python/FastAPI** (for AI/ML) and **Go** (for high-performance core services) deployed on **Kubernetes**.

2. Core Requirements and Architectural Mandates

The design is driven by the following prioritized requirements:

Priority	Requirement	Architectural Solution
Critical	**Horizontal Scalability**	Kubernetes Orchestration and Stateless Microservices.
Critical	**Maintainability**	Independent Services, Service Mesh (Istio), and Comprehensive Observability (Prometheus/Jaeger).

High	**MCP Registry**	Dedicated **Registry Service** with PostgreSQL and Secret Manager for secure credential storage.
High	**Intelligent Routing**	**Routing & Orchestration Service** for intent detection and agent selection.
High	**Multimodal Gateway**	**Multimodal Processing Service** for Voice (STT/TTS), Vision, and Document Ingestion.
Medium	**Dark Mode**	Next.js/Tailwind CSS frontend with native dark mode support.

3. Microservices Architecture Design

The architecture is composed of loosely coupled services communicating primarily via **gRPC** (synchronous) and **Kafka** (asynchronous).

3.1. Core Microservices

Service	Technology Focus	Primary Function
API Gateway	Go	Single entry point, authentication, rate limiting, WebSocket management.
Auth Service	Go	User identity and authorization management.
Registry Service	Go	CRUD operations for MCP metadata; health check integration.
Monitoring Service	Go	Active health checks on all registered MCP endpoints and internal services.
Routing & Orchestration	Python/FastAPI	Intent detection, context management, and

		selecting/calling the optimal MCP agent.
Multimodal Processing	Python/FastAPI	Handles STT, TTS, Document Ingestion (chunking, embedding), and Vector DB interaction.
Glazyr Connector	Python/FastAPI	Dedicated interface for requesting and translating real-time visual context from Glazyr.

3.2. Data Stores and Infrastructure

Component	Technology	Role
Orchestration	**Kubernetes**	Deployment, scaling, and management of all microservices.
Event Bus	**Apache Kafka**	Central nervous system for resilient, asynchronous communication.
Relational DB	**PostgreSQL**	Primary storage for user data and MCP metadata.
Vector DB	**Qdrant/Pinecone**	Storage for document embeddings (RAG).
Secret Management	**Vault/Cloud Manager**	Secure, isolated storage for MCP API keys and credentials.

4. Detailed Data Flow Example

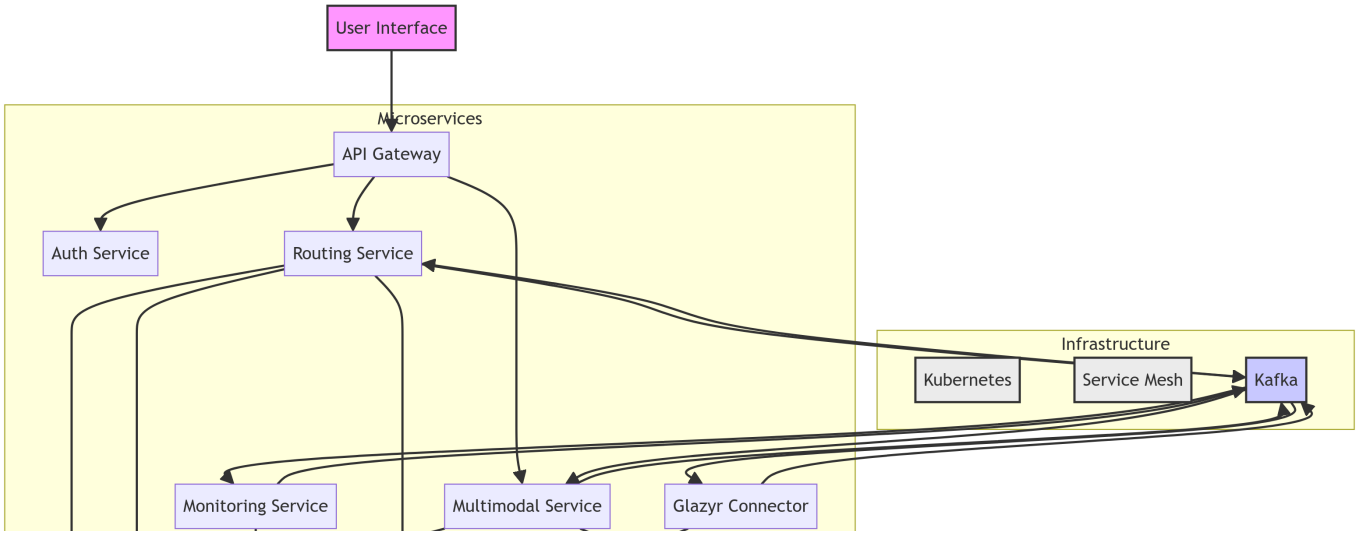
The use of the Event Bus (Kafka) ensures that complex, long-running tasks like multimodal processing do not block the user interface, improving perceived performance.

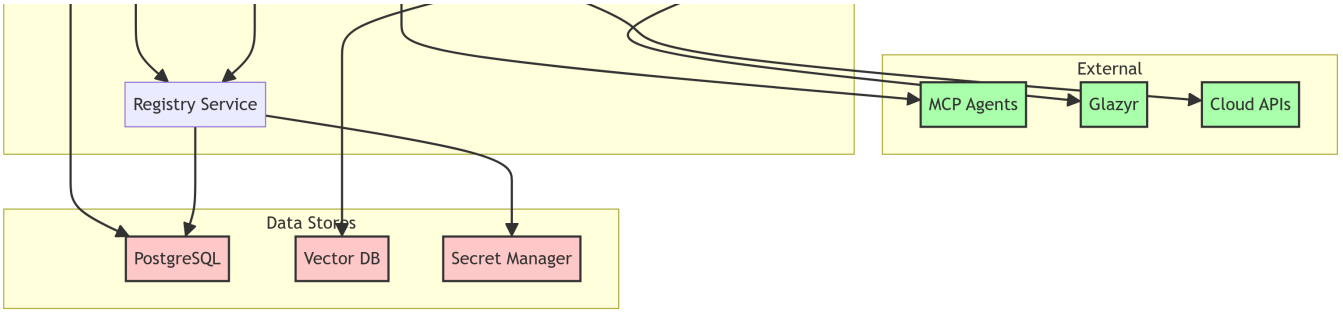
Step	Component	Action
1. Input	**UI -> API Gateway**	User input (e.g., voice stream) is sent.

2. Processing	**Multimodal Service**	Converts voice to text (STT); publishes `USER_QUERY_RECEIVED` event to Kafka.
3. Orchestration	**Routing Service**	Consumes event; determines intent (e.g., needs Glazyr context); publishes `GLAZYR_CONTEXT_REQUESTED` event.
4. Context Fetch	**Glazyr Connector**	Consumes event; fetches visual context from Glazyr; publishes `CONTEXT_READY` event.
5. Agent Call	**Routing Service**	Consumes `CONTEXT_READY`; calls selected **Registered MCP Agent** (via gRPC).
6. Output	**Routing Service**	Publishes `MCP_RESPONSE_READY` event.
7. Final Output	**Multimodal Service**	Consumes event; converts text to speech (TTS); sends final response to **UI** via WebSocket.

5. Architecture Diagram

The following diagram illustrates the component breakdown and communication paths of the new Microservices Architecture.





6. Conclusion

This Microservices Architecture provides a robust, scalable, and maintainable foundation for the MCP Registry and Management Platform. By adopting cloud-native patterns and a polyglot stack, the system is future-proofed against technical debt and is fully equipped to handle the complex demands of multimodal AI and central MCP management.