Self Project 2025-06-15

Autonomous Microservice Composition via LLM Agents in an MCP Control Plane

An agentic MCP control plane that converts natural-language intent into executable service DAGs using registry metadata, schema retrieval, orchestration, retries, and telemetry.

No cover image available.

Project Snapshot

DATE
2025-06-15

CATEGORY
Self Project

SHORT DESCRIPTION
An agentic MCP control plane that converts natural-language intent into executable service DAGs using registry metadata, schema retrieval, orchestration, retries, and telemetry.

TAGS / TECH STACK
MCP Agentic AI Orchestration Microservices

Project Links

Project Link Project Link GitHub Repository

Full Project Content

> A control plane for autonomous microservice composition that turns user intent into executable service graphs.

GitHub Repository: [anubhaparashar/Autonomous-Microservice-Composition-via-LLM-Agents-in-an-MCP-Control-Plane]

This project implements an **MCP control plane** for autonomous microservice composition. The repository centers on a single FastAPI app, `control_plane.py`, which combines a Redis-backed service registry, an OpenAI-powered DAG planner, PostgreSQL + `pgvector` metadata access, and an execution orchestrator built with HTTPX and NetworkX.

Vision

Modern service ecosystems are rich in APIs, but composing them reliably from natural-language intent is still hard. This project explores a control-plane design where an LLM planner transforms intent into a JSON DAG, and an orchestration layer executes that graph with retries, fallback handling, and service-to-service coordination.

Why This Matters

> **Note** The README describes the repository as the control-plane implementation for the **Autonomous Microservice Composition** paper, with a FastAPI app that handles planning, execution, and combined plan-and-execute flows.

> **Important** The control plane explicitly combines four pieces: **Redis** for service registry, **OpenAI GPT-4o-mini** for planning, **PostgreSQL with pgvector** for metadata/embeddings, and **HTTPX + NetworkX** for orchestration.

> **Tip** This architecture is useful because it separates **service knowledge**, **reasoning**, and **execution** into distinct layers that can evolve independently.

> **Warning** The public repo is compact and implementation-focused, so this post should be read as a technical walkthrough rather than a full benchmark or production-hardening guide.

> **Caution** Any real deployment would need additional controls around authentication, validation, timeout strategy, observability, and execution isolation.

What the Control Plane Does

```
User Intent
    ↓
LLM-Based DAG Planner
    ↓
JSON Workflow Graph
    ↓
Execution Orchestrator
    ↓
Microservice Calls
    ↓
Results + Errors + Fallback Handling
```

The repo README and `control_plane.py` show this flow through three REST endpoints: `/plan`, `/execute`, and `/plan_and_execute`.

Project Attributes

Attribute	Description
`problem-statement`	Composing multiple microservices from natural-language intent is complex, brittle, and difficult to adapt in real time.
`primary-objective`	Build an MCP control plane that can plan and execute multi-step service workflows autonomously.
`core-technologies`	FastAPI, Redis, PostgreSQL, `pgvector`, OpenAI GPT-4o-mini, HTTPX, and NetworkX.
`planning-layer`	Uses an LLM to convert user intent plus service metadata into a JSON DAG.

Attribute	Description
`execution-layer`	Executes the graph with topological ordering, HTTP calls, and fallback behavior.
`service-knowledge`	Stores service metadata such as endpoint, schemas, cost profile, and fallback in Redis.
`adaptive-inputs`	The README mentions telemetry integration and schema embeddings for more context-aware planning.
`deployment-mode`	Python application served with Uvicorn through a FastAPI entry point.

Repository Structure

At the time of writing, the public repo is intentionally small and centered on the main application file.

```
Autonomous-Microservice-Composition-via-LLM-Agents-in-an-MCP-Control-Plane/
├── README.md
└── control_plane.py
```

Setup Summary

The README lists three required environment variables:

```
export REDIS_URL="redis://localhost:6379/0"
export POSTGRES_DSN="host=localhost dbname=metadata user=postgres password=secret"
export OPENAI_API_KEY="<your_openai_api_key>"
```

It also lists the main dependencies as:

```
pip install fastapi uvicorn redis httpx networkx psycopg2-binary pgvector openai
```

And the app is started with:

```
uvicorn control_plane:app --reload --host 0.0.0.0 --port 8000
```

Core Components

1. Service Registry

The README says the service registry stores metadata such as endpoint, input schema, output schema, cost profile, and fallback options in Redis.

```
{
  "name": "user-profile",
  "endpoint": "http://user-profile-service/api",
  "input_schema": {},
  "output_schema": {},
  "cost_profile": 0.005,
  "fallback": "http://user-profile-fallback/api"
}
```

This keeps the planner and orchestrator decoupled from hardcoded service definitions.

2. LLM-Based Planner

The planner uses OpenAI GPT-4o-mini to transform the user intent into a JSON DAG. In `control_plane.py`, the planner builds a prompt from the available services and user intent, calls `openai.ChatCompletion.create`, and parses the returned JSON into a `PlanResponse`.

```
class PlanRequest(BaseModel):
    intent: str

class PlanResponse(BaseModel):
    graph: dict
```

3. PostgreSQL + pgvector Metadata

The code initializes a PostgreSQL connection and registers vectors through `pgvector.psycopg2.register_vector`. A helper method fetches service schema embeddings from a table named `service_schemas`, showing how semantic metadata can be incorporated into planning.

```
def _fetch_embeddings_metadata(self):
    with self.conn.cursor() as cur:
        cur.execute("SELECT name, input_schema_vector FROM service_schemas;")
        return cur.fetchall()
```

4. Execution Orchestrator

The orchestrator builds a directed graph with NetworkX, then executes nodes in topological order. It assembles inputs from prior results and payload values, sends HTTP

requests with HTTPX, and attempts fallbacks when available.

```
for name in nx.topological_sort(G):
    node = G.nodes[name]
    service_url = node["endpoint"]
    inputs = {k: results.get(v, payload.get(v)) for k, v in node["inputs"].items()}
```

API Surface

The FastAPI app exposes three endpoints:

```
@app.post("/plan", response_model=PlanResponse)
def plan_intent(req: PlanRequest):
    return planner.plan(req.intent)

@app.post("/execute", response_model=ExecuteResponse)
async def run_graph(req: ExecuteRequest):
    return await orch.execute(req.graph, req.payload)

@app.post("/plan_and_execute", response_model=ExecuteResponse)
async def plan_and_run(req: PlanRequest):
    plan = planner.plan(req.intent)
    return await orch.execute(plan.graph, {})
```

These three endpoints create a clean split between planning-only, execution-only, and a combined end-to-end flow.

Why This Design Is Interesting

- It keeps **planning** and **execution** separate
- It uses a registry instead of hardcoding services
- It supports topological execution of dependency graphs
- It includes fallback behavior for failed services
- It leaves room for telemetry- and embedding-aware planning

This makes the repository a useful prototype for control-plane thinking around LLM-driven service composition.

Current Limitations

- The public repo is compact and does not yet include a broader project structure
- Security, auth, and multi-tenant concerns are not the visible focus of the starter

- There is no complete benchmark or evaluation harness in the repo root
- Retry policy is described in the README, but the visible code is still a concise reference implementation

---

What I Would Add Next

- Typed DAG schemas and stronger validation
- Structured retry policies with exponential backoff
- Authentication and service authorization
- Richer telemetry capture and observability dashboards
- Safer prompt construction and planner guardrails
- Execution sandboxes and rate limiting

These additions would make the control plane much more deployment-ready.

Key Innovation

> **Important** The central idea is combining **LLM-based workflow planning** with a **service-aware execution control plane**, rather than using the LLM only as a text generator.

Conclusion

This project is a strong prototype for **autonomous microservice composition**. Its value lies in how it unifies registry metadata, semantic context, DAG planning, and orchestrated execution into one control-plane design.

> From user intent to executable service graph — that is the core promise of this MCP control plane.