Project Documentation

Title: Distributed AI Agents for Traffic and Environmental Data Monitoring
Author: Ashutosh Pattanayak

📘 Table of Contents

1. Introduction

This project simulates a distributed system of AI agents that generate synthetic environmental and traffic data, register themselves with Consul for service discovery, and provide data monitoring through a central Flask web-based dashboard.

Detailed Installation & Setup Guide

1)Prerequisites

- Python 3.9+
- Docker & Docker Compose
- Git
- Consul

2) Clone the project

git clone <repository-url>

cd "CEI_platform\CEI_platform"

3) Install Docker & Docker Compose

Follow the official Docker documentation for your OS.

4) Start Consul

You can use Docker to run Consul:

docker run -d --name=consul -e CONSUL_BIND_INTERFACE=eth0 -p 8500:8500 consul

Or install locally and run:

consul agent -dev

Access Consul UI at http://localhost:8500.

For detailed installation guide follow these steps

Download official binary

1) **Go to the official Consul download page**
https://developer.hashicorp.com/consul/downloads

2) **Download the right zip file** for your OS (Windows, Linux, or macOS).

3) **Unzip the file**:

- **Linux/macOS**:

Windows:

Extract the `.zip` and place `consul.exe` in a folder in your PATH (e.g., `C:\Program Files\Consul\`), or add its location to your environment variables.

4) **Verify installation**:

```
Bash
consul version
```

You should see output like:

```
Consul v1.x.y (123abcde)
```

-

5) Build & run all agents + controller

From the root directory (with your docker-compose.yml):

docker-compose up -d --build

This spins up all agents and the controller in separate containers, each exposing their Flask API.

Each agent runs independently in its own container

- Example:
    - traffic_agent container → exposes http://localhost:5000
    - noiseagent container → exposes http://localhost:5002
    - humidityagent, co2_agent, temperatureagent → each in separate containers on unique ports.

The controller runs in a separate container

- This container typically exposes your dashboard web app (e.g., http://localhost:8000).
- It queries Consul to discover agents and pulls data from each agent's /data endpoints.

Containers communicate on a shared Docker network

- Your docker-compose.yml defines a common network, so the controller can talk to agents by their service names (e.g., traffic_agent:5000) even though they're in different containers.

Consul runs in its container (or locally)

- If you started Consul via Docker (docker run consul), it's also in a separate container.

🔎 **Why is this separation important?**

- Isolation → If one agent crashes or needs to be restarted, others aren't affected.
- Scalability → You can scale specific agents independently.
- Maintainability → Update or rebuild one agent without touching others.
- Flexibility → Agents can be deployed on different servers or cloud instances later.

6) Verify services

- Agents should appear in the Consul UI under **Services**.
- Health endpoints (e.g., curl http://localhost:5001/health) should return {"status": "healthy"}.

📦 **Docker Usage**

Why Docker?

<u>Consistent Environment for Each Agent</u>
Every agent (traffic_agent, noiseagent, etc.) has dependencies: Python, Flask, specific Python

packages, and your own code. Docker guarantees that whether you run it on your laptop, a server, or someone else's computer, the environment is identical. No "works on my machine" problems.

Isolation of Agents

Each agent runs in its own container with isolated file systems, processes, and network namespaces. If one agent crashes, it doesn't affect the others — and they don't pollute each other's environments.

Simplified Deployment

You can distribute your Docker images or docker-compose.yml file and deploy the entire system — agents + controller — on any machine with Docker installed. You don't need to manually install Python or requirements for each agent.

Scalability & Flexibility

Since each agent runs as a container, you can scale them independently. For example, if you wanted two traffic agents at different intersections, you could spin up two containers with different configurations.

Seamless Integration with Consul

Docker lets you deploy all agents in a single virtual network, making it easier for agents and Consul to communicate, even if you move the whole system to the cloud (AWS, GCP, etc.) later.

Better Resource Control & Monitoring

You can limit resources per container (CPU, RAM), see logs, and monitor agent health using Docker tools or orchestration platforms like Docker Swarm or Kubernetes.

Reproducibility for Development & Testing

Every time you build your Docker image, you get the same environment. This makes testing new agent features or debugging much faster and more predictable.

Build an individual agent image

bash

```
docker build -t traffic_agent ./traffic_agent/
```

Run the agent container manually

docker run -d -p 5001:5000 --name traffic_agent traffic_agent

Common commands

- See running containers: docker ps
- Stop: docker stop traffic_agent
- Logs: docker logs traffic_agent

docker-compose commands

- Build and start: docker-compose up -d --build
- View logs: docker-compose logs -f
- Stop: docker-compose down

2. Project Objectives

- Create modular agents generating synthetic data (traffic congestion, CO_2 levels, humidity, temperature, noise)
- Register agents dynamically with Consul
- Build a centralized dashboard to visualize agents' health, intelligence, requirements, and data
- Enable data export for analysis (JSON/CSV)

3. System Architecture

Architecture Summary:

- Agents (traffic_agent, noiseagent, humidity_agent, temperatureagent) run as separate Flask services inside Docker containers.
- Each agent registers itself with Consul on startup, and can be viewed on localhost:8500
- The controller (dashboard) queries Consul for registered agents and fetches data or health status, and the dashboard of the Flask app can be viewed on localhost:8000

4. Agent Details

Each agent:

- Runs a Flask app on a unique port.
- Exposes endpoints:
    - /health — returns JSON health status
    - /intelligence — capabilities of agents are provided like the average values of co2 emissions, average traffic congestion, average noise levels, average humidity levels and average temperature
    - /requirements – displays data that is required by the user such as for traffic_agent the requirement displayed is average_vehicle_count, and similarly, with other agents

- /data — latest synthetic reading
- /data/history – can see the history of data displayed in a Json file
- /data/export/json and /data/export/csv — historical data export

- Registers with Consul to announce its existence.

5. Data Flow & Endpoints

- Agents register with Consul via `/v1/agent/service/register`.
- Controller queries Consul for healthy agents.
- Controller calls `/health`, `/intelligence`, `/data` on agents' port such as port 5001 for co2_agent similarly with others
- Data visualized in the dashboard and available for export.

Example /health response:(Let's say for traffic_agent can be viewed on localhost:5000/health)
And this is what can be expected as output if agents are live and running{"status": "healthy"}

Example /data response:(Similarly, this is also for traffic_agent and can be viewed on localhost:5000/data since the port it is registered on is 5000)


```
{
 "congestion_status": "Moderate Congestion",
 "timestamp": "2025-07-01T19:24:42.697955",
 "vehicle_count": 61
}
```


Example /intelligence: For traffic_agent we have it on localhost:5000/intelligence


```
{
 "average_vehicle_count": 0,
 "data_points_analyzed": 1,
 "max_vehicle_count": 0,
 "min_vehicle_count": 0,
 "most_common_congestion_status": "Low Congestion",
 "timestamp": "2025-07-03T06:46:04.305344"
}
```


6. Code Explanation

Agent Registration with Consul:

```
def register_with_controller():
  try:
    response = requests.post(CONTROLLER_URL, json=metadata)
    if response.status_code == 200:
```

```
            metadata["uuid"] = response.json().get("uuid")
            print(f"[INFO] UUID received from controller: {metadata['uuid']}")
            save_metadata()
        else:
            print(f"[ERROR] Failed to register: {response.text}")
    except Exception as e:
        print(f"[ERROR] Controller registration failed: {e}")

def register_with_consul():
    try:
        agent_ip = socket.gethostbyname(socket.gethostname())  # Get container's IP

        service = {
            "ID": metadata["uuid"],
            "Name": metadata["agent_name"],
            "Address": agent_ip,
            "Port": PORT,
            "Meta": {
                "sensor_type": metadata["sensor_type"],
                "location": metadata["location"],
                "unit": metadata["unit"],
                "frequency": metadata["frequency"]
            },
            "Check": {
                "HTTP": f"http://{agent_ip}:{5001}/health",  # Correct health check URL
                "Interval": "10s"
            }
        }

        conn = http.client.HTTPConnection("consul", 8500)
        conn.request(
```

```
        "PUT",
        "/v1/agent/service/register",
        body=json.dumps(service),
        headers={"Content-Type": "application/json"}
    )
    res = conn.getresponse()
    print(f"[INFO] Registered with Consul. Status: {res.status} {res.reason}")
    conn.close()

except Exception as e:
    print(f"[ERROR] Failed to register with Consul: {e}")
```

register_with_controller()

Purpose:
Registers the agent with your **central controller** (your dashboard's Flask app) so it can recognize the agent, assign it a UUID, and store its metadata.

How it works:
1) Sends an HTTP POST request to CONTROLLER_URL with the agent's metadata as JSON.
2) Checks if the response status code is 200 (success):

- If successful, it extracts the UUID from the response, saves it into metadata["uuid"], prints confirmation, and saves the updated metadata locally by calling save_metadata().
- If not successful, it prints an error with the response text.
- If any exception occurs (e.g., controller unreachable, network error), it prints a descriptive error message.

What it enables:

- Ensures the controller knows all active agents and assigns them unique UUIDs for tracking.
- Stores important agent info like name, type, location, and more in the controller's database or memory.

---

register_with_consul()

Purpose:
Registers the agent with **Consul**, a service registry, so the controller and other services can discover it dynamically.

How it works:

1) Uses socket.gethostbyname(socket.gethostname()) to get the **container's actual IP address** — crucial in Docker because localhost inside a container doesn't mean the host machine; it points to the container itself. This ensures the health check URL points to the real IP.

2) Builds the service dictionary:

- "ID": the unique UUID from metadata.
- "Name": the agent's name.
- "Address": the container's IP address obtained earlier.
- "Port": the agent's Flask port.
- "Meta": dictionary containing descriptive information about the agent:
    - sensor_type, location, unit, frequency.
- "Check": Consul health check configuration:
    - "HTTP": URL Consul should poll every 10s to verify the agent is healthy (points to /health).
    - "Interval": every 10 seconds.

3) Opens an HTTP connection to Consul running at consul:8500 (the Consul container's standard hostname in Docker).

4) Sends an HTTP PUT request to /v1/agent/service/register with the service JSON as the request body.

5) Prints registration status code and reason to confirm successful registration.

6) Closes the HTTP connection.

7) If any exception occurs, prints a descriptive error.

Why these functions matter together

- **register_with_controller()** tells the **controller** that an agent exists, giving it a UUID and registering metadata for central tracking.
- **register_with_consul()** tells **Consul** that the agent is alive, where it can be reached, and how it should be health-checked, enabling dynamic discovery and load balancing.

7. Deployment Guide

Prerequisites:

- Docker & docker-compose installed

Run All Agents and Dashboard:

docker-compose up -d --build

Access Dashboard:

- Central dashboard UI: http://localhost:8000
- Consul UI: http://localhost:8500

8. User Guide

Navigate to the dashboard to:

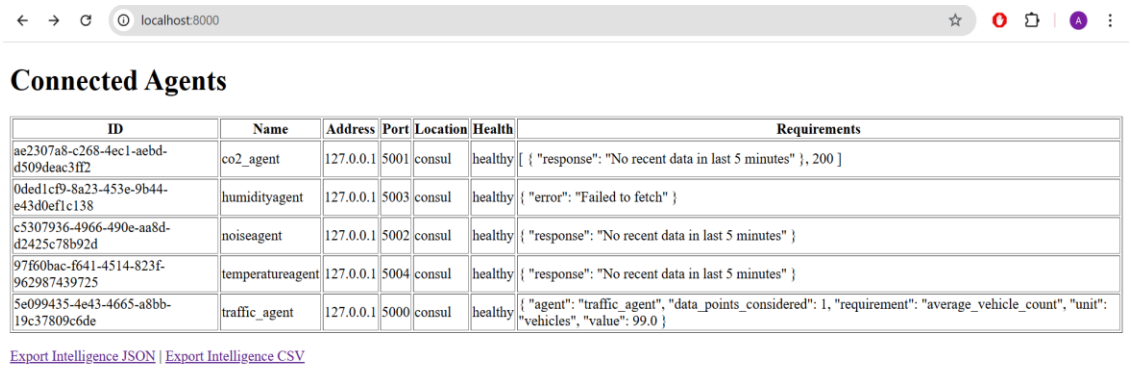
- View agents and their health status.
- See capabilities of /intelligence endpoint and data streams.
- Export data (JSON/CSV) for analysis.

Monitor agents in Consul UI:

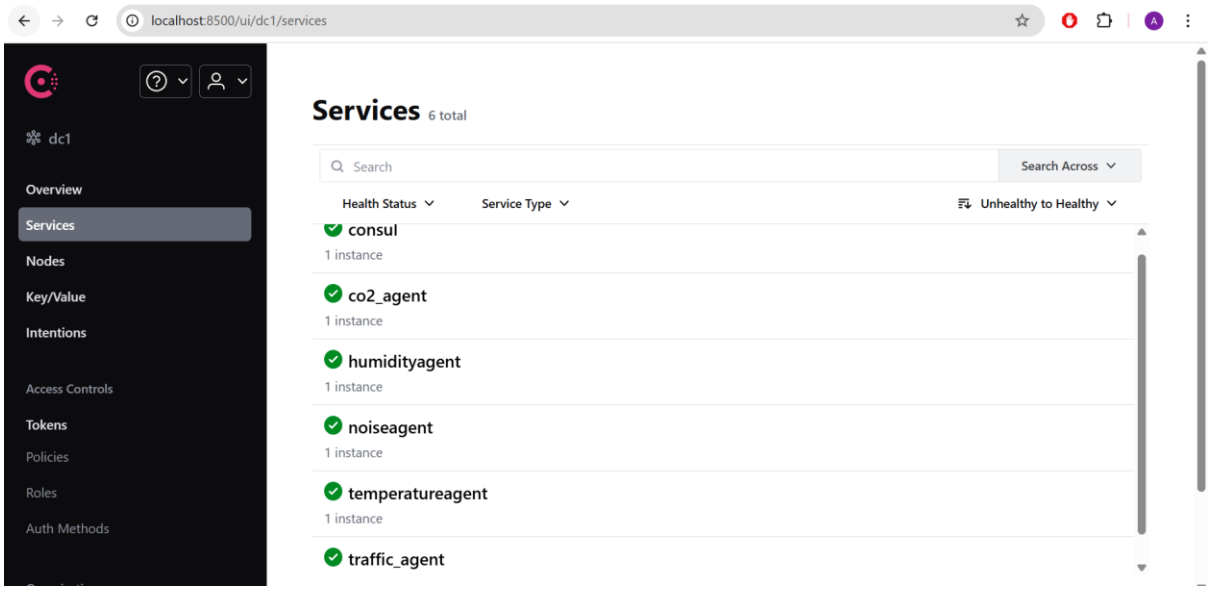
- Registered services appear with their health.

9. Results & Observations

- All agents registered successfully with Consul.
- Dashboard shows real-time health and data from each agent.
- Data export tested with accurate JSON/CSV outputs.
- Consul UI displays healthy services for each agent.



This is the Flask web-based dashboard showing agent health status, and requirements, and capabilities of agents at localhost:8000/requirements and localhost:8000/intelligence respectively

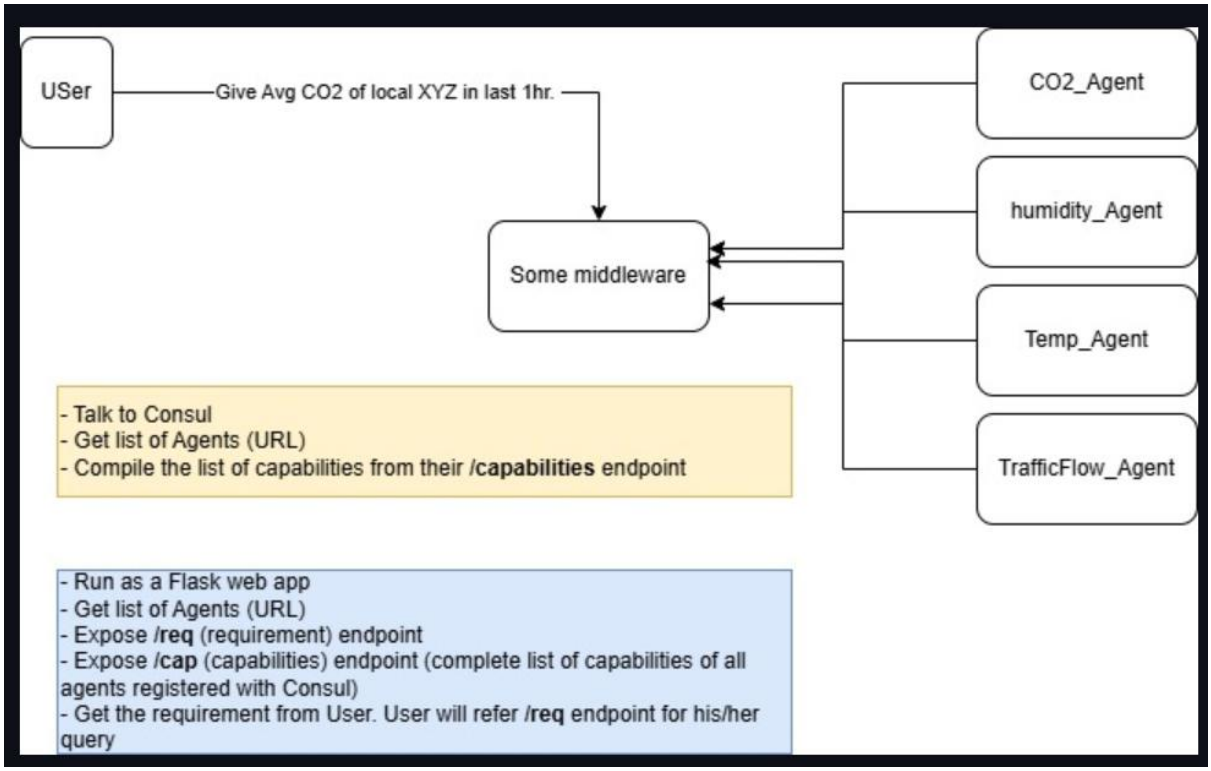


This is the consul interface that is showing the service discovery of all the agents

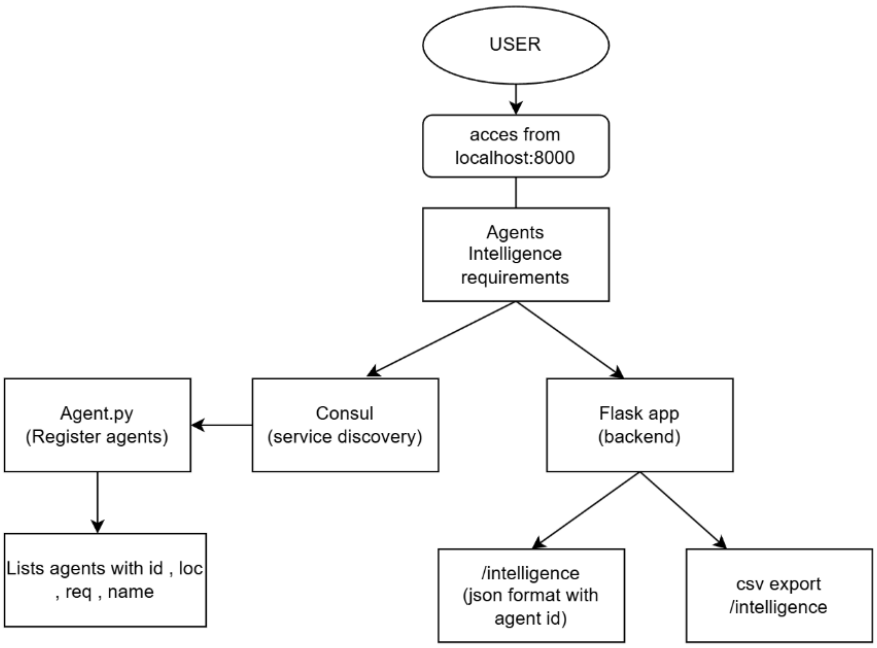
11. **Appendix**

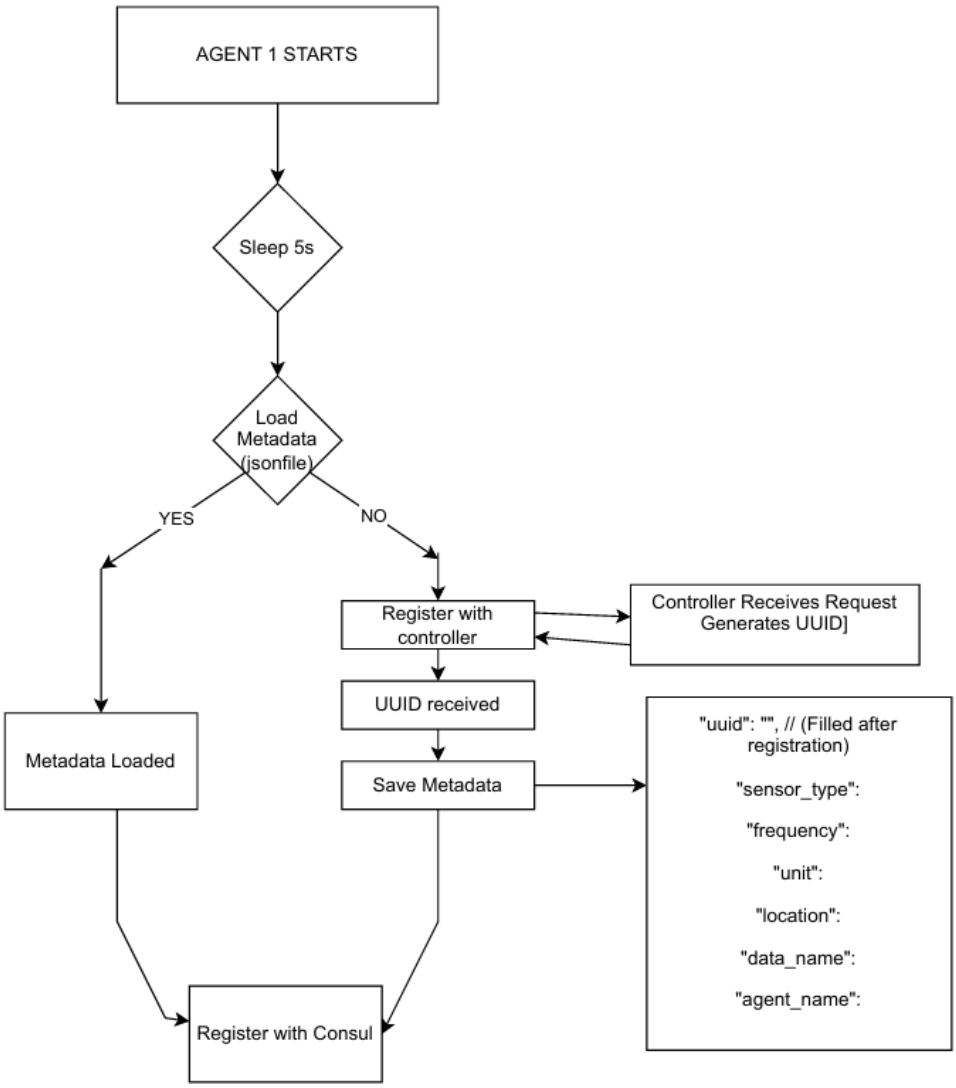
Platform for Clustered Edge Intelligence with discoverability and observability capabilities.

Overall Sy.



Flash web-based app working module:-





Above is the flow diagram of the registration process of an agent

11. Conclusion

This project demonstrates a modular distributed monitoring system using a microservices architecture with Consul for dynamic service discovery. It provides environmental and traffic metrics in a centralized dashboard, proving feasibility for smart city monitoring.

Key accomplishments include:

- Implementation of **multiple independent agents**, each responsible for generating specific synthetic sensor data (traffic, CO_2, humidity, temperature, noise).
- Dynamic **agent registration** and **health monitoring** using Consul, ensuring agents are discoverable and their availability is tracked.
- A **central Flask-based dashboard** that aggregates and visualizes agent data, enabling operators to monitor system health and export metrics for further analysis.

- Modular architecture that allows **individual agents to be scaled or updated independently**, improving maintainability and system resilience.
- Clear separation of responsibilities, aligning with best practices in microservices design.
- A **self-describing system**, where each agent exposes metadata, capabilities, intelligence, and requirements through standardized APIs.

Future improvements could include:

- Integration of **real sensor hardware** for live data collection.
- Enhanced **data analytics**, such as trend detection and anomaly alerts.
- Adding **authentication and authorization** mechanisms for secure agent–controller communication.
- Extending the dashboard with **real-time charts** and **historical data visualization**.
- Deploying the system to the cloud or edge devices for production-scale smart city environments.

This project demonstrates a solid foundation for **scalable, flexible, and maintainable distributed monitoring systems**, essential for next-generation urban management solutions.