A.S.A.P (Autonomous Search and Assistance Protocol) - Disaster Recovery using Multi Agent Systems

Version: 1.0
Date: 2025-03-27

Team: Phoenix(?)

Table of Contents

---

1. Introduction

The Autonomous Search and Assistance Protocol is designed to autonomously detect, assess, and respond to disaster events using a multi-agent framework. It integrates simulated sensor data from drones, ground robots, weather APIs, and more to generate actionable rescue tasks. The system is built with simulation in mind for rapid prototyping and will later be integrated with physical devices.

---

2. System Overview & Architecture

2.1 System Goals

- **Real-Time Data Ingestion:** Receive and process asynchronous sensor data from multiple sources.
- **Robust Decision Making:** Evaluate inputs using heuristics / LLM's / machine learning to decide on rescue actions.
- **Dynamic Task Allocation:** Dispatch tasks to specialized agents based on priority and available resources.
- **Feedback and Logging:** Update the system continuously using agent feedback.
- **Simulation-First Approach:** Create realistic, simulated data feeds to mirror real-world sensor inputs.

2.2 High-Level Architecture Diagram

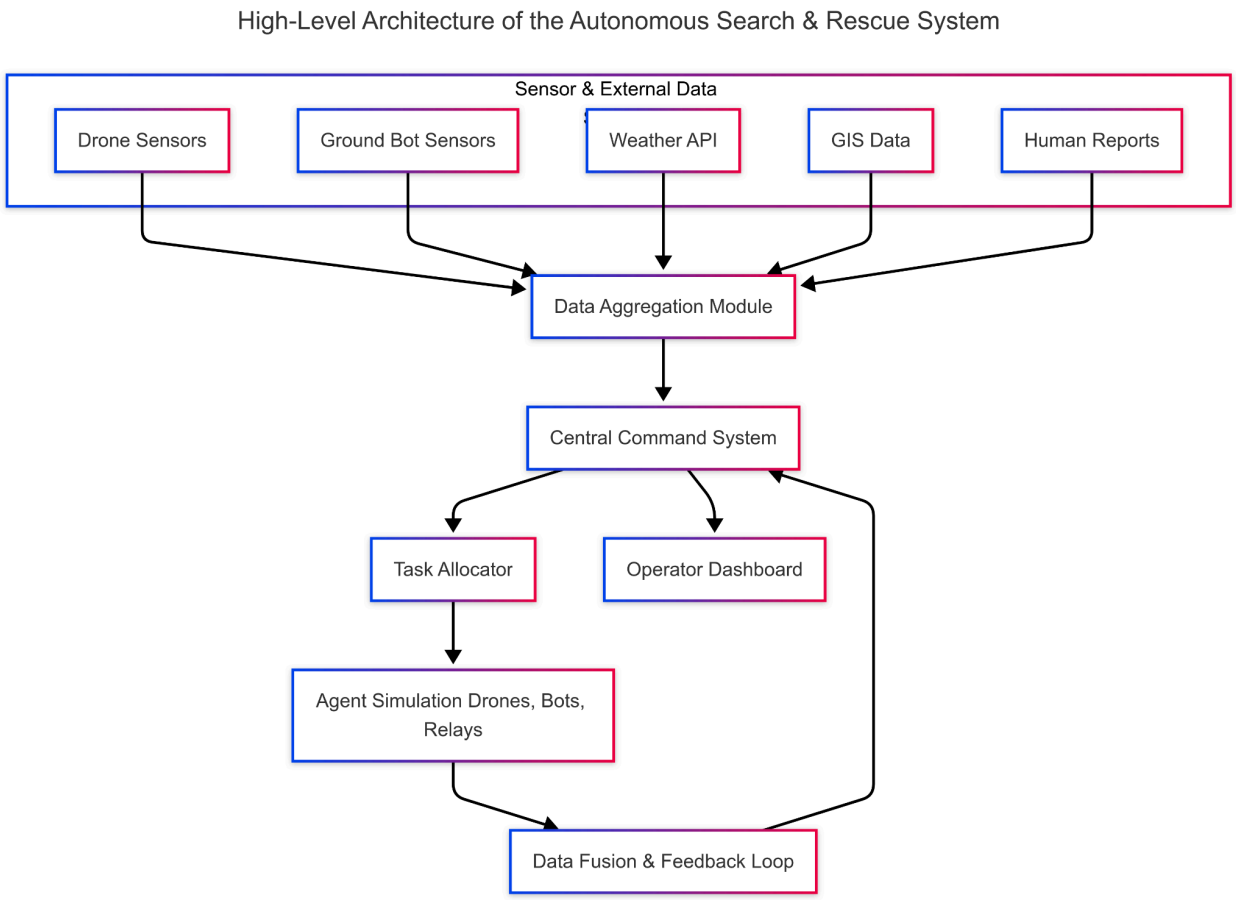


Figure 1: High-Level Architecture of the Autonomous Search & Rescue System

---

3. Component Breakdown

Data Aggregation Module

3.1 Overview

The Data Aggregation Module collects sensor data from multiple sources, normalizes it into a consistent JSON format, and forwards it to the Central Command System. It supports both push-based (real-time sensor reporting) and pull-based (periodic API polling) mechanisms.

3.2 Input Types & Examples for different Data sources

3.2.1 Drone Sensors

Frequency: Every 1–5 seconds

Raw Input:

```
 {
  "drone_id": "drone_1",
  "timestamp": 1712345678,
  "gps": {"lat": 47.6062, "lon": -122.3321},
  "heat_signature": true,
  "object_detected": "human",
  "battery": 78
}
```

Normalized Output:

```
{
  "source": "drone_1",
  "timestamp": 1712345678,
  "location": {"lat": 47.6062, "lon": -122.3321},
  "data_type": "survivor_detection",
  "value": true,
  "confidence": 0.92
}
```

3.2.2 Ground Bot Sensors

Frequency: Every 5–10 seconds

Raw Input:

```
 {
  "bot_id": "ground_bot_3",
  "timestamp": 1712345680,
  "gps": {"lat": 47.6070, "lon": -122.3340},
  "proximity_alert": false,
  "terrain_clear": true,
  "survivor_confirmed": false
}
```

Normalized Output:

```
 {
```

```
  "source": "ground_bot_3",
  "timestamp": 1712345680,
  "location": {"lat": 47.6070, "lon": -122.3340},
  "data_type": "ground_verification",
  "value": false,
  "confidence": 0.85
}
```

3.3.3 Weather API

Frequency: Every 5 minutes

Raw Input:

```
 {
  "station_id": "weather_api",
  "timestamp": 1712345700,
  "temperature": 18.5,
  "wind_speed": 5.2,
  "visibility": "clear"
}
```

Normalized Output:

```
 {
  "source": "weather_api",
  "timestamp": 1712345700,
  "location": {"lat": 47.6062, "lon": -122.3321},
  "data_type": "weather_conditions",
  "value": {"temperature": 18.5, "wind_speed": 5.2, "visibility": "clear"},
  "confidence": 1.0
}
```

3.3.4 GIS Data

Frequency: One-time load

Raw Input:

```
{
  "map_id": "city_map",
  "data": "GeoJSON string representing road blockages and zones"
```

```
}
```

Normalized Output:

```
 {
  "source": "GIS_data",
  "timestamp": 1712345600,
  "data_type": "static_map",
  "value": "GeoJSON data...",
  "confidence": 1.0
}
```

3.3.5 Human Reports

Frequency: On-demand

Raw Input:

```
 {
  "reporter": "user_123",
  "timestamp": 1712345710,
  "location": {"lat": 47.6050, "lon": -122.3300},
  "report": "I see smoke near the old warehouse."
}
```

Normalized Output:

```
 {
  "source": "human_report",
  "timestamp": 1712345710,
  "location": {"lat": 47.6050, "lon": -122.3300},
  "data_type": "emergency_report",
  "value": "smoke detected",
  "confidence": 0.80
}
```

3.3 Input/Output Mechanism & Frequency

- **Mechanism:**

- **Push:** Real-time sensors (drones, bots) push data via protocols like MQTT, WebSockets, or HTTP POST.

- **Pull:** External APIs (weather, GIS) are polled periodically.

- **Frequency:**

  - Drone data: every 1–5 seconds
  - Ground bot data: every 5–10 seconds
  - Weather API: every 5 minutes
  - GIS Data: loaded once
  - Human Reports: event-driven

Central Command System

3.4 Overview

The Central Command System is the decision-making hub. It receives aggregated data, evaluates inputs using heuristic / LLM / ML-based logic, and issues rescue tasks to agents. And informs First responders in human understandable language! Go LLM!.

3.5 Input & Output Examples

- **Input from Aggregator:**
  (Refer to the normalized examples above.)

Output (Task Dispatch Instruction):

```
 {
    "task_type": "rescue",
    "target_location": {"lat": 47.6062, "lon": -122.3321},
    "priority": "high",
    "assigned_agent": "ground_bot_2",
    "timestamp": 1712345685
}
```

3.6 Decision-Making Process

- **Data Reception:**
  Receives asynchronous data via queues.

Evaluation Logic:
Implements a heuristic function or make:

```
 def evaluate_reading(sensor_data):
    score = 0
    if sensor_data.get("data_type") == "survivor_detection" and
sensor_data.get("value"):
        score += 10
    if sensor
_data.get("confidence", 0) > 0.9:
        score += 5
    return score
```

- **Threshold-Based Action:**
  If the score exceeds a threshold (e.g., 15), a rescue task is generated.

- **Task Allocation:**
  Tasks are queued for dispatch to the appropriate agents.

Agent Task Allocation & Execution

3.7 Overview

This component simulates autonomous agents (drones, ground bots) that receive tasks, execute them, and provide feedback.

3.8 Task Structure Example

```
{
    "task_type": "rescue",
    "target_location": {"lat": 47.6062, "lon": -122.3321},
    "priority": "high",
    "assigned_agent": null,
    "timestamp": 1712345685
}
```

3.9 Agent Simulation Details

- **Drone Agent:**
  Simulates flight, image capture, and sensor updates.

- **Ground Bot:**
  Simulates terrain navigation and survivor verification.

For Hackathon, we will run agents as independent processes/threads (using `asyncio` or `multiprocessing`) and communicate via queues or messaging protocols.

3.10 Each Drone & Robot as an Independent Agent

- Every drone and robot runs as a **separate agent** (either on the same machine, different machines, or in the cloud).

- Agents **register with a Task Allocator** and receive tasks dynamically.

- **Health Monitoring** is implemented to detect failures and redistribute tasks

3.11 Core Components for Multi-Agent Coordination

1. **Task Assignment System**

    - Assigns search/rescue tasks based on **capability, location, and battery level**.

    - Uses a **priority queue** to determine which agent gets which task first.

    - Can use **Reinforcement Learning (RL) or heuristic-based scheduling**.

2. **Agent Health Monitoring System**

    - **Heartbeat Mechanism:** Each agent pings a task allocator every X seconds.

    - If an agent misses multiple heartbeats, it's marked **"down"** and tasks are reassigned.

3. **Dynamic Reallocation Mechanism**

    - If a drone/robot fails mid-task, the system **detects failure & reassigns the task to another agent**.

3.12 Workflow when an Agent Fails

Scenario: A drone fails mid-rescue mission

Step 1: Central Command detects the missing heartbeat.

Step 2: Ongoing tasks from the failed drone are marked **"REASSIGN NEEDED"**.

Step 3: A new drone (or ground robot if it's closer) picks up the task.

Step 4: The system logs failure & notifies human operators if necessary.

3.13 Alternative Decentralized approach:

- Drones/robots use a **peer-to-peer (P2P) system** to negotiate tasks.
- If a drone fails, **neighboring agents detect and adjust**.
- Uses **gRPC, WebSockets, or MQTT for direct agent communication**.

Data Fusion & Feedback Loop

3.14 Overview

This module fuses data from agents, re-evaluates task outcomes, and sends feedback to the Central Command System to allow dynamic task re-allocation.

3.15 Feedback Mechanism Example

```
{
    "agent_id": "ground_bot_2",
    "task_id": "1712345685",
    "status": "completed",
    "timestamp": 1712345705,
    "details": "Survivor confirmed; location verified."
}
```

Operator Dashboard

3.16 Overview

The Operator Dashboard provides real-time visualization and control, enabling human operators to monitor system status, view sensor data, and manually override decisions if necessary. We will use LLM to interpret the feed from the System Queues and update real time human readable messages. We also provide an option for human operators to provide feedback in real time, which will be provided to the Central command system.

3.17 Dashboard Features

- **Map View:**
  Display of agent locations and active tasks.
- **Status Logs:**
  A live feed of sensor inputs, task statuses, and system alerts, as human readable messages.
- **Manual Controls:**
  Buttons and input fields to reassign tasks or adjust thresholds.

---

4. Simulation Environment

4.1 Realistic Data Feeds

- Mimic sensor inputs from drones, bots, and APIs.
- Lets use a Synthetic Data Generator, or just create random inputs through Python.
- **[Optional]** Keep data formats similar to real systems so we can integrate with real physical devices in future.

4.2 Tools & Frameworks

- **Python & Asyncio:** For handling asynchronous data streams.
- **In-Memory Data Structures:** TBD `deque`/SQLite
- **Visualization:** Flask or any other Framework for the dashboard.
- **Messaging:** In-memory queues or some other Queue service.

4.3 Simulation Data Feed Example

A very simple Python module to Simulate sensor data. We can use this to build prototype, and lets build more complex simulations if we have time.

```
import asyncio
import random
import time
from collections import deque

# In-memory cache simulating a short-term data store
data_cache = deque(maxlen=100)

async def simulate_drone_data():
    while True:
        data = {
            "source": "drone_1",
            "timestamp": int(time.time()),
            "location": {"lat": round(random.uniform(47.6000, 47.6100), 4),
                         "lon": round(random.uniform(-122.3400, -122.3300),
4)},
            "data_type": "survivor_detection",
            "value": random.choice([True, False]),
            "confidence": round(random.uniform(0.7, 1.0), 2)
        }
        data_cache.append(data)
        print("Simulated Drone Data:", data)
        await asyncio.sleep(1)

async def simulate_weather_data():
    while True:
        data = {
            "source": "weather_api",
            "timestamp": int(time.time()),
            "location": {"lat": 47.6062, "lon": -122.3321},
            "data_type": "weather_conditions",
            "value": {"temperature": round(random.uniform(15, 25), 1),
                      "wind_speed": round(random.uniform(0, 10), 1),
                      "visibility": "clear"},
            "confidence": 1.0
        }
        data_cache.append(data)
        print("Simulated Weather Data:", data)
```

```
        await asyncio.sleep(300)  # 5-minute interval

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(simulate_drone_data())
    loop.create_task(simulate_weather_data())
    loop.run_forever()
```

---

5. Detailed Workflow Diagrams

5.1 Data Aggregation Workflow

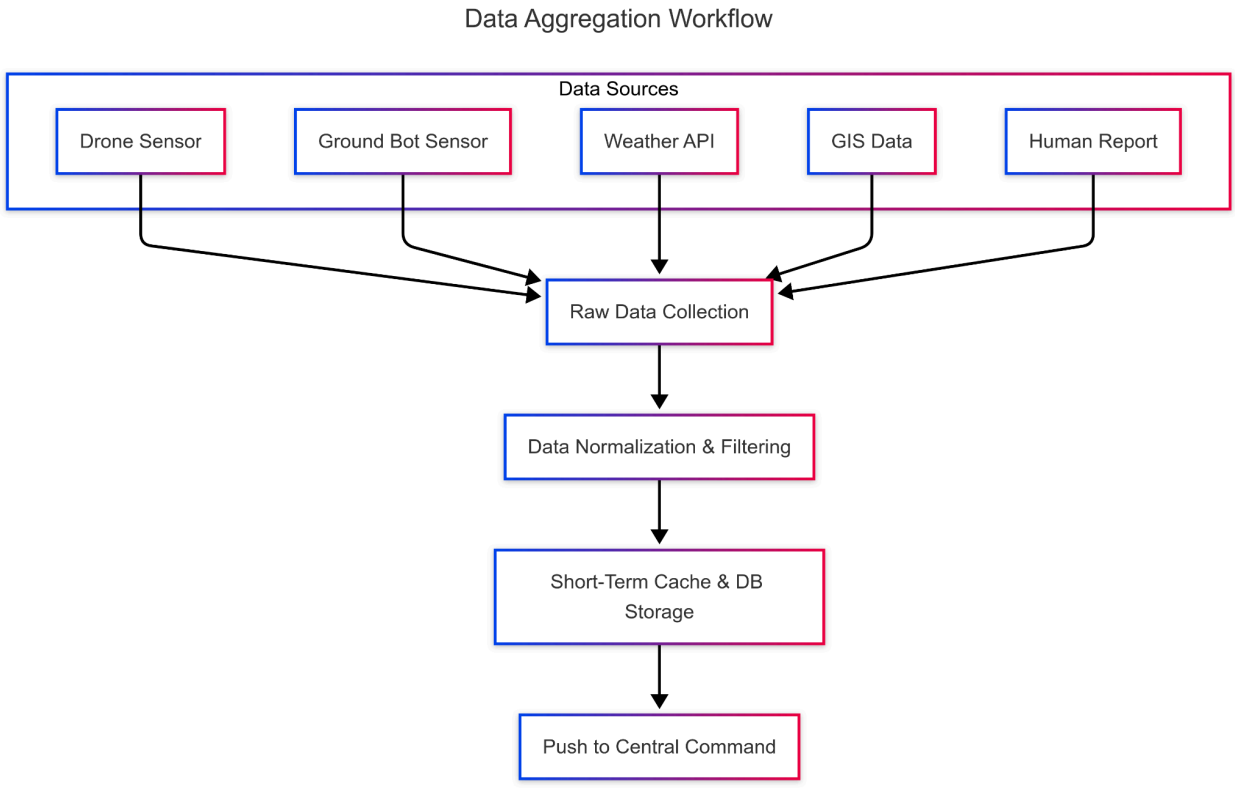


Figure 2: Data Aggregation Workflow

5.2 Central Command & Task Dispatch Workflow

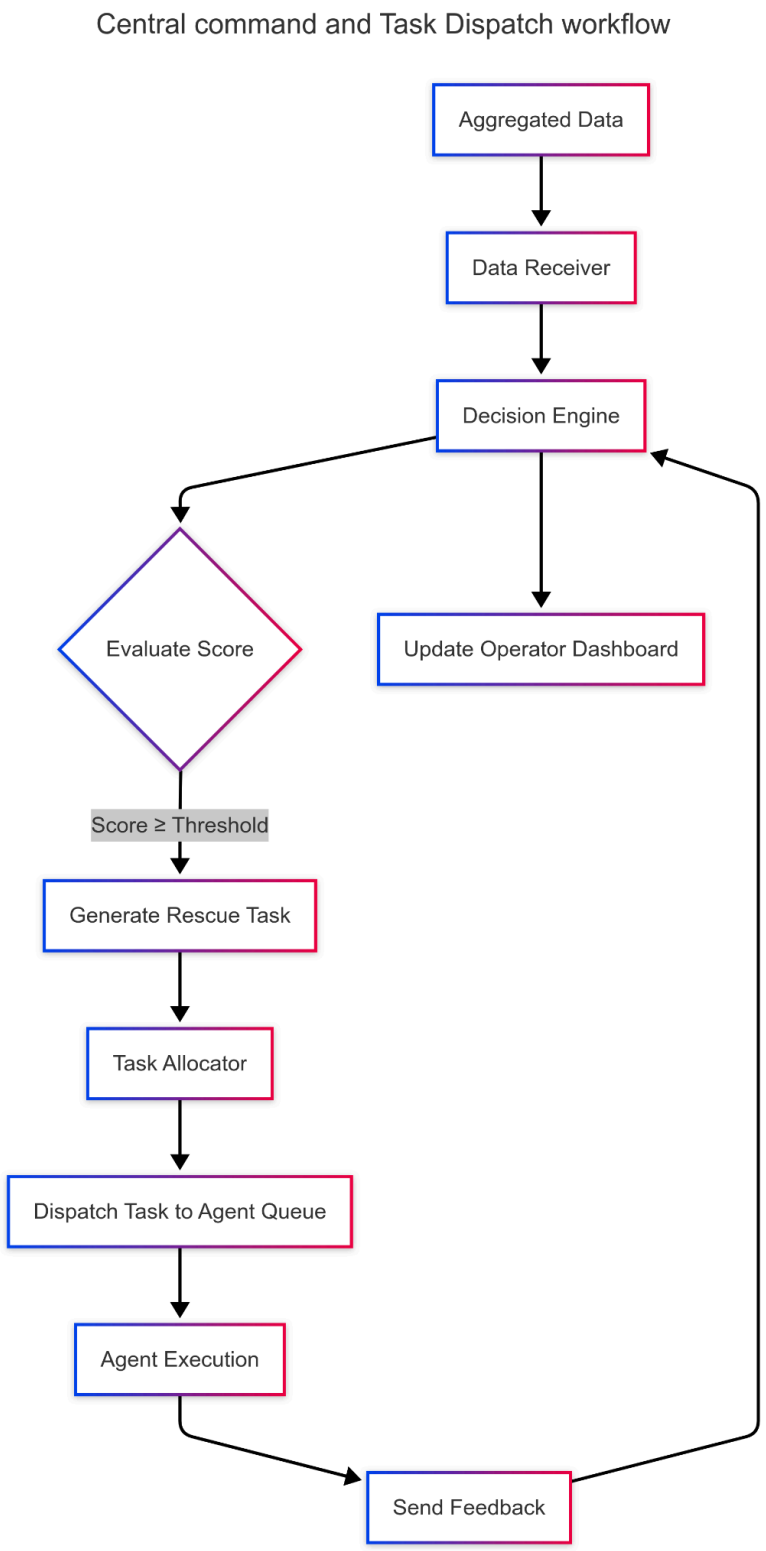
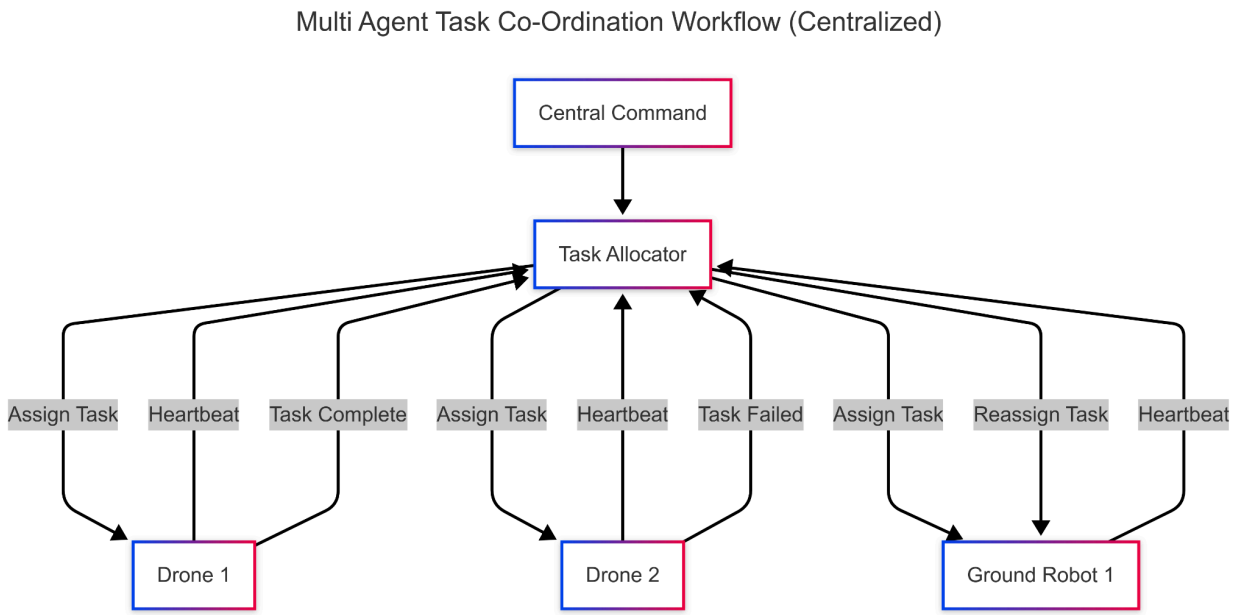


Figure 3: Central Command & Task Dispatch Workflow
The Evaluate Score section can be replaced with Foundational Models calls to make decisions, or LLM can be opted if we decide to go with it. We could also replace it with custom ML models.

5.3 Multi-Agent Task Allocation Workflow (Centralized)



https://www.researchgate.net/publication/310588656_Consensus_in_Multi-Agent_Systems

---

6. Implementation Details

6.1 Technology Stack

- **Language:** Python 3.9+
- **Libraries:**
    - Asynchronous: TBD(Maybe `asyncio`, `aiohttp`)
    - Messaging: `TBD`
    - Web Framework: TBD (Maybe `Flask`)

    - Data Storage(Is it needed for hackathon?): TBD, `deque` (cache), SQLite/PostgreSQL (archive)

    - Visualization: TBD

- **Simulation Tools:** Python-based simulation; later integration with ROS2 or AirSim.
  I looked up ROS2 - Its Robotics Simulator. We could simulate Drones, Drone Camera feed, etc. But it takes time to set it up.

6.2 Project Structure

```
asap_project/
├── agents/
│   ├── drone_agents/
│   │   ├── __init__.py
│   │   ├── drone_search.py
│   │   ├── drone_rescue.py
│   │   ├── drone_mapping.py
│   │   ├── config.py
│   │   └── utils.py
│   ├── ground_agents/
│   │   ├── __init__.py
│   │   ├── ground_search.py
│   │   ├── ground_rescue.py
│   │   ├── hazard_detection.py
│   │   ├── config.py
│   │   └── utils.py
│   ├── coordination_agent/
│   │   ├── __init__.py
│   │   ├── coordinator.py
│   │   ├── task_allocator.py
│   │   ├── failure_handler.py
│   │   ├── config.py
│   │   └── utils.py
├── data_aggregator/
│   ├── __init__.py
│   ├── aggregator.py
│   ├── config.py
│   ├── utils.py
├── central_command/
│   ├── __init__.py
│   ├── command_center.py
```

```
│    ├── mission_planner.py
│    ├── config.py
│    ├── utils.py
├── common/
│    ├── __init__.py
│    ├── messaging.py
│    ├── logger.py
│    ├── config.py
├── tests/
│    ├── test_drone_agent.py
│    ├── test_ground_agent.py
│    ├── test_aggregator.py
├── main.py
├── requirements.txt
├── README.md
```

Why Separate Packages for Agents?

Modularity – Each agent/module has a clear, isolated responsibility.
Reusability – Components can be reused in different projects or upgraded independently.
Scalability – Easier to add new agents or modify existing ones.
Fault Isolation – Failures in one agent won't impact others.
Parallel Development – We can work on different agents simultaneously.

Communication System options:

- **Message Broker (e.g., RabbitMQ, Kafka, Redis Pub/Sub)** – Agents publish and subscribe to events asynchronously.
- **REST APIs** – If we want synchronous interaction between services.
- **gRPC/WebSockets** – If real-time communication is required.

6.3 Starter Code Snippets

Central Command (central_command.py):

```
import asyncio
import json

# In-memory queues for sensor data and tasks
```

```
sensor_data_queue = asyncio.Queue()
task_queue = asyncio.Queue()

def evaluate_reading(sensor_data):
    score = 0
    if sensor_data.get("data_type") == "survivor_detection" and
sensor_data.get("value"):
        score += 10
    if sensor_data.get("confidence", 0) > 0.9:
        score += 5
    return score

async def decision_engine():
    while True:
        sensor_data = await sensor_data_queue.get()
        score = evaluate_reading(sensor_data)
        if score >= 15:
            task = {
                "task_type": "rescue",
                "target_location": sensor_data["location"],
                "priority": "high",
                "timestamp": sensor_data["timestamp"]
            }
            await task_queue.put(task)
        sensor_data_queue.task_done()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(decision_engine())
    loop.run_forever()
```

Agent Simulation (agents/drone_agent.py):

```
import asyncio
import random
import time

async def drone_agent(task_queue):
    while True:
        task = await task_queue.get()
```

```
        print(f"Drone received task: {task}")
        # Simulate task execution: flying, imaging, etc.
        await asyncio.sleep(random.uniform(1, 3))
        feedback = {
            "agent_id": "drone_1",
            "task_id": task.get("timestamp"),
            "status": "completed",
            "timestamp": int(time.time()),
            "details": "Survivor detected at location."
        }
        print("Drone feedback:", feedback)
        task_queue.task_done()

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    dq = asyncio.Queue()
    loop.create_task(drone_agent(dq))
    loop.run_forever()
```

---

7. Future Integration Considerations

- **Real Device Integration:**
  Replace simulated sensor feeds with actual hardware using ROS2 or other device APIs.
- **Distributed Deployment:**
  Scale agents across multiple machines using Docker, MQTT, or DDS.
- **Advanced Decision Making:**
  Integrate reinforcement learning models to refine task prioritization.
- **Security & Robustness:**
  Secure data feeds, enforce authentication, and implement failover strategies for mission-critical operations.

Action item:

- Send comms to first responder, when survivors are confirmed
- Send comms to Hospitals ahead of time