Emergency Response: A Multi-Agent System

Sheena Maria Lang, Antonio Lobo Santos, Zachary Parent,
María del Carmen Ramírez Trujillo and Bruno Sánchez Gómez

January 10, 2025

Contents

1 Introduction

This report presents the final implementation and results of our multi-agent system (MAS) for emergency response coordination. Building upon our previous designs from Tasks 1 and 2, we have developed a complete, functional system that demonstrates the effectiveness of agent-based approaches in managing complex emergency scenarios.

The system is implemented using CrewAI, a framework that enables the creation and coordination of specialized agent crews. Each crew is designed with specific responsibilities and operates through well-defined processes, ensuring efficient handling of emergency situations. The implementation includes:

- **Emergency Services Crew:** Handles initial emergency assessment and coordination
- **Firefighter Agent Crew:** Manages firefighting resources and operations
- **Medical Services Crew:** Coordinates medical response and hospital resources
- **Public Communication Crew:** Manages public information and communication

Report Structure:

- Section 2 details the design and implementation of each crew, including their process definitions and data models
- Section 3 explains the interaction mechanisms between crews and the overall system flow
- Section 4 presents the results of system testing and validation
- Section 5 concludes with insights and potential future improvements

The implementation builds upon our previous design while introducing several refinements based on practical considerations and testing results. These modifications are documented and justified throughout the report. The complete source code, along with setup instructions and required input files, is provided in the accompanying project repository.

2 Crew Design and Implementation

2.1 General Design Principles

For each crew in our system, a corresponding Python file is used to instantiate the configuration. These configurations are structured based on CrewAI's YAML schema recommendations for crews[1], tasks[2], and agents[3].

These files were generated using CrewAI CLI, a universal tool for creating configurations. The following command demonstrates how to initialize a new crew configuration:
`crewai create crew my_new_crew`[4].

The Python file is structured to include the following key elements:

- **Imports**: Required modules, including CrewAI components such as 'Agent', 'Task', 'Crew', and 'Process'.
- **Tool Instantiation**: Creation of tools for task-specific functionalities.
- **Agent Configuration**: Agents are defined with specific properties, including their roles, goals, delegation capabilities, verbosity, and parameters for interacting with the language model (e.g., temperature settings).
- **Task Configuration**: Tasks are defined with descriptions, expected outputs, dependencies, and execution modes. These configurations ensure tasks are properly structured and validated. Last task of each crew includes `output_pydantic` to ensure that the dictionaries returned to the crew are consistent.
- **Schema Augmentation**: Pydantic schemas can be added to the expected output of tasks using utility functions such as `add_schema_to_task_config`. This function modifies the task configuration by appending the schema JSON to the expected output property, ensuring that the LLM can take it into account.
- **Crew Composition**: The crew integrates agents and tasks into a defined process, executed sequentially, to accomplish its objectives.

The YAML configurations for agents and tasks specify several general properties:

[1] `https://docs.crewai.com/concepts/crews#yaml-configuration-recommended`
[2] `https://docs.crewai.com/concepts/tasks#yaml-configuration-recommended`
[3] `https://docs.crewai.com/concepts/agents#yaml-configuration-recommended`
[4] `https://docs.crewai.com/concepts/cli#1-create`

Agent Configuration: Agents are defined using YAML properties to specify:

- **Role:** The role the agent plays within the crew.
- **Goal:** The overarching objective or mission assigned to the agent.
- **Delegation Capabilities:** Whether the agent is allowed to delegate tasks.
- **Language Model Parameters:** Specific settings such as the model used and the temperature to control the randomness of the output.

Task Configuration: Tasks are defined in YAML with properties that include:

- **Description:** A clear explanation of the purpose and workflow of the task.
- **Expected Output:** The structure and format of the task output, often validated against a schema.
- **Dependencies:** Other tasks that provide context for the task.
- **Execution Mode:** Specifies whether the task is executed synchronously or asynchronously (e.g., 'async_execution: true').

This modular and schema-driven approach ensures flexibility, reusability, and validation throughout the configuration process.

A design overview of the system.

2.2 Tools

In this section, we describe the various tools developed for the Emergency Planner system. These tools are designed to facilitate different aspects of emergency management, including calculating route distances and managing database entries related to hospitals and incidents. Each tool is implemented with specific functionalities to address different requirements in emergency scenarios.

2.2.1 Route Distance Tool

Purpose The Route Distance Tool calculates the driving route distance between an origin and a destination based on their coordinates. This is essential for determining the quickest routes for emergency response teams.

Implementation

- **Input:** The tool requires the x and y coordinates of both the origin and destination locations.
- **Execution:** The city map graph is loaded from a GraphML file, and the shortest path is calculated using the travel time as the weight. The total distance is then computed.
- **Output:** The tool returns the total driving distance in kilometers.

The Route Distance Tool is a critical component to bring the Emergency Planner system closer to the real world. It gives the agents access to accurate geographical information, enabling them to make informed decisions about resource allocation and response times.

2.2.2 Database Management Tools

Purpose The Database Management Tools include the Hospital Reader, Hospital Updater, and Incident Retrieval tools. These tools manage and update the database entries related to hospitals and incidents, ensuring that the information is current and accurate.

Implementation

- **Hospital Reader Tool:**
  - **Input:** No input parameters are required for this tool.
  - **Execution:** The tool connects to the SQLite database and executes a query to fetch all hospital records.
  - **Output:** The tool returns a list of hospitals, including their IDs, locations, and available resources.
- **Hospital Updater Tool:**

  - **Input:** The tool requires the hospital ID, number of beds reserved, number of ambulances dispatched, and number of paramedics deployed.
  - **Execution:** The tool connects to the SQLite database and executes an update query to modify the hospital's available resources.
  - **Output:** The tool returns a confirmation of the update operation.

- **Incident Retrieval Tool:**
  - **Input:** The tool requires the x and y coordinates of the location, fire severity, fire type, and a summary of the new incident.
  - **Execution:** The tool connects to the SQLite database, retrieves related incidents based on proximity, fire severity, and fire type, and inserts the new incident into the database.
  - **Output:** The tool returns a list of related incidents.

The Database Management Tools are essential for maintaining up-to-date and accurate information on hospitals and incidents. By efficiently managing and updating the database, these tools ensure that the data is consistent throughout crews and runs, thus helping mitigate the hallucinative nature of the LLM-based agents.

2.3 Emergency Services Crew

2.3.1 Design

Purpose The Emergency Services Crew is tasked with the critical responsibility of assessing emergency situations and ensuring the appropriate response teams are dispatched effectively. This foundational role involves processing incoming emergency calls, structure the received information and decide which crews should be notified.

Changes In the initial design, the Emergency Services Crew was intended to utilize two specialized tools:

- *Database Management:* A system for systematically entering and recording emergency details.
- *GPS Mapping:* Software for locating and verifying the caller's position, assessing the scene, and prioritizing the severity of the incident if necessary.

However, after evaluation, it was determined that the core purpose of the crew was receiving the already transcribed incoming calls and deciding which team should receive the information, so both task could be accomplished without these tools. Removing them simplified the workflow, reduced overhead, and maintained focus on the essential tasks of the crew.

Structure and Components The Emergency Services Crew leverages the `CrewBase` framework to orchestrate its agents and tasks in a sequential process, ensuring efficient and orderly operations. The key components of the design are:

- **Agents:**
  - `emergency_call_agent`: Processes incoming emergency calls using a predefined configuration, ensuring the accurate extraction of relevant details.
  - `notification_agent`: Decides which crews need to be notified about the emergency, based on the nature and severity of the situation. This ensures the appropriate response teams are activated in a timely manner.
- **Tasks:**
  - `receive_call`: Captures and processes the details of emergency calls using schemas derived from the `EmergencyDetails` model, ensuring structured and consistent data handling.
  - `notify_other_crews`: Evaluates the information detailed in the `EmergencyDetails` and decides if other crews are needed, utilizing the `CallAssessment` schema to ensure clarity and comprehensiveness in the communication.
- **Crew:** The crew is constructed using the `Crew` class, integrating the agents and tasks into a seamless workflow. This design ensures the efficient progression from receiving an emergency call to notifying the relevant response teams.

The modular design of the Emergency Services Crew enables it to integrate seamlessly with other components of the multi-agent system while maintaining flexibility for updates or extensions.

2.3.2 Implementation

Emergency Call Agent The `emergency_call_agent` uses the configuration specified in `self.agents_config["emergency_call_agent"]` to ensure proper handling of call data. It acts as the first step in the emergency response workflow, extracting and structuring information from the calls.

```
@agent
def emergency_call_agent(self) -> Agent:
    return Agent(config=self.agents_config["emergency_call_agent"])
```

Notification Agent The `notification_agent` uses the configuration provided in `self.agents_config["notification_agent"]`. This agent ensures that only the relevant teams are alerted, streamlining the response process and avoiding unnecessary notifications.

```
@agent
def notification_agent(self) -> Agent:
    return Agent(config=self.agents_config["notification_agent"])
```

Receive Call Task This task processes the incoming call details and structures them according to the schema provided by the `EmergencyDetails` model. This ensures data consistency and validity.

```
@task
def receive_call(self) -> Task:
    config = add_schema_to_task_config(
        self.tasks_config["receive_call"], EmergencyDetails.model_json_schema()
    )
    return Task(config=config)
```

Notify Other Crews Task This task uses the processed emergency details to notify other relevant crews. It structures its output using the `CallAssessment` schema, ensuring clarity in communication.

```
@task
def notify_other_crews(self) -> Task:
    config = add_schema_to_task_config(
        self.tasks_config["notify_other_crews"], CallAssessment.model_json_schema()
    )
    return Task(config=config, output_pydantic=CallAssessment)
```

Crew Construction The agents and tasks are combined into a sequential process using the `Crew` class. This design ensures an orderly workflow from receiving calls to notifying the appropriate teams, where the Emergency Services Crew acts as a critical component for initial emergency assessment and coordination across multiple teams within the system.

```
@crew
def crew(self) -> Crew:
    """Creates the Emergency Services Crew"""
    return Crew(agents=self.agents, tasks=self.tasks, process=Process.sequential)
```

2.4 Firefighter Agent Crew

2.4.1 Design

2.4.2 Implementation

2.5 Medical Services Crew

2.5.1 Design

2.5.2 Implementation

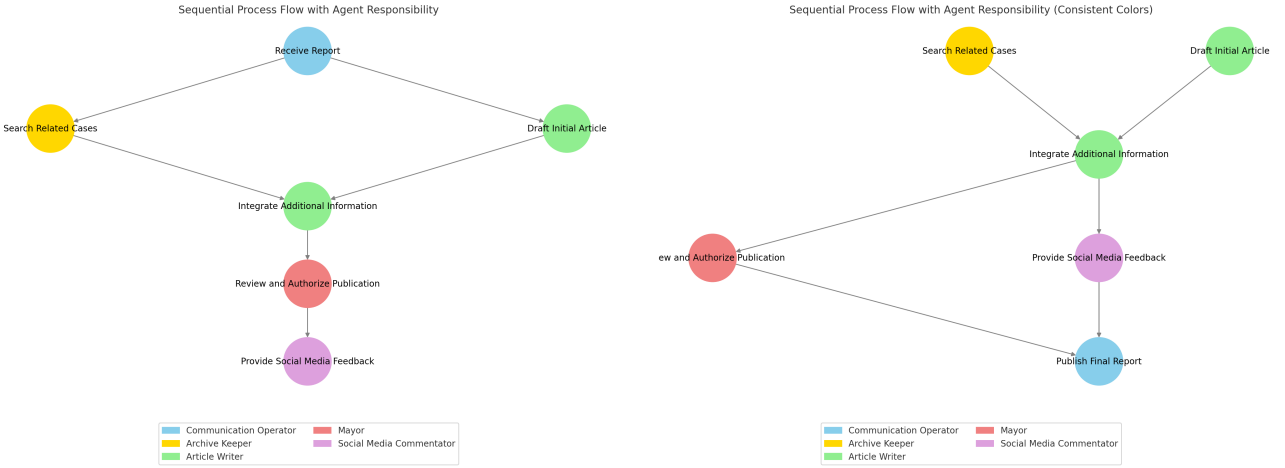
2.6 Public Communication Crew

2.6.1 Design

Purpose The Public Communication Crew is tasked with the essential responsibility of ensuring clear, accurate, and timely information dissemination during emergency situations. This includes relaying updates between

emergency teams and the public, maintaining historical records for decision-making, and crafting engaging and informative public communications.

Changes Initially, the tasks of the Public Communication Crew (PCC) were planned in a different order and with a distinct execution flow. A comparison of the initial and updated workflows is presented in **Figure** 1. In the updated process, the first task has been removed, and a new task has been added to consolidate all the information into the desired format. Additionally, the steps involving the Mayor's approval and social media commentary are now performed asynchronously.



(a) Initial process

(b) Updated process

Figure 1: Comparison of the Public Communication Crew's execution flow. (a) shows the initial process, while (b) illustrates the updated process with task removal, task addition, and asynchronous steps.

Structure and Components The Public Communication Crew leverages the `CrewBase` framework to orchestrate its agents and tasks, ensuring cohesive and efficient operations. The key components of the design are:

- **Agents:**
  - `communication_operator`: Relays information between the flow and the public communication team, acting as the primary joint to ensure clarity, correctness and structured format.
  - `archive_keeper`: Manages and retrieves historical data related to past incidents, supporting informed decision-making and comparative analysis.
  - `article_writer`: Drafts and refines articles for public dissemination, focusing on clear and engaging communication.
  - `mayor`: Reviews and approves public communications to ensure alignment with city policies and standards.
  - `social_media_commentator`: Provides lighthearted, constructive feedback on emergency responses, engaging the public while maintaining morale.
- **Tasks:**
  - `search_related_cases`: Analyzes historical records to identify similar incidents, highlighting trends and providing context for current events.
  - `draft_initial_article`: Crafts clear and concise initial drafts of public articles based on structured incident data.
  - `integrate_additional_information`: Refines drafts by integrating relevant historical or supplementary details, ensuring the article is ready for publication.
  - `review_and_authorize_publication`: Evaluates the refined draft for quality and policy alignment, providing final approval or feedback.
  - `provide_social_media_feedback`: Offers public-facing commentary that combines wit and constructive critique to enhance communication strategies.

- `publish_final_communication`: Consolidates all elements into a cohesive, structured and definitive public communication ready for dissemination.

- **Crew:** The crew is constructed using the `Crew` class, which integrates the agents and tasks into a seamless workflow. This design ensures effective progression from incident analysis and draft creation to final public dissemination and archival.

The modular design of the Public Communication Crew enables it to adapt to a wide range of scenarios, ensuring accurate, engaging, and timely communication with both internal teams and the public.

2.6.2 Implementation

3 Crew Interactions and Flow

3.1 Flow Design and Coordination

The Emergency Planner Flow is designed to handle emergency situations by coordinating multiple crews. The flow begins with the retrieval of a call transcript, followed by the processing of the call by emergency services. Based on the assessment, firefighters and medical services are dispatched in parallel. Public communication is managed after both teams report or during approval retries. Once the emergency is resolved, the flow concludes with the generation of a comprehensive emergency report, which includes summaries and timestamps from all participating crews.

3.1.1 State Management

The system maintains a centralized state using a Pydantic model [5], 'EmergencyPlannerState', which tracks all aspects of the emergency response. This includes the call transcript, assessments, and response reports. The state model ensures type-safe storage and accommodates partial updates.

3.2 Technical Implementation

The flow is orchestrated using CrewAI's decorators, which define the sequence and conditions for crew operations. Key flow control points include:

- `@start()`[6] for initiating the call transcript retrieval.
- `@listen()`[7] for establishing dependencies between operations, such as emergency services processing and the dispatch of firefighters and medical services.
- `@router()`[8] for handling conditional flow control, particularly for public communication approval.

3.2.1 Router Implementation

The router manages public communication approval, checking if the mayor has approved the communication. If not, it retries up to a maximum count. This ensures that public communication is handled appropriately and efficiently. This includes the use of `and_` and `or_` to combine multiple conditions. This is key for the retry mechanism for public communication approval.

Complex Logic for Public Communications `and_` [9] and `or_` [10] are used to combine multiple conditions. This is key for the retry mechanism for public communication approval.

```
@listen(or_(and_(firefighters, medical_services), "retry_public_communication"))
def public_communication(self):
    # ...
```

[5] https://docs.crewai.com/concepts/flows#structured-state-management
[6] https://docs.crewai.com/concepts/flows#start
[7] https://docs.crewai.com/concepts/flows#listen
[8] https://docs.crewai.com/concepts/flows#router
[9] https://docs.crewai.com/concepts/flows#conditional-logic-and
[10] https://docs.crewai.com/concepts/flows#conditional-logic-or

Router Logic for Public Communication Approval The router emits different messages based on the conditions, either triggering a retry or saving the full emergency report.

```
@router(public_communication)
def check_approval(self):
    logger.info("Checking approval")
    if self.state.public_communication_report.mayor_approved:
        return "save full emergency report"
    elif self.state.mayor_approval_retry_count >= MAX_MAYOR_APPROVAL_RETRY_COUNT:
        return "save full emergency report"
    self.state.mayor_approval_retry_count += 1
    return "retry_public_communication"
```

3.3 Justification of Design Choices

The design choices are justified by the need for a robust and flexible system that can handle complex emergency scenarios. The use of CrewAI's flow decorators allows for clear and maintainable code, while the parallel processing capabilities ensure timely responses from different crews.

4 Testing

4.1 Unit Tests

4.2 Integration Tests

The integration testing of the Emergency Planner system is conducted to ensure that all components, such as the crews and the flow, work together seamlessly to handle emergency scenarios. The testing process involves simulating real-world operations using test transcripts and verifying the generation of comprehensive emergency reports.

Flow Simulation The system is initiated using the 'crewai flow kickoff' command, which triggers the entire flow from start to finish. This command simulates the intake of an emergency call, processing by various crews, and the generation of a final report. The test transcripts, stored in 'call_transcripts.txt', provide detailed scenarios that the system must handle, including fires with specific hazards and the presence of injured individuals.

Component Processing and Report Generation During the integration tests, the system reads a selected transcript and processes it through the emergency services, firefighters, medical services, and public communication teams. Each crew generates a response report, which is compiled into a full emergency report saved to 'emergency_report.md'. This report includes the call transcript, response summaries, and public communication details, providing a comprehensive overview of the incident and response efforts.

Test Cases Two test cases were designed to test the possible scenarios that the system can handle. One with the presence of injured individuals and one without. We also were interested in observing if the retry system worked when the mayor did not immediately approve the public communication message.

1. **Test Case 1: Fire with Injured Individuals**

   Transcript A fire of electrical origin has broken out at coordinates (x: 41.71947, y: 2.84031). The fire is classified as high severity, posing significant danger to the area. Hazards present include gas cylinders and flammable chemicals, further escalating the risk. The fire is indoors, and there are 5 people currently trapped. Additionally, there are 2 injured individuals with minor and severe injuries respectively requiring immediate attention.

   Expected Output The medical team should be dispatched to the scene to attend to the injured individuals. The firefighters should be dispatched to the scene to extinguish the fire. The public communication team should craft a message to inform the public about the emergency.

2. **Test Case 2: Large-Scale Fire without Injured Individuals**

Transcript A gas fire has broken out at coordinates (x: 41.71892, y: 2.84127). The fire is classified as high severity at an abandoned warehouse facility. The hazard present is a storage area containing multiple industrial gas cylinders, which poses a significant risk of explosion. The fire is indoors, but the building has been confirmed empty and unused for several months. No individuals are trapped or at risk, and a security check of the premises has confirmed no squatters or unauthorized persons are present. The fire poses a risk of spreading to neighboring structures if not contained quickly.

Expected Output No medical assistance is required. The firefighters should be dispatched to the scene to extinguish the fire. The public communication team should craft a message to inform the public about the emergency.

Verification and Readiness The integration tests verify that each component of the system functions correctly and that the overall flow produces the expected outputs. By simulating real-world scenarios, the tests ensure that the Emergency Planner is robust, efficient, and ready for deployment in actual emergency situations.

4.3 Results

5 Conclusion

6 References