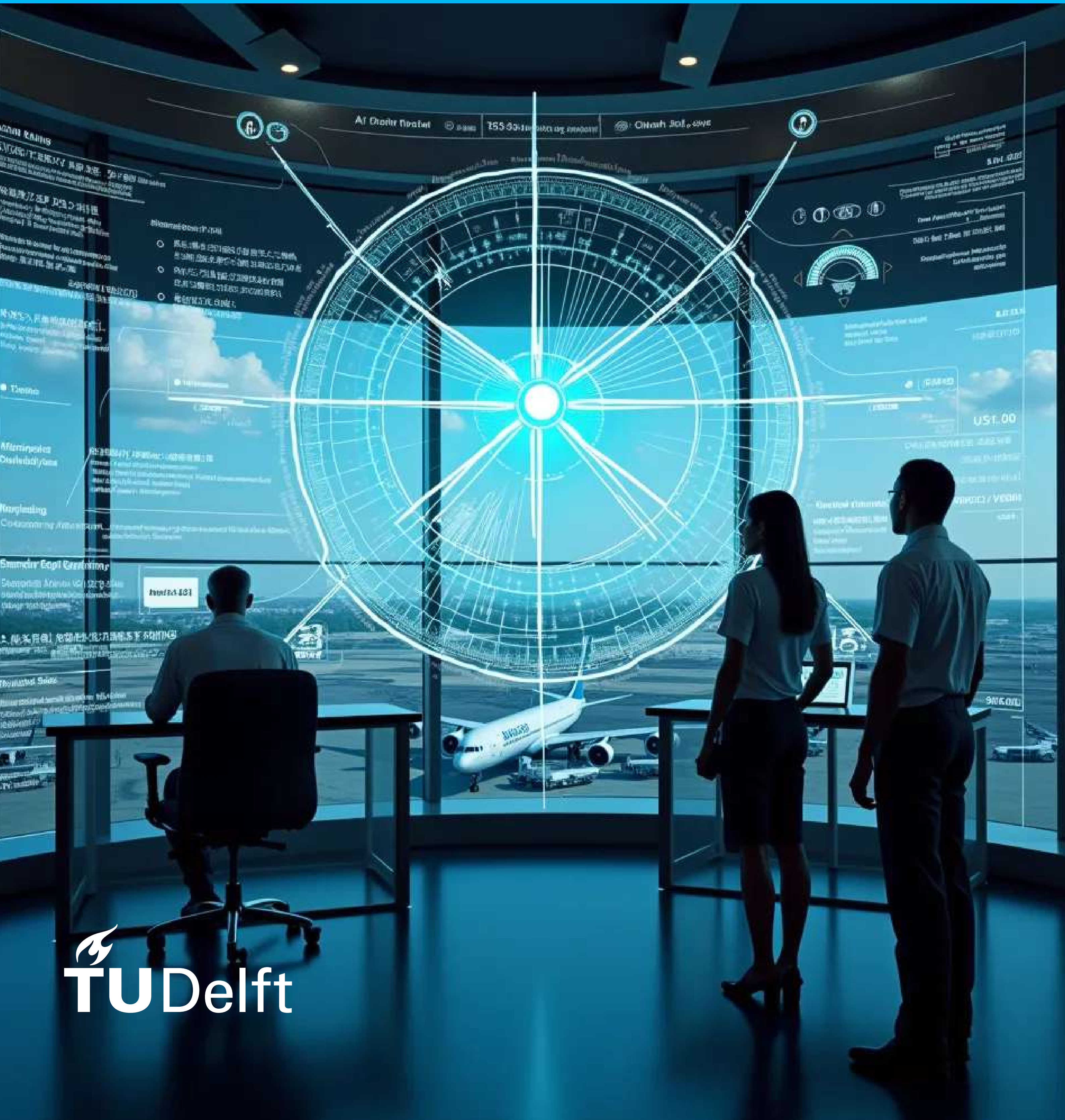
Automatic Control With Human-Like Reasoning

Exploring Language Model Embodied Air Traffic Agents

Justas Andriuskevicius



Automatic Control With Human-Like Reasoning

Exploring Language Model Embodied Air Traffic Agents

Thesis report 2024/11/05

by

Justas Andriuskevicius

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on November 5, 2024 at 08:30

Thesis committee:	
Chair:	Prof.dr.ir Jacco Hoekstra
Supervisors:	Dr. Junzi Sun
External examiner:	Dr. Alessandro Bombelli
Place:	Faculty of Aerospace Engineering, Delft
Project Duration:	December, 2023 - Novemeber, 2024
Student number:	5062632

Faculty of Aerospace Engineering · Delft University of Technology

Acknowledgements

I would like to express my gratitude to everyone who supported me throughout my thesis journey. First and foremost, I extend my thanks to my supervisor, Dr. Junzi Sun, for his encouragement and valuable contributions. His assistance with coding, troubleshooting, and offering fresh and innovative ideas, alongside his broader perspective on the project, was invaluable. This project's acceptance into the SESAR conference would not have been possible without his guidance and support.

I am also grateful to Professor Jacco Hoekstra for his insightful guidance, which significantly elevated the quality of my thesis.

I would also like to acknowledge visionplatform.ai, where I had the opportunity to work part-time. The projects I worked on aligned closely with my thesis, providing me with practical experience that strengthened my theoretical understanding.

A special thanks goes to my family, who have been my support throughout this process. I am also thankful to Xingyue for constant encouragement and for providing an additional API key, which allowed me to carry out more experiments. Finally, I would like to thank Dequan, Luke, Jiaxuan, and all my friends for their encouragement and support during this journey.

To all of you, thank you for your guidance, belief, and support.

Abstract

Recent developments in language models have created new opportunities in air traffic control studies. The current focus is primarily on text and language-based use cases. However,these language models may offer a higher potential impact in the air traffic control domain, thanks to their ability to interact with air traffic environments in an embodied agent form. They also provide a language-like reasoning capability to explain their decisions, which has been a significant roadblock for the implementation of automatic air traffic control. This thesis investigates the application of a language model-based agent with function-calling and learning capabilities to resolve air traffic conflicts without human intervention. The main components of this research are foundational large language models, tools that allow the agent to interact with the simulator, and a new concept, the experience library. An innovative part of this research, the experience library, is a vector database that stores synthesized knowledge that agents have learned from interactions with the simulations and language models. To evaluate the performance of our language model-based agent, both open-source and closed-source models were tested. The results of our study reveal significant differences in performance across various configurations of the language model-based agents. The best-performing configuration was able to solve almost all 120 but one imminent conflict scenario, including up to four aircraft at the same time. Most importantly, the agents are able to provide human-level text explanations on traffic situations and conflict resolution strategies.

Contents

Nomenclature

List of Abbreviations

ADS-B Automatic Dependent Surveillance-Broadcast

AI Artificial Intelligence

ANN Approximate Nearest Neighbors

ATC Air Traffic Control

ATCOs Air Traffic Controllers

ATFCM Air Traffic Flow and Capacity Management

ATFM Air Traffic Flow Management

ATM Air Traffic Management

BPE Byte-Pair Encoding

CAS Calibrated Airspeed

CPA Closest Point of Approach

DATIS Digital Automatic Terminal Information Service

DCPA Distance to Closest Point of Approach

FAA Federal Aviation Administration

GDP Ground Delay Program

GPT Generative Pre-trained Transformer

GS Ground Speed

HNSW Hierarchical Navigable Small World

HRV Heart Rate Variability

JSON JavaScript Object Notation

LLM Large Language Model

METAR Meteorological Aerodrome Report

ML Machine Learning

NASA National Aeronautics and Space Administration

nm Nautical Miles

PZ Protection Zone

RAG Retrieval-Augmented Generation

ReAct Reasoning and Acting framework

RL Reinforcement Learning

RLHF Reinforcement Learning with Human Feedback

SESAR Single European Sky ATM Research

SI Situations of Interest

TCPA Time to Closest Point of Approach

tLOS Time to Loss of Separation

U-Space Unmanned Aircraft Systems (UAS) Space

V/S Vertical Speed

VA Visual Analytics

XAI Explainable Artificial Intelligence

Part I

Scientific Article

Automatic Control With Human-Like Reasoning: Exploring Language Model Embodied Air Traffic Agents

Justas Andriuškevičius, Junzi Sun*
Faculty of Aerospace Engineering, Delft University of Technology
Delft, the Netherlands
*thesis supervisor

***Abstract*—Recent developments in language models have created new opportunities in air traffic control studies. The current focus is primarily on text and language-based use cases. However, these language models may offer a higher potential impact in the air traffic control domain, thanks to their ability to interact with air traffic environments in an embodied agent form. They also provide a language-like reasoning capability to explain their decisions, which has been a significant roadblock for the implementation of automatic air traffic control.**

This paper investigates the application of a language model-based agent with function-calling and learning capabilities to resolve air traffic conflicts without human intervention. The main components of this research are foundational large language models, tools that allow the agent to interact with the simulator, and a new concept, the experience library. An innovative part of this research, the experience library, is a vector database that stores synthesized knowledge that agents have learned from interactions with the simulations and language models.

To evaluate the performance of our language model-based agent, both open-source and closed-source models were tested. The results of our study reveal significant differences in performance across various configurations of the language model-based agents. The best-performing configuration was able to solve almost all 120 but one imminent conflict scenarios, including up to four aircraft at the same time. Most importantly, the agents are able to provide human-level text explanations on traffic situations and conflict resolution strategies.

***keywords* –** Air traffic control, self-learning agents, large language models, experience library, function-calling

I. Introduction

Air traffic management is a system that is critical for ensuring global airspace safety and operational efficiency. As air traffic volumes increase, so does the complexity of managing numerous flights and workloads for operators simultaneously [1], which raises the risk of incidents due to operational misunderstandings. These factors have historically contributed significantly to aviation accidents.

One of the main developments in air traffic management is the introduction of artificial intelligence in air traffic control to reduce the workload of air traffic controllers. The SESAR AISA project [2] was an early attempt to incorporate AI into air traffic management by creating a system for artificial situational awareness through the use of knowledge graphs and machine learning for traffic prediction. Another SESAR project, ARTIMATION [3], also aims at producing a transparent AI through visualization. Reinforcement learning (RL) has shown remarkable capabilities in solving complex, sequential decision-making tasks across various domains, including ATM. RL-based approaches are particularly well-suited for tasks like conflict detection and resolution, where they can optimize decisions based on a reward system. By learning through trial and error, RL can generate efficient solutions for handling air traffic conflicts. However, a key limitation of RL is its lack of transparency. While RL can provide optimal actions, it typically does so without explaining the reasoning behind its decisions, which can hinder trust in safety-critical environments such as ATM.

The SESAR TAPAS project [4] explored RL within ATM, aiming to offer actionable solutions to human operators. RL was used to develop AI-generated solutions for air traffic flow and capacity management (ATFCM) and conflict detection and resolution. While TAPAS advanced the field by integrating explainable AI (XAI) and visual analytics (VA) to make these solutions accessible to operators, the system still relied heavily on visual explanations. The RL models themselves did not provide any inherent reasoning, offering solutions without accompanying explanations. Operators had to rely on VA tools to interpret these solutions, but even then, the reasoning behind the decisions remained largely opaque. Especially when the system resolved multiple conflicts simultaneously, created challenges for operators who often preferred to address conflicts one by one.

Moreover, while RL can generate valid resolutions, its inability to convey human-like reasoning limits its integration into human-operated air traffic control systems. This is especially critical in real-time operations, where air traffic controllers need to understand the rationale behind decisions to ensure safety and maintain control.

Overall, the human-in-the-loop simulations revealed a gap between artificial and human situational awareness, highlighting room for improvement in AI's complex decision-making processes. This gap requires AI to offer more nuanced and human-like reasoning capabilities in air traffic management.

Since 2023, researchers have experimented with the integration of large language models (LLM) into air traffic management. Large language models are advanced AI systems capable of understanding and generating human-like text. Their proficiency in real-time decision-making has the potential to improve operational efficiency and automate labor-intensive tasks. Most of the data used to train state-of-the-art large language models, such as the latest Common Crawl dataset [5] comprises over 250 billion web pages, sources information from publicly accessible internet sites.

This extensive training equips large language models with a broad understanding of air traffic management standards, including guidelines from the International Civil Aviation Organisation, Federal Aviation Administration regulations, and other global and local aviation protocols. Consequently, large language models can interpret these contents effectively.

Several recent studies have explored use cases for aviation applications. For example, [6] employs language models to understand ground delay program text data. [7] fine-tunes the open-source language models to better understand the aviation context. A recent study [8] uses a language model for text classification and clustering based on air traffic flow management regulations and weather reports.

However, these use cases are primarily focused on natural language processing; they have not utilized the full potential of language models in managing air traffic operations nor looked into how AI can provide human-like reasoning.

A new concept, the language model embodied agent, Voyager [9], was introduced last year, which represents an innovative step in leveraging the language model's reasoning capability. It is designed for open interactions within the Minecraft game environment, where Voyager agents can explore the virtual world autonomously and, most importantly, acquire skills by experience and then apply skills.

In a similar context, we also hypothesize that large language models may act as intelligent assistants for air traffic control operators, helping to manage routine tasks. More critically, these agents can play a decisive role in conflict resolution strategies—identifying potential conflicts and suggesting optimal maneuvering strategies. By leveraging the function-calling capability of the language model, they can also interact with the air traffic simulator and start learning air traffic control experiences like a new air traffic controller in training.

An exciting avenue for enhancing LLM-based approaches is integrating reinforcement learning with human feedback (RLHF), though it is outside the scope of this paper. RLHF allows LLMs to learn not only from data but also from feedback provided by human operators. This integration holds promise for improving conflict resolution in ATM by maximizing the overall "return" in terms of both efficiency and human trust.

In real-life operations, we envision a 24/7 monitoring system where the large language model agent constantly checks for potential conflicts, resolves them, and communicates reasoning and commands to both operators and pilots using radiotelephony phraseology. However, in the current project, we test our agent in a simulation environment. Instead of continuous monitoring, we load pre-defined conflict scenarios, and the agent resolves each conflict before moving on to the next. Our goal is to ultimately develop a virtual air traffic assistant that can reason and explain its decisions like a human, automating conflict resolution directly within the simulation environment.

This paper explores a novel application of the large language model embodied agents in air traffic control. Our agent is able to interact with air traffic scenarios, monitor traffic, build up experiences, and resolve conflicts, all the while providing reasons for its behavior like an air traffic controller. The study assesses how effectively large language model agents can resolve air traffic conflicts and discusses in detail the limitations and potential for adopting our approach with human-like reasoning capabilities to assist air traffic controllers.

The structure of the paper is as follows. section II explains the main concept of language model embodied air traffic control agents and how they gain experiences and communicate with the BlueSky simulator. section III explains the details of our experiments in using different types of agent architectures to solve aircraft conflicts and presents the results. We then provide in-depth discussion in section IV before concluding our work in section V.

II. Methodology

In this section, we discuss our efforts to develop two different large language model embodied agent frameworks, which are capable of interacting with the BlueSky simulator [10], monitoring and interpreting traffic situations automatically, and producing instructions to solve air traffic conflicts autonomously and in real-time.

A. Understanding Large Language Models

Large Language Models (LLMs) are sophisticated AI systems that understand and generate human-like text by analysing and predicting language patterns. At their core, LLMs operate on a text-to-text basis, meaning they take a sequence of text as input and produce a sequence of text as output. This text processing capability is enabled by a series of steps, primarily involving tokenization, embedding, and the use of neural network architectures called Transformers.

Tokenization is the first step in processing text. Tokenization involves breaking down text into smaller, manageable units called tokens, which can be individual words, subwords, or even single characters, depending on the model's design. For example, the word "aircraft" is tokenized into two subwords, air craft [11]. This process is essential because it converts the raw text into a format that the LLM can interpret and manipulate later on.

`GPT-4o` and `Llama3-70B` models use a tokenization method based on Byte-Pair Encoding (BPE) [12], implemented with the tiktoken library [13], [11]. Tiktoken is an efficient tokenization library designed to handle various language modelling tasks, including handling large texts and diverse vocabularies. BPE is a tokenization method that starts with a base vocabulary consisting of individual characters and iteratively merges the most frequent pairs of characters or subwords to form longer tokens. This approach allows the model to create a compact vocabulary that can efficiently represent both common and rare words, including those not seen during training. The tiktoken library, which both `GPT-4o` and `Llama3-70B` use for tokenization, implements BPE in a way that is optimized for performance and flexibility. Tiktoken uses a precomputed vocabulary and merges rules to tokenize

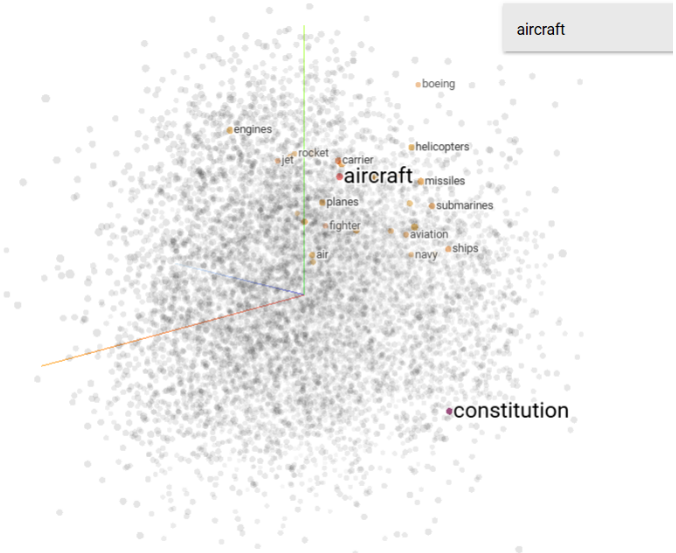


Figure 1. High-Dimensional Embeddings Vector Space

input text quickly. It is designed to handle a wide range of text data, from natural language sentences to code and domain-specific terminologies.

After tokenization, each token is converted into a numerical representation, known as an embedding. An embedding is a high-dimensional vector that captures the semantic meaning of a token in the context of its surrounding words. These vectors are designed so that semantically similar words (like "planes" and "aircraft") have embeddings that are close together in this high-dimensional space, while dissimilar words (like "aircraft" and "constitution") are far apart (Figure 1).

The LLM uses these embeddings as inputs to its neural network architecture. The most common architecture for LLMs is the Transformer [14], which is designed to handle large sequences of data and understand complex language patterns. The Transformer model comprises multiple layers, each containing two key components: the self-attention mechanism and feed-forward neural networks. The self-attention mechanism is a critical component that allows the model to focus on different parts of the input text at each step. Instead of treating each word in isolation, self-attention assigns a weight to each token in relation to every other token in the input sequence. This means that the model can learn which words are most important to predicting the next word, taking into account the entire context of the sentence. For example, in the sentence "The aircraft is on a collision course," the model learns to associate "collision" strongly with "course" and "aircraft" with "is," which helps it understand the overall meaning more accurately.

The self-attention mechanism calculates a weighted sum of the token embeddings, where the weights are determined by how relevant each token is to every other token. These weights are computed using dot products of the embeddings, followed by a normalization step using the softmax function, which converts the scores into probabilities. The self-

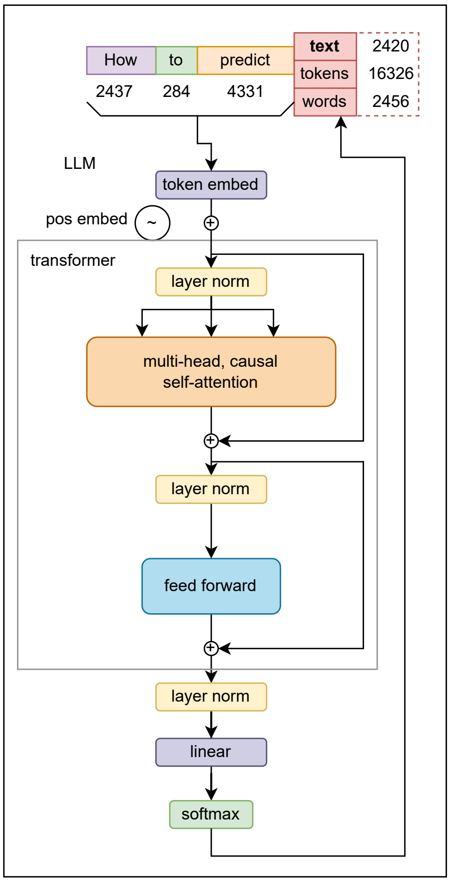


Figure 2. Transformer Predicting The Next Token [15]

attention mechanism allows the model to learn relationships between distant words in the input sequence, capturing long-range dependencies that are crucial for understanding complex sentences.

The outputs of the self-attention layers are then fed into feed-forward neural networks that further refine the embeddings by applying a series of transformations. Each layer in the Transformer architecture learns progressively more abstract and complex features of the language, from basic syntax to intricate semantic relationships. As the model processes the input sequence through multiple layers, it generates a probability distribution over the possible next tokens. The token with the highest probability is selected as the next word in the generated text. This process is repeated iteratively to produce coherent and contextually relevant text output.

While LLMs are powerful in generating human-like responses, their initial design is limited to static text output. They cannot interact with external environments or systems dynamically. This is where function calling comes into play, significantly extending the capabilities of LLMs beyond text generation.

B. Function Calling

Function calling is a powerful feature that extends the capabilities of Large Language Models (LLMs) beyond static text generation, enabling them to interact dynamically with external systems, tools, or environments. Unlike traditional LLMs that only generate text-based responses, function calling allows these models to execute specific actions or functions, retrieve external data, and perform tasks that require real-time interaction with other software or systems. Function calling involves instructing an LLM to trigger a predefined function based on its understanding of the input prompt. Certain LLMs are trained to recognise when to call functions using a combination of supervised fine-tuning and reinforcement learning. During the fine-tuning phase, the models are exposed to datasets containing examples where function calls are required. These examples include both the natural language input and the corresponding structured function call outputs. By learning from these examples, the model develops the capability to determine when and how to call functions in various contexts. In our system, Python functions are referred to as tools, and the LLM has the capability to determine when and which tools to call, along with the appropriate arguments.

There are quite a few frameworks for function calling, but The OpenAI Tools and ReAct (Synergizing Reasoning and Acting in Language Models) framework are the main popular choices. OpenAI tools framework is not only for OpenAI models, but other models, including open-sourced can use the framework. It is built to handle multiple tools with a JSON-based function calling mechanism, allowing structured input directly passed to functions. This framework is highly modular and capable of interacting with tools through predefined function signatures. The agent uses JSON-formatted inputs, making it easier to handle multiple arguments and various data types like strings, numbers, and arrays. This structured approach ensures higher reliability when interacting with tools. Each step is well-defined, and the function returns are processed without the need for complex string parsing. On the other hand, the ReAct Agent employs a string-based reasoning framework, where the language model dynamically generates both reasoning traces and task-specific actions in an intertwined manner. The string-based interaction means that the model processes textual instructions and requires parsing of each action step. This comes at the cost of requiring more complex parsing of string inputs and outputs, which is less efficient compared to JSON-based function handling.

The primary difference between these two frameworks lies in how they handle input and output. OpenAI Tools framework uses structured, machine-friendly JSON format, which enables it to handle complex, multi-argument function calls with minimal parsing. This is ideal for scenarios where precise tool interactions are essential. In contrast, the ReAct framework interprets and processes text-based reasoning traces, but requiring constant string parsing.

In the example below, the tool CONTINUEMONITORING that monitors the airspace requires a single numeric input, which is the duration in seconds, but the model's output, "60 (monitor for 1 minute)," combines an explanation with the number, leading to a validation error because the tool cannot interpret the mixed format. This issue arises due to the non-deterministic nature of LLMs, making string parsing challenging across various scenarios.

```
Thought: I need to monitor the airspace for conflicts
and resolve them by changing the heading of aircraft.
Action: CONTINUEMONITORING
Action Input: 60 (monitor for 1 minute)

error: 1 validation error for CONTINUEMONITORING schema
```

In contrast, the OpenAI Tools framework utilizes a JSON-based function-calling mechanism that specifies arguments and their corresponding values more explicitly. For example, when invoking the ‘CONTINUEMONITORING‘ tool within the OpenAI Tools framework, the function call would look like this:

```
Invoking: CONTINUEMONITORING with {'duration': 10}
```

Here, the argument 'duration' is clearly defined with a numeric value of 10 (LLM selected to monitor for 10 seconds), avoiding any ambiguity or need for additional parsing. This structured approach not only ensures higher reliability when executing tool calls but also allows for the seamless handling of complex inputs with multiple arguments and data types. As a result, we found the OpenAI Tools framework to be far more suitable as it has better precision and error-free execution.

With the OpenAI Tool framework the LLM can generate a response not only in the content section, providing a descriptive answer, but also includes additional key-value pairs specifying the tool information. The function `SendCommand(command)` is a tool that sends commands directly to the BlueSky simulator. This information indicates which tool the LLM intends to use, along with the generated arguments required for that specific tool.

```
AIMessage(
  content="There is a conflict between FLIGHT1 and FLIGHT2. The
        current heading of FLIGHT1 is 000 degrees, and the
       heading of FLIGHT2 is 300 degrees. To resolve the
       conflict by changing headings, I will adjust the
       heading of FLIGHT1 to create a greater separation.\n\
       nLet's change the heading of FLIGHT1 to 030 degrees to
       increase the horizontal separation.",
  additional_kwargs={
    'tool_calls': [
      {
        'id': 'call_y1HiHWG2KvUxmnzaQAP0l488',
        'function': {
          'arguments': '{"command":"HDG FLIGHT1 030"}',
          'name': 'SENDCOMMAND'
        },
        'type': 'function'
      }
    ]
  },
  tool_calls=[
```

```
    {
      'name': 'SENDCOMMAND',
      'args': {'command': 'HDG FLIGHT1 030'},
      'id': 'call_y1HiHWG2KvUxmnzaQAP0l488'
    }
  ]
)
```

C. Data Communication with BlueSky

The communication between the BlueSky simulator is designed to enable real-time interaction, allowing commands to be sent to the simulator and output to be received efficiently. This communication is facilitated through a BlueSky client that connects to the BlueSky simulator and handles data exchange.

The BlueSky Client object is used to establish a connection to the simulator at a specific IP address and port. The client is responsible for managing all communication between the simulation tools and BlueSky.

To send commands to the BlueSky simulator, the client uses the `send_event` method. This method allows the client to send specific commands formatted as strings to the simulator:

```
client.send_event(b"STACK", command)
```

The output from the BlueSky simulator is captured and processed using a function that continuously checks for new output. This function repeatedly calls the client to update its state and captures any new output that the simulator produces. The captured output is stored in memory and processed until the function determines that the simulator has completed processing the current command.

To achieve this, a context manager is employed to temporarily redirect and capture output from the standard output (stdout), where BlueSky typically prints its outputs. The client method is called repeatedly to check for new outputs, and the captured information is processed accordingly.

The use of the context manager for capturing standard output is essential for two main reasons: firstly, it ensures that the output captured by the client matches exactly what is displayed in the BlueSky GUI (graphical user interface) terminal, allowing both the LLM agent and human users to see the same information. This alignment facilitates better debugging and understanding of the LLM agent's actions in controlling the airspace. Secondly, the client does not have a built-in method to return an output after executing a command — nothing is returned. By redirecting the standard output to a temporary buffer, the client can capture, process, and utilize the output programmatically. This approach enables real-time monitoring and interaction with the simulation environment while maintaining consistency between the outputs seen by the LLM agent and those displayed on the GUI.

To extend the BlueSky simulation functionality and enable more advanced operations for the LLM agent, several custom plugins were developed. These plugins are used in the tools (subsection II-F which the LLM agent has control over it.

Algorithm 1: Receive Output from BlueSky

```
Initialize empty output buffer
Set counter for consecutive empty outputs to 0
while True do
    Use capture_stdout to redirect output
    Call Client update method to check for new output
    Retrieve captured output from buffer
    if captured output is empty then
        Increment empty output counter
    else
        Reset empty output counter
        Append captured output to buffer
    end
    if empty output counter exceeds threshold then
        break loop
    end
end
Call Client update method one last time
return complete output buffer
```

The first plugin, `GETACIDS`, is designed to extract detailed information about all aircraft currently in the simulation. By sending `GETACIDS` command to BlueSky, each aircraft's position (latitude and longitude), heading, altitude, speed and Mach number is returned by the simulator. At the start of the conflict, there is no way for the LLM agent to know which aircraft are currently in the airspace. The ability to access the aircraft details simplifies the process of monitoring air traffic and provides the agent with real-time data to make more informed decisions.

Algorithm 2: Get All Aircraft Information Plugin

```
Initialise empty list ac_ids_lst
Set ac_ids_lst to list of all aircraft IDs from
  traf.id
Echo list of aircraft call signs
foreach aircraft ac in ac_ids_lst do
    Echo ac;
end
```

The second plugin, `SHOWTCPA`, focuses on conflict detection by showing aircraft pairs that are in conflict and essential conflict information. BlueSky's built-in conflict detection method (enabled via the ASAS system) is used to detect potential conflicts between aircraft. The plugin provides detailed information on the conflict pairs, including the TCPA, heading differences, horizontal, vertical and total separations, and the remaining distance and time until the closest point of approach. This plugin is essential for the LLM agent to monitor airspace safety. By retrieving details about aircraft conflicts, the agent can analyse the conflicts in terms of time and distance, enabling it to take precautionary actions to resolve these conflicts.

Algorithm 3: Get Conflict Information Plugin

```
if no conflict pairs are detected from traf.cd then
    Echo "No conflicts detected"
    return
end
Echo "Aircraft Pairs in Conflict and their TCPA (sec):"
Initialise empty set processed_pairs
Initialise empty set involved_aircraft
Convert traf.cd.confpairs_unique to sorted list of
  conflict pairs
foreach conflict pair pair in sorted conflict pairs do
    Retrieve tcpa_value, qdr_value, dcpa_value,
      tLOS_value
    Retrieve index_0 and index_1 of aircraft from
      traf.id
    Calculate heading difference d_hdg
    Calculate horizontal distance in meters using
      haversine
    Add the aircraft in pair to involved_aircraft
    Prepare conflict information including TCPA,
      heading difference, horizontal distance, vertical
      separation, DCPA, and tLOS
    Echo conflict information
end
Echo the number of conflict pairs
Echo aircraft altitude information
```

The Haversine function that is used in algorithm 3 is a mathematical formula used to calculate the shortest distance between two points on the surface of a sphere, based on their latitude and longitude. It accounts for the curvature of the Earth, making it ideal for measuring distances on the Earth's surface. One of the advantages of the Haversine function is its relatively simple mathematical formulation, which provides sufficient accuracy for most practical purposes while being computationally efficient. Although the Earth isn't a perfect sphere in reality, we assume it is for the purpose of our simulator experiments, where the Haversine formula is acceptable.

We use the BlueSky built in state-base conflict detection method to detect conflicts. State-based conflict detection is a method used in air traffic management to predict potential conflicts between aircraft by extrapolating their current states—positions and velocities—into the future. This technique assumes that each aircraft will maintain its present trajectory and speed over a specified look-ahead time interval. By analysing these projected paths, the system identifies if and when aircraft might breach predefined safety zones, both horizontally and vertically. In this approach, the initial step involves collecting the current positions and velocities of all aircraft within a certain airspace. Using these data, the system calculates the relative positions and velocities between pairs of aircraft. It determines the bearing and distance from one aircraft (ownship) to another (intruder) and computes the components of their velocities in the eastward and northward directions. The core of state-based conflict detection lies in calculating the time to the closest point of approach (CPA) and the minimum separation distance at that point. The time to CPA, denoted as t_{CPA}, is found by analysing the relative motion of the aircraft and identifying the moment when they are closest to each other if they continue on their current paths. The minimum distance at CPA, d_{CPA}, is then compared against predefined horizontal and vertical protection zones—typically represented by a radius r_{PZ} for horizontal separation and a height h_{PZ} for vertical separation. A conflict is predicted if the calculated d_{CPA} is less than the horizontal protection zone ($d_{\mathrm{CPA}} < r_{\mathrm{PZ}}$) and if, at that time, the vertical separation is also less than the vertical protection zone. The system computes the times when aircraft enter and exit these protection zones, considering both horizontal and vertical dimensions. By comparing these times, it identifies intervals during which both separation criteria are violated simultaneously. This method efficiently provides predictions of potential conflicts without requiring knowledge of the aircraft's future intent. However, it operates under the assumption that all aircraft will maintain their current velocities, which may not account for upcoming changes in flight paths. Despite this limitation, state-based conflict detection remains a necessary tool for detecting conflicts due to its simplicity and computational efficiency.

The third plugin, `CRASHDETECTION`, is responsible for monitoring and detecting crashes or close calls between aircraft. BlueSky does not automatically inform the user when a crash or near miss occurs, so this plugin is needed for tracking safety violations. The plugin calculates the distance between aircraft in the simulation, and if the distance between any two aircraft falls below 300 meters, it logs the event as a crash or close call. This allows to evaluate LLM agent's performance. The crash detection process is based on the Haversine formula. By combining horizontal and vertical distances between aircraft, the plugin determines the total distance between aircraft pairs and logs any instances where the separation falls below the safety threshold.

The 300-metre threshold ensures reliable crash detection in the simulation. Using smaller values like 200 metres led to cases where aircraft passed the threshold between updates. Lowering the time step (currently 0.1 seconds) introduced small delays in communication between BlueSky and the LLM agent, especially with an increasing number of conflicting aircraft pairs.

D. Large language model embodied Agent

The main concept behind agents is utilising a language model to decide the order and selection of actions to perform. The language model functions as a reasoning engine, dynamically determining which actions to take and their execution sequence. By providing a proper application programming interface, LLMs can be integrated with various tools and virtual or real environments, transforming them into embodied agents.

Algorithm 4: Crash Detection Algorithm

```
calculate_distance(aircraft_list) : foreach conflict pair of aircraft (i, j) do
    Calculate horizontal distance using haversine
    Calculate altitude difference
    Compute total distance using the Pythagorean theorem
    if total distance < 300 meters then
        Log crash with aircraft IDs
        Print crash information to console
    end
end
while simulation is running do
    Every 0.1 seconds, get the list of all aircraft positions and altitudes
    Call calculate_distance to check for conflicts
end
```

An embodied agent can either utilize specific tools, such as Python functions with arguments or operate independently to generate responses.

In our research, we designed such agents that can interact with the air traffic control interface - the BlueSky simulator. By providing the proper objective in a text (called *prompt*), the agent is set to solve conflict scenarios.

Figure 3 shows the overview of the process, beginning with the construction of a prompt that integrates the system prompt, user prompt, and tools descriptions. The large language model then evaluates whether a tool is needed for the task at hand. If a tool is required, the agent executes the selected tool with the specified arguments. For example, to change an aircraft's altitude, the agent would use a `SendCommand()` tool with a generated altitude command. This command is sent to the simulator, and the output from the simulator is then integrated back into the prompt for further processing. This cycle repeats until the large language model determines that no additional tools are needed.

In the end, the agent can also provide a summary of the situation and reasons for the conflict-solving strategies. An experience document (subsection II-G) is then created and subsequently uploaded to an experience library, which can be retried to further enhance the agent's knowledge base and capabilities for more complex tasks.

To demonstrate this process, Figure 4 presents a scenario where a single agent effectively resolves a converging three-aircraft conflict. The resolution process begins with the agent querying all relevant aircraft data through the `GetAllAircraftInfo()` tool. The agent automatically assesses the conflict dynamics between each pair of aircraft using `GetConflictInfo()`. Based on the results, the agent then strategically issues a heading change to aircraft AB112, directing it to alter its course to 225 degrees. This directive is executed via the `SendCommand()` tool, utilising the command `HDG AB112 225`.

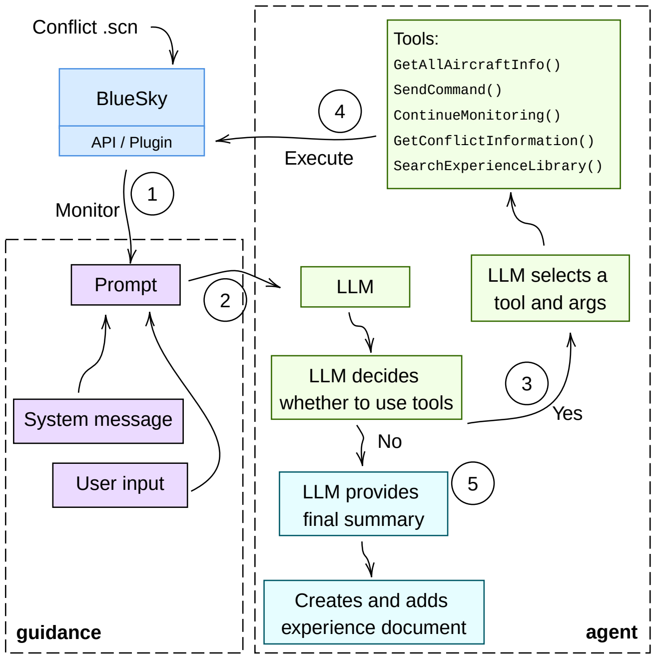


Figure 3. The language model embodied single agent setup

After this initial conflict mitigation, the agent re-evaluates the aircraft and conflict information. It then proceeds to issue another command - this time decreasing the altitude of aircraft AB426 by 2000 feet, further solving the remaining conflict. After re-assessing the situation and confirming the resolution of all potential conflicts, the agent concludes its task, having successfully ensured a safe outcome.

We have also developed a multi-agent system capable of handling an unrestricted number of LLM embodied agents and facilitating increasingly complex challenges. This system is illustrated in Figure 5.

In this multi-agent system, we designed three types of agents: the planner, the executor, and the verifier. The planner agent is responsible for generating a conflict resolution plan. It begins this process by monitoring the airspace and analysing detected conflicts.

Once a plan is formulated, it is passed onto the executor agent. The sole function of this agent is to issue appropriate commands to BlueSky. The inclusion of the executor agent serves a critical functional purpose beyond transparency and exposition. While it might appear that its tasks could be integrated into the planner or verifier agents, the executor plays a distinct role in our system architecture. Research has shown that requiring LLMs to produce their answers in strict, structured formats can negatively impact their reasoning performance. For instance, the paper "Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models" [16] highlights that imposing rigid output constraints can degrade an LLM's ability to reason effectively. While function calling does involve structured output, it is specifically designed to facilitate the integration of LLMs with software functions. The model is guided to produce outputs that match the expected parameters of predefined functions. LLMs that have ability to do function calling are designed already in such way during training and fine tuning process. In our system, the planner agent generates conflict resolution plans in natural language to leverage its full reasoning capabilities without the constraints of a structured format. The executor agent then interprets these natural language instructions and translates them into executable commands for the simulation environment. Moreover, because LLM outputs can vary due to their non-deterministic nature and differences across models, designing a traditional parser to handle all possible variations in the planner's output would be challenging. The executor agent, powered by an LLM, acts as an advanced interpreter that can consistently process the planner's diverse outputs. Integrating this functionality into the planner or verifier could complicate their primary functions and potentially reduce overall system effectiveness. Thus, the executor agent is essential for ensuring accurate and reliable executions of the resolution plan.

After the execution of the plan, the verifier agent plays a critical role in ensuring the efficacy of the conflict resolution. This agent continues to monitor the airspace to confirm whether any conflicts remain unresolved. If conflicts persist, the verifier agent devises a new resolution plan, which is once again forwarded to the executor agent for implementation. Conversely, if no further conflicts are detected, the conflict-solving task is concluded. Additionally, when the experience library is activated, both the planner and verifier agents can search in this library to retrieve insights from previously encountered conflicts.

E. Prompt

The prompt serves as a critical link between the objectives, agent actions, and the underlying language model. We have designed a prompt template to ensure the clarity and relevance of the information processed by the LLM, containing four different components:

```
system_prompt:    pre-crafted text on role and objectives
user_input:       instructions from human
chat_history:     memories about llm inputs and outputs
agent_scratchpad: memories about environment interactions
```

Figure 4. Single Agent solving 3 aircraft conflicts without experience library. The LLM embodied agent automatically decides when and what commands (in green text) are to be invoked at all stages.

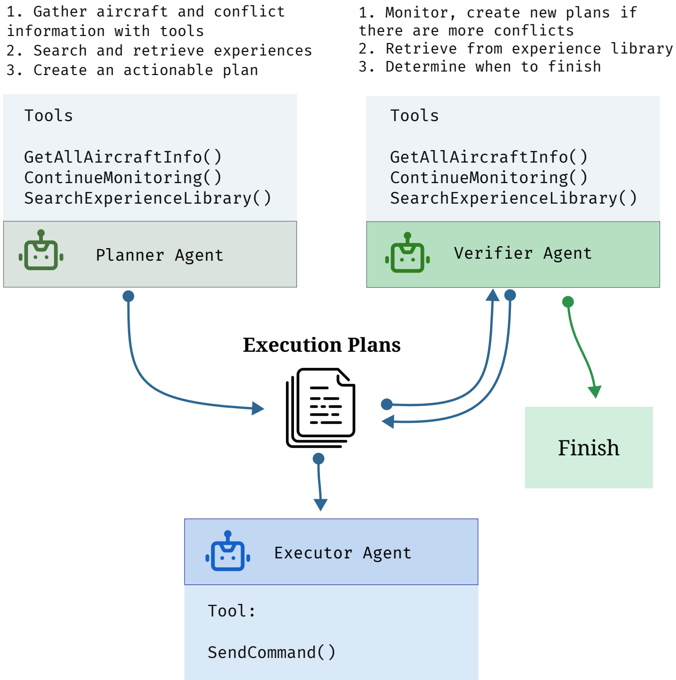


Figure 5. The structure of the multiple language model embodied agent, containing planner, verifier, and executor agents.

System Prompt: This component is crafted to provide both context and explicit instructions to the agent. Each agent receives tailored directives specific to their role. For instance, the planner agent is instructed to gather aircraft information, monitor airspace, and provide an actionable plan according to the separation requirements that would also avoid introducing new conflicts.

User Input: A brief instruction on tasks and preferences that can enhance agent performance. For instance, the planner agent may be asked to check for conflicts and create a plan based on preferences, such as changing heading, altitude, or both. These instructions are less detailed than the system prompt.

Chat History: Acting as a memory block, which stores the inputs and the output with the language model, it maintains a continuous record of interactions.

Agent Scratchpad: This component memorises descriptions of the tools used, logs all intermediate steps, and records results from the tools. It is vital for tracking the agent's operational processes and the adaptations made during task execution.

Tools are also converted into a prompt and appended to the final prompt. In order for LLMs to interact with tools through function calling, these tools need to be converted into a structured format. An example below, shows the process of converting a Python function (representing a tool) into a JSON format that can be understood by the LLM. This transformation ensures that the LLM can reference the available tools and their corresponding functionalities during interaction.

`GetAllAircraftInfo()` tool retrieves aircraft information from the BlueSky:

```
@langchain_tool("GETALLAIRCRAFTINFO")
def GetAllAircraftInfo(command: str = "GETACIDS"):
    """
    Get each aircraft information at current time: Position [
        Pos] (lat, lon), Heading [Hdg] (deg),
    Track [Trk](deg), Altitude [Alt](ft), Vertical speed [V/S](
        feet per mintute. Negative V/S - flying down,
        Positive V/S - flying up, 0 - stays at same altitude)
        , Speed [CAS/TAS/GS] (calibrated/true/ground air
        speed, knots per second) and mach number.
    Parameters:
    - command: str (default 'GETACIDS')
    Example usage:
    - GetAllAircraftInfo('GETACIDS')
    Returns:
    - str: all aircraft information
    """
    client.send_event(b"STACK", "OP")
    client.send_event(b"STACK", "GETACIDS")
    time.sleep(0.8)
    sim_output = receive_bluesky_output()
    return sim_output
```

The tool is transformed into a structured JSON format. The key elements of the tool—such as its name, description, parameters, and expected output—are encoded in this format. The JSON format for the above tool looks like:

```
{
    "name": "GETALLAIRCRAFTINFO",
    "description": "Get each aircraft information at current
        time: Position [Pos] (lat, lon), Heading [Hdg] (deg)
        ,\nTrack [Trk](deg), Altitude [Alt](ft), Vertical
        speed [V/S] (feet per minute. Negative V/S - flying
        down, Positive V/S - flying up, 0 - stays at same
        altitude), Speed [CAS/TAS/GS] (calibrated/true/ground
         air speed, knots per second) and mach number.\n\
        nParameters:\n- command: str (default 'GETACIDS')\n\
        nExample usage:\n- GetAllAircraftInfo('GETACIDS')\n\
        nReturns:\n- str: all aircraft information",
    "parameters": {
        "type": "object",
        "properties": {
            "command": {
                "default": "GETACIDS",
                "type": "string"
            }
        }
    }
}
```

- **Name:** The unique identifier for the tool, which is used to invoke the tool in the LLM's function-calling mechanism.
- **Description:** A detailed explanation of what the tool does, including the expected input and output. This description helps the LLM understand how to use the tool effectively.
- **Parameters:** The input parameters required for the tool. In this case, the tool accepts a command string, with a default value of `'GETACIDS'`.

F. Tools

Our system integrates several specialized tools (functions in Python programming language) to facilitate interactions between the large language model and the BlueSky simulator. These tools are crucial for the effective execution of tasks and data retrieval:

- `GetAllAircraftInfo()` : This tool sends a command to BlueSky and retrieves a comprehensive list of aircraft, detailing their position, heading, track, altitude, vertical speed, calibrated, true airspeed, and ground speed, as well as Mach number.
- `GetConflictInfo()` : This tool sends a command to BlueSky and retrieves information about aircraft pairs in conflict. It provides details such as Time to Closest Point of Approach (TCPA), heading differences, separation distances (total, vertical, and horizontal), distance to Closest Point of Approach distance (DCPA), time to Loss of Separation (tLOS), and altitude information.
- `ContinueMonitoring(duration)` : This tool commands BlueSky to retrieve changes in conflict status over a specified duration, enabling ongoing monitoring of the airspace. It retrieves conflict information at the beginning and at the end of the duration.
- `SendCommand(command)` : This tool sends a traffic command to BlueSky and retrieves the resulting output from the simulator, allowing for dynamic interaction with the simulation environment.
- `SearchExperienceLibrary(args)` : This tool queries the experience library and returns the most relevant experience document based on different arguments, including conflict description, number of aircraft involved, and the formation of the conflict.

It is important to emphasize that the large language model decides when to utilize a tool, and it is also responsible for generating proper functional arguments that enable precise and context-appropriate responses. This function-calling capability enhances the agent's ability to interact with and manipulate the environment effectively and freely.

As shown in Figure 6, the process of calling a tool by an LLM involves several key steps. It begins with the initiation of the LLM, which receives the entire input prompt. This input prompt comprises the system prompt, the user input, the chat history, and the agent's scratchpad (subsection II-E). Upon receiving the full prompt, the LLM can either generate a normal text response, denoted as `content`, and (or) it may decide to perform a function call, which is represented as `tool_calls`. In this instance, rather than generating any textual response, the LLM opts to call a function. The function to be invoked is specified by its `name`, and any necessary input arguments are provided through `args`. In this example when the function is called, a command is sent to the BlueSky simulator, which processes the request and returns the output about all aircraft information. The entire message generated by the LLM encapsulated in `AIMessage`, is subsequently appended to the prompt, specifically within the agent's scratchpad. Additionally, the returned function output is appended as well. This process is cyclical in nature, allowing the LLM to continue generating additional function calls as necessary. If no further function call is generated, it is assumed that the LLM intends to conclude its current turn.

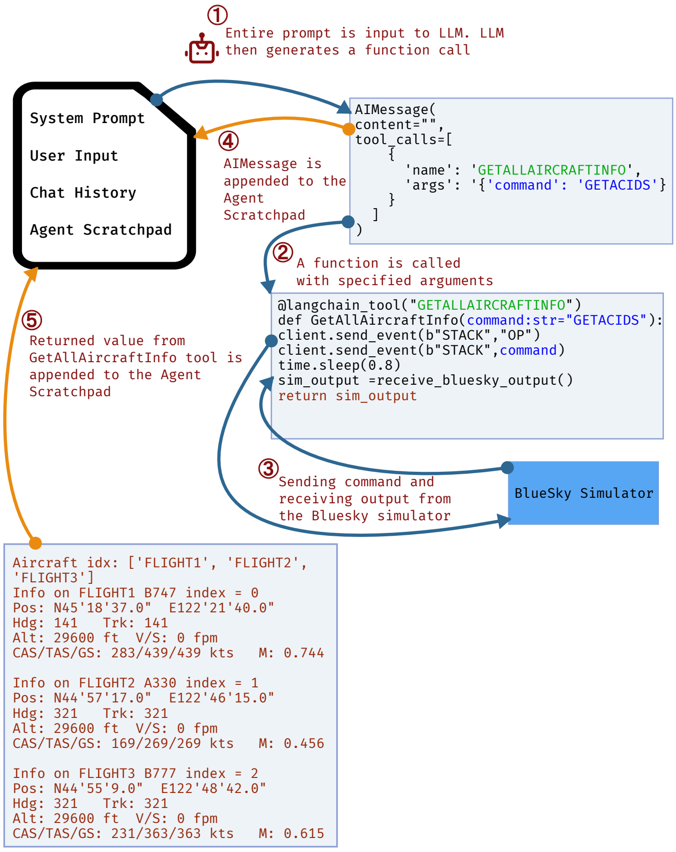


Figure 6. LLM deciding to call a tool

In principle, it is also possible for the agent to write its own tools, considering that sufficiently large language models are also capable of code generation. However, this was not tested in our experiments.

G. Experience Library

The Experience Library is a crucial component that enables our LLM embodied agent to recall stored memories about past conflict solution experiences. We use an open vector database, `Chroma` [17], to store and retrieve past conflict resolutions effectively.

A vector database is a specialised type of database designed to store, manage, and retrieve vector embeddings, which are high-dimensional numerical representations of data. These embeddings contain semantic information crucial for tasks like similarity search or memory recall. Vector databases excel at comparing these embeddings, making them ideal for applications that need to handle complex data relationships, such as recalling past experiences. Unlike traditional databases, which store and query exact matches of scalar data (like text or numbers), vector databases are optimised for storing and searching

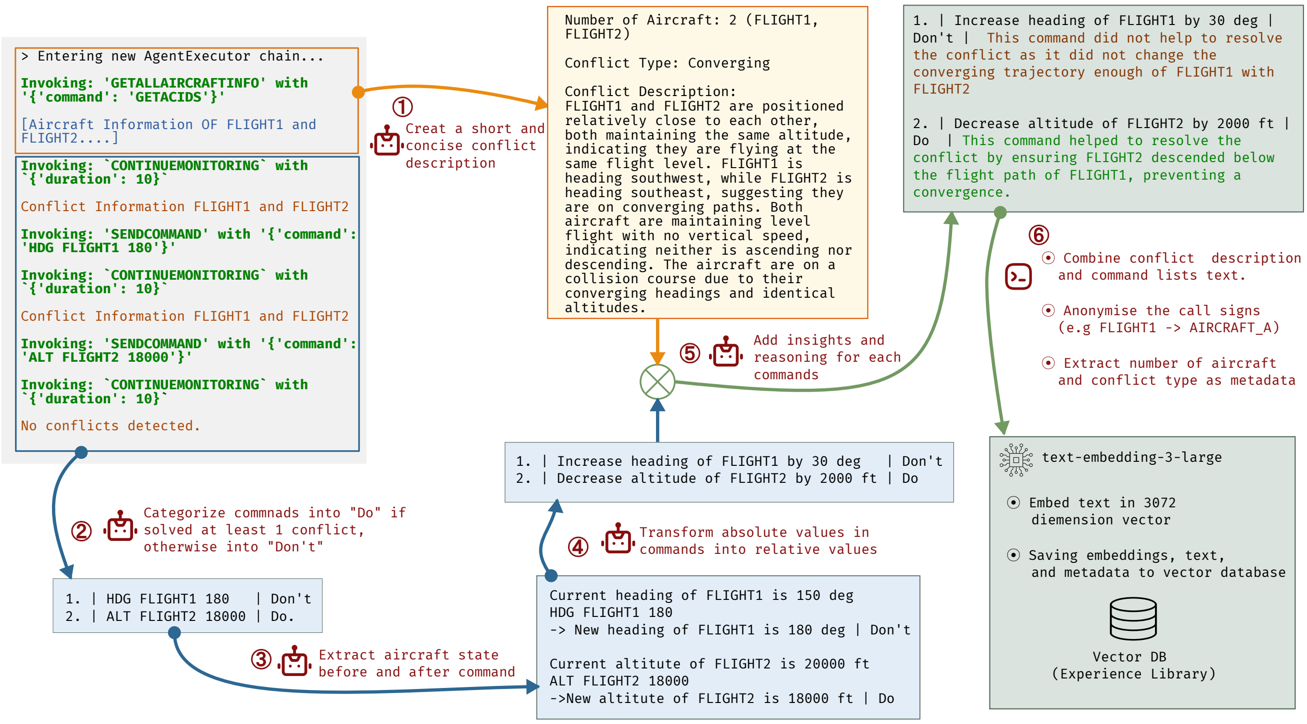


Figure 7. Creating the experience document from the operation logs of the agent

vectors—dense, multi-dimensional arrays of numbers. These vectors are compared using similarity measures rather than exact matching, enabling tasks like finding the closest match to a given query or retrieving semantically similar results. The core mechanism behind a vector database involves three main stages:

1) **Indexing:** When data (such as text or images) is fed into an embedding model, the model creates a vector embedding that captures its meaning. This embedding is stored in the database.
2) **Querying:** When a user searches or queries the system, the query is converted into a vector embedding using the same model. The vector database then compares this embedding to those already stored, identifying the closest or most similar vectors.
3) **Retrieving:** The most similar vectors are returned, typically linked to the original data they represent, such as text or past experiences.

This structure makes vector databases ideal for recalling experiences, as they can quickly find and retrieve relevant memories based on semantic similarity rather than exact keyword matches.

G.1 Creation of Experience Documents

After an LLM agent is done with a conflict (can be resolved or unresolved), it processes the entire conflict resolution log to create an *experience document*. The conflict resolution log provides a detailed record of the agent's interactions and decisions throughout the conflict resolution process. It tracks each tool invocation by the agent, capturing the data returned from these tools, such as updated status information or conflict metrics. A concise conflict description is generated with the language model based on the initial states of the aircraft and the conflict information. It then categorises the executed commands into whether they are helpful or not helpful.

Commands that have eliminated at least one conflict pair are deemed helpful, while others are not. The absolute values (like altitudes and headings) of these commands are converted into relative values to ensure the applicability of past experiences to new situations. Using absolute values, such as specific altitudes (e.g., 20,000 feet), would limit the reuse of conflict resolution strategies to identical scenarios. For example, a command to climb to 22,000 feet might have been effective in resolving a conflict at a certain altitude but would not be relevant if the new conflict occurs at 15,000 feet. By converting these absolute commands into relative values, such as a climb of 2,000 feet, the solution becomes adaptable to any similar conflict, regardless of the initial conditions. This transformation ensures that the experience document can guide the LLM agent in resolving conflicts across varying contexts, promoting flexibility and more generalised learning.

The conflict description and categorized list of commands are then combined into a single experience document. Finally, the language model enhances the document by generating insights and reasoning for each command, tailored to the specific conflict.

These insights are concise explanations that provide context for why certain commands were effective or ineffective. For instance, after categorising a command as helpful, the language model might generate reasoning like: *"This command was beneficial because it moved AIRCRAFT_D to an even higher* *altitude, ensuring it was well clear of AIRCRAFT_A and AIRCRAFT_B, thereby resolving the conflict by maximising vertical separation."*

The language model first reviews the conflict description to understand the scenario, including factors such as the number of aircraft involved, their positions, and how the conflict unfolded. It then analyses the sequence of commands, considering the order in which they were executed. This allows the LLM to tailor its reasoning to the specific dynamics of the conflict, ensuring that the insights it generates are relevant and context-aware. The insights are designed to be brief but informative, highlighting why a particular action was effective and offering clear guidance for future similar scenarios.

Additionally, aircraft callsigns are anonymised in the final steps of creating the experience document. This ensures that when an agent retrieves the document later, it won't be confused by the presence of the same callsigns in both the current conflict and the experience document.

The conflict description is encoded into a 3072-dimensional vector embedding using the `text-embedding-3-large` model from OpenAI[1]. The `text-embedding-3-large` model was chosen due to its robustness and simplicity to use. It provides a pre-trained model that does not require self-hosting. This model is also well-suited for generating high-quality embeddings across a variety of text types, making it versatile for different domains, including conflict analysis.

The embeddings of the experience, along with text and metadata on conflict type and the number of aircraft, are then uploaded to the vector database.

The entire experience generation process is illustrated in Figure 7. It is worth noting that we only need to encode the conflict description. This is because when an agent searches the experience library, it can describe the current conflict. Matching conflict descriptions directly yields higher similarity and the most relevant results than when comparing the full document with commands, suggestions, and insights.

G.2 Experience Library Search

When an agent wants to retrieve the closest memory from past experiences before solving the conflicts, it invokes the `SearchExperienceLibrary()` tool (shown in Figure 9). The agent first generates a concise description of the current conflict, including the number of aircraft involved and the type of conflict. The initial metadata filtering reduces the search space in terms of aircraft formation and number of aircraft. The conflict description is also encoded as a 3072-dimensional vector with the embedding model.

The search process employs the Hierarchical Navigable Small World (HNSW) [18] algorithm alongside Cosine Similarity to perform the vector search. The system returns the experience document that exhibits the highest textual similarity.

The choice of the HNSW algorithm is motivated by its efficiency and scalability, making it particularly suited for the task of vector search in large databases. HNSW is an approximate nearest-neighbor search algorithm that builds on the concept of navigable small-world networks. It is designed to handle the challenges of searching in high-dimensional spaces, where brute-force methods become computationally impractical. In a vector database, each document or experience is represented by a high-dimensional vector, and the goal of the search process is to find the top K vectors that are most similar to a given query vector. A naive approach would involve comparing the query vector with every vector in the database. However, this brute-force method has a time complexity of *O(n)*, where n is the number of vectors in the database. As the size of the database grows, this approach becomes prohibitively slow.

HNSW addresses this issue by creating a graph-based index, where each node represents a vector, and edges connect nodes to their most similar neighbours. The graph is built iteratively, starting with a small subset of vectors and gradually adding more, connecting each new vector to its closest existing nodes in the graph. This results in a multi-layered structure where nodes are connected at various levels of granularity, allowing the search to be both broad and deep. The top level of the graph is the most sparse, while the bottom level is the most dense. This design is intentional because the sparse top level allows the search to cover a broad area of the vector space quickly, providing a coarse but efficient starting point. As the search progresses to lower levels, which are denser, it allows for finer and more precise navigation among similar vectors, ultimately leading to the most accurate results (Figure 8[2]). The sparsity at the top reduces the number of nodes the algorithm needs to evaluate initially, while the density at the bottom ensures thorough exploration of the most promising areas of the vector space. When performing a search, HNSW begins at a random node and navigates through the graph by moving to the neighbouring node that is closest to the query vector. This process is repeated until no closer neighbours are found. Due to the hierarchical nature of the graph, HNSW efficiently narrows down the search space, skipping large portions of the database that are unlikely to contain similar vectors. This results in a time complexity of *O(logn)*, representing a substantial reduction in search time as compared to the brute-force method.

HNSW Space Configuration: In the context of the Chroma vector database, several parameters configure the HNSW algorithm:

- **hnsw:space**: *cosine*
  - Controls the distance metric used in the HNSW index.
- **hnsw:construction_ef**: *100*
  - Controls the number of neighbours explored when adding new vectors to the HNSW graph.
- **hnsw:M**: *16*

[1]Many other embedding models can be used, for example, https://huggingface.co/models?other=text-embeddings-inference

[2]Image source: https://github.com/vearch/vearch/wiki/Hnsw-Real-time-Index-Detailed-Design

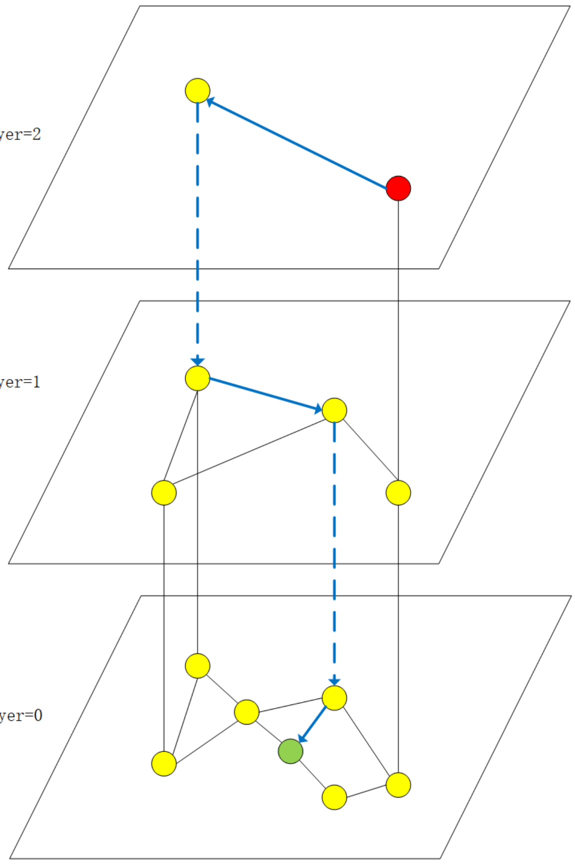


Figure 8. Searching in HNSW

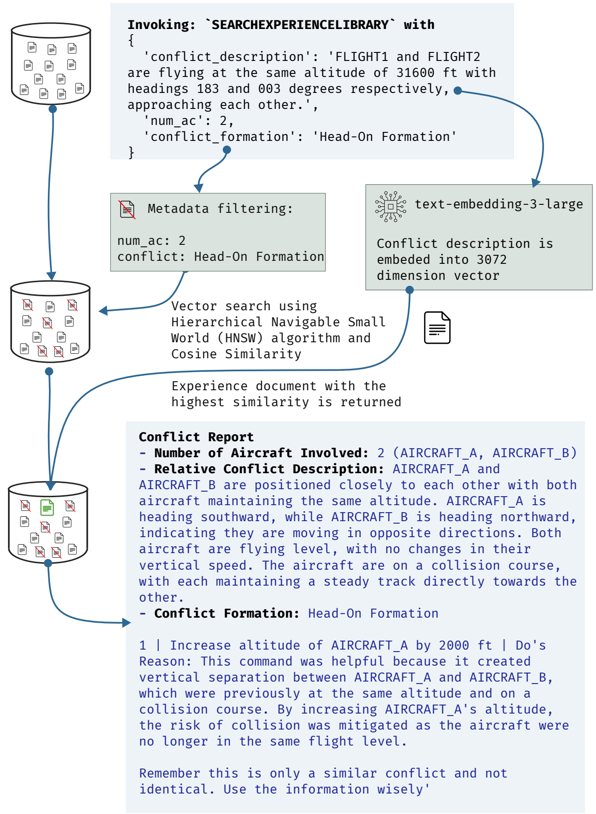


Figure 9. Filtering and searching in the experience library based on experience embeddings

  - Determines the maximum number of connections (neighbours) for each node in the graph.
- **hnsw:search_ef**: *10*
  - Controls the number of neighbours explored during the search process.

Cosine similarity is preferred over $l2$ distance (Euclidean Distance), particularly in high-dimensional vector spaces, due to a phenomenon known as the "curse of dimensionality." As the number of dimensions increases, the $l2$ distance between vectors tends to become more uniform, which reduces its effectiveness in distinguishing between vectors. This is because the distance between any two points in a high-dimensional space becomes almost the same, making it harder to identify which vectors are truly similar or different. In contrast, cosine similarity measures the cosine of the angle between two vectors, focusing on their directional alignment rather than their magnitude. Since cosine similarity is based on the angle rather than the distance, it remains effective in high-dimensional spaces, where the orientation of vectors (i.e., their direction in space) is often more informative than their absolute distance. This makes cosine similarity more robust and reliable for high-dimensional data.

III. Experiments and Results

In this section, we describe the experimental setup and evaluate the results of various agent configurations under a diverse set of simulated conflict scenarios. We explore the performances of different agent models with and without access to the Experience Library, a tool that enhances the learning process by providing past experiences. Our experiments are systematically structured to assess the agents' effectiveness in handling increasingly complex scenarios, providing insight into their ability to manage air traffic conflicts.

The experiments are divided into three main phases. First, we conduct initial tests with a limited dataset to identify the most promising models and configurations. This step helps narrow down the range of models and temperature settings for more extensive testing. In the second phase, we generate a larger dataset of conflict scenarios to thoroughly evaluate the selected models, focusing on different types of conflicts and aircraft numbers. Finally, we analyse the results, comparing the performance of single-agent and multiple-agent setups, with and without access to the Experience Library.

A. *Initial tests*

An initial experiment was conducted with a small dataset containing 12 conflict scenarios, which included conflict formations such as: head-on, parallel, t-formation and converging with three different aircraft numbers each: 2, 3, and 4 aircraft. In all scenarios, there is no altitude difference between aircraft in conflict. We tested a single agent configuration without experience library with the following models: `Llama3-8B`, `Llama3-70B`, `Mixtral- 8x7b`, `Gemma2-9b-it` and `GPT-4o`. These models were selected because they represent some of the top-performing open-source language models available, encompassing both large models (`Llama3-70B` and `Mixtral-8x7B`) and smaller models (`Gemma2-9b-it` and `Llama3:8B`). Including a range of model sizes allowed us to compare how model complexity impacts performance. Additionally, the commercial model `GPT-4o` was included to provide a benchmark against a leading closed-source model. The selection was also influenced by computational resource and hosting constraints, as detailed in subsection IV-F.

We also evaluated a range of temperatures: `0.0`, `0.3`, `0.6`, `0.9`, and `1.2`. Temperature is the main hyperparameter for LLM models, and it can significantly influence their performance. It controls how conservative or creative the model's predictions are, with lower temperatures resulting in more deterministic outputs and higher temperatures encouraging more diverse and exploratory behaviour. In Figure 10 the bar charts illustrate the predicted next token probabilities for "FLIGHT2: descend to ... " sentence example at two different temperatures using the `GPT-4o` model. At temperature 0.0, the model shows a clear preference for a single token ("260") with a probability of 53.88%, demonstrating a more deterministic behaviour. As the temperature increases to 0.6, the distribution of probabilities becomes more balanced, with "260" still maintaining the highest probability (48.71%) but other tokens gaining more likelihood, reflecting increased diversity in token selection.

By testing various temperature values, we aimed to identify the optimal setting that balances consistency and creativity for effective conflict resolution.

This initial experiment was designed to identify the most promising models based on a limited set of scenarios. The tests narrow down the number of models to focus on for later more extensive testing. We score the effectiveness of the setting based on the criteria in Table I. The scoring system used in this study is designed to reflect the critical outcomes that air traffic controllers aim to achieve in real-world conflict resolution scenarios. The highest score (1) is assigned when a conflict is successfully resolved, meaning that the aircraft maintains a safe separation throughout the scenario, aligning with the primary goal of air traffic management, which is to prevent any loss of separation. This represents a successful intervention where the agents demonstrated effective conflict resolution strategies. A score of 0 indicates a loss of separation (LoS), which occurs when two aircraft come closer than the minimum safe distance without colliding. While not immediately resulting in a collision, this outcome is considered undesirable in actual air traffic control operations, as it breaches safety protocols and could escalate into a more severe incident if not corrected promptly. The lowest score (-1) is given when a near miss or collision occurs. The inclusion of this scoring criterion allows us to assess the agent's ability to solve air traffic conflicts and ensures that the models are evaluated against strict safety standards.

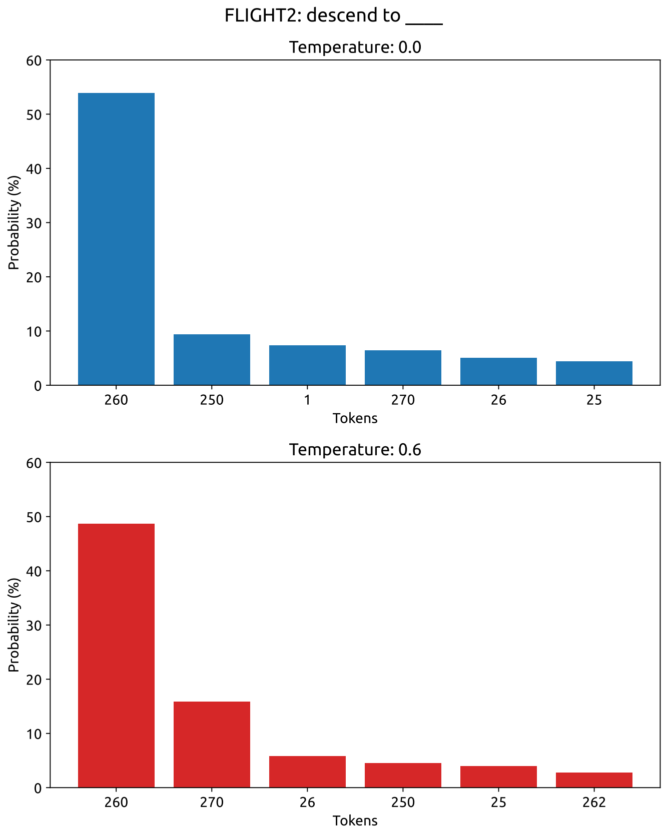


Figure 10. Effect of temperature on token prediction with GPT-4o

TABLE I. Scores for evaluating conflict resolution

Score	Outcome Description
1	Conflict is solved (successful)
0	Conflict results in loss of separation (unsuccessful)
-1	Conflict results in near miss or collision (unsuccessful)

Based on the test results in Figure 11, the `Llama3-70B` (open-source) and `GPT-4o` (commercial) models demonstrated the highest success rates in solving air traffic conflicts. In contrast, the smaller models and `Mixtral-8x7B` faced challenges, achieving success rates of only 58.3% for `Gemma2-9B`, 41.7% for `Llama3-8B`, and 50% for `Mixtral-8x7B`. Both `Llama3-70B` and `GPT-4o` exhibited optimal performance at lower temperatures (0.0 and 0.3), though an anomaly occurred at temperature 1.2 for `Llama3-70B`, which solved the initial dataset without any errors.

Although a temperature of 0.0 provides reliable results for both models, introducing a small degree of temperature is preferred, as it encourages the models to generate more creative solutions without deviating too far from accurate predictions. For both models, the temperature is capped at 2 to avoid erratic behaviour. It was decided that `GPT-4o` would be tested further with a temperature of 0.3.

To determine the optimal temperature for `Llama3-70B`, an additional test was conducted on a larger dataset comprising 120 scenarios (subsection III-B). Out of 120 cases, 35 were solved at a temperature of 1.2, while 62 were solved at 0.3. Therefore, in conclusion, both `GPT-4o` and `Llama3-70B` will be tested further with a temperature of 0.3 on the expanded dataset of 120 scenarios, alongside various agent configurations.

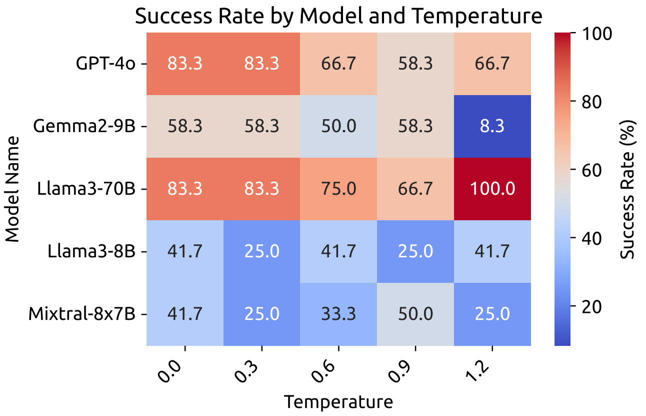


Figure 11. Success rate for different models and temperatures, tested on 12 conflict cases

B. Generating conflict scenarios dataset

To assess the performance of the `Llama3-70B` and `GPT-4o` models in solving air traffic conflicts, we generated a dataset comprised of 120 distinct conflict scenarios for BlueSky.

The pseudocode illustrates the process of generating aircraft conflict scenarios. The procedure begins with writing the ASAS activation command, which uses a state-based method to detect potential conflicts between aircraft. Following this, the first aircraft is introduced into the simulation using the `CRE` command. Subsequently, the `CRECONFS` command generates conflicts between aircraft in the scenario. The second aircraft is always placed in conflict with the first, as it is the only available option at that point. For each subsequent aircraft, there is the possibility of being in conflict either with the first aircraft or any of the previously generated aircraft, allowing for varied and realistic conflict scenarios. The parameter *dpsi* represents the heading difference between the conflicting aircraft, with values such as 0 degrees indicating parallel flight paths and 180 degrees representing head-on conflicts. In addition, horizontal and vertical TLOS (Time to Loss of Separation) values, height differences, and speeds are selected randomly to further diversify the conflict scenarios. All conflicts are created with a Distance at the Closest Point of Approach set to 0 nautical miles, implying that without any intervention from the LLM agent, these scenarios would result in a collision. This setup effectively creates real-world conflict situations, providing a comprehensive basis for testing conflict resolution strategies.

The dataset contains 40 scenarios, each with two, three, or four aircraft conflicts. The conflicts are categorised into four primary types: 1) head-on, where aircraft are on a direct collision course; 2) T-formation, which involves perpendicular flight paths; 3) parallel, where aircraft fly close parallel courses; and 4) converging, where multiple aircraft are on intersecting paths heading towards the same point. There are 30 conflict scenarios in these four types.

Algorithm 5: Generate Conflict Scenario File

Input: num-aircraft, conflict-type, dH-values, tlos-hor-values, tlos-ver-values, folder-path
Output: Conflict scenario file written in specified format

Open the file for writing in the given folder path
Write the ASAS activation command:
  00:00:00.00>ASAS ON
Randomly generate latitude, longitude, heading, flight level, speed, and aircraft type for the first aircraft
Write the first aircraft creation command:
  00:00:00.00>CRE FLIGHT1 aircraft_type lat long heading flight_level speed
Write the PAN command to centre the view:
  00:00:00.00>PAN lat long
for *j from 2 to num-aircraft* **do**
  Select target aircraft ID randomly
  Determine dpsi angle based on conflict type
  Select random values for horizontal TLOS, vertical TLOS, speed, and aircraft type
  Use corresponding dH value from dH-values list
  Write the conflict creation command:
    00:00:00.00>CRECONFS FLIGHTj aircraft_type target_id dpsi 0 tlos_hor dH tlos_ver speed
end
Close the file
return Created scenario file

In addition to conflict type, we also consider changes in flight levels. Half scenarios have all aircraft at the same level, while others half involve climbing, descending, and level flights, adding further complexity to the conflict dynamics. Examples are shown in Figure 12.

All scenarios are designed under the assumption that, without timely intervention, the aircraft involved will inevitably collide. This design ensures that each scenario presents a genuine challenge that tests the models' abilities to effectively navigate and resolve potential airborne conflicts in high-risk situations.

It is also worth noting that all these scenarios present imminent conflicts with very short response time. They are incredibly challenging for human operators, especially when involving more than two aircraft.

C. Results

These conflict scenarios are tested with single-agent and multiple-agent configurations using different language models. Figure 13 shows the success rates across different agent

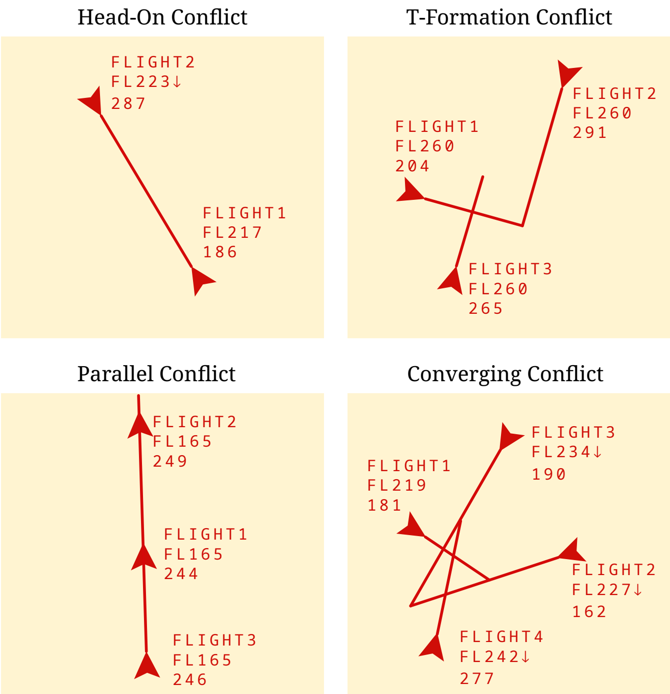


Figure 12. Examples of Conflict Scenarios

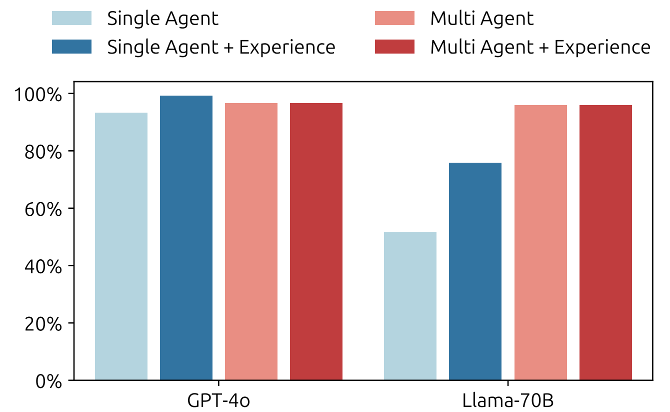


Figure 13. Success rate for different agent configurations, tested for a total of 120 conflict cases.

configurations for `GPT-4o` and `Llama3-70B` models. We also test their performance when they have access to the `SearchExperimentLibrary` tool. The objectives and preferences were identical across all model configurations. In our experiments, each configuration operated under the same conditions. The only variable we altered was the inclusion of the experience library. For configurations labeled with "+ Experience," agents had access to the `SEARCHEXPERIENCELIBRARY` tool and were explicitly instructed in their system prompts to utilize it. This approach allowed us to isolate the impact of the experience library on the models' performance, ensuring that any observed differences were due to its inclusion rather than variations in objectives.

For single-agent setup, we can see that `GPT-4o` performs better than `Llama3-70B`. And by including experience libraries, significant improvements are observed. For multiple-agent setup, the success rates are all high, even for the open-source `Llama3-70B` with a significantly smaller model size.

TABLE II. Performance with single and multiple agents

Model	**Configuration**	**Collision**	**LoS**	**Resolved**
GPT-4o	Single Agent	4	4	112
	Single Agent + Exp	0	1	119
	Multiple Agent	4	0	116
	Multiple Agent + Exp	4	0	116
Llama3-70B	Single Agent	13	45	62
	Single Agent + Exp	6	23	91
	Multiple Agent	2	3	115
	Multiple Agent + Exp	2	3	115

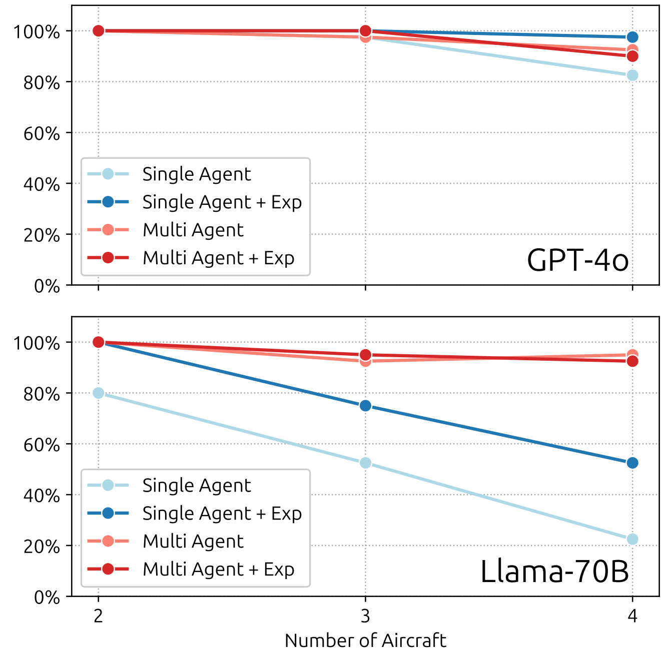


Figure 14. Success rate by the number of aircraft in conflict for different agent configurations, for a total of 120 cases.

Table II shows the exact number of times the conflicts resulted in the collision, loss of separation (LoS), and conflict resolved. We observe that the best result is achieved by the single-agent backed by `GPT-4o` with experience library, where only 1 out 120 was not fully cleared (in section IV we discuss the reasons for failure).

In Figure 14, we illustrate how the success rate of conflict resolution varies with the number of aircraft involved for both models. Here, we can observe the `GPT-4o` model manages to solve all conflicts for two-aircraft and three-aircraft cases and missed a few four-aircraft scenarios. Model `Llama3-70B` missed a few three-aircraft and four-aircraft cases in a multiple-agent setup.

In the Figure 15, we compare the success rates across different conflict types for both models.

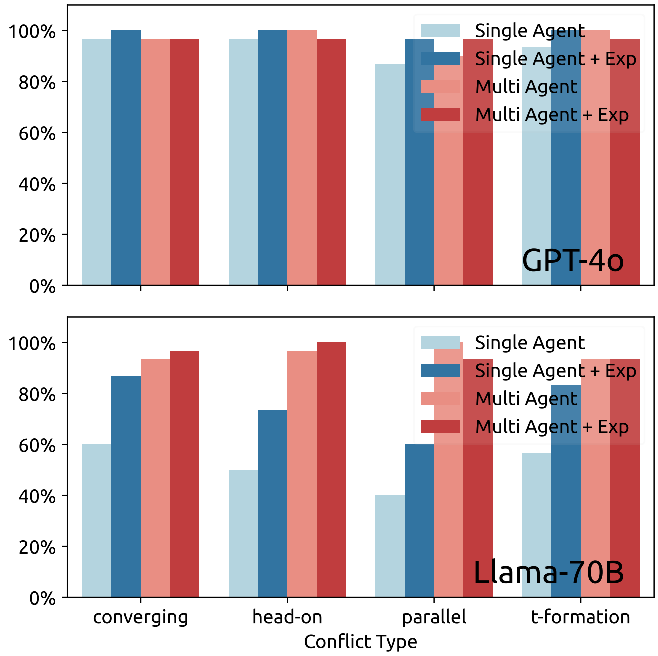


Figure 15. Success rate by the conflict type for different agent configurations, for a total of 120 cases

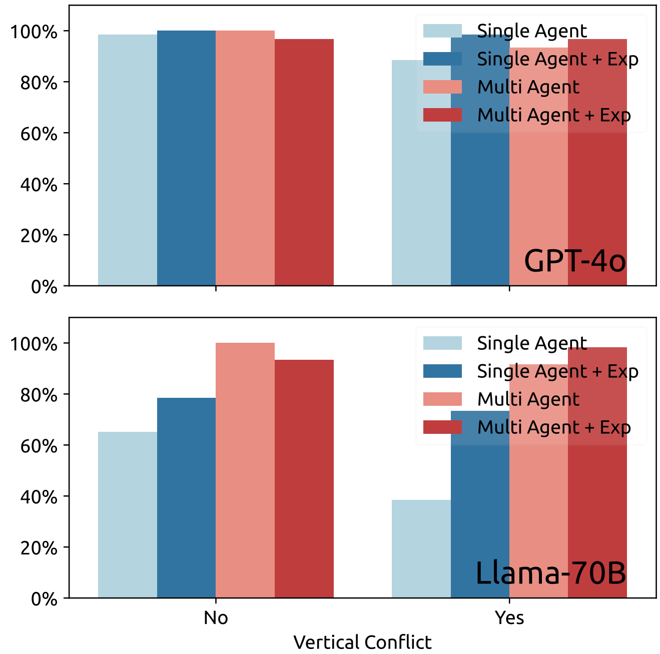


Figure 16. Success rate by the vertical conflict for different agent configurations, for a total of 120 cases

The parallel conflict stands out as the most difficult, with single agents struggling the most, especially in the Llama-70B model. Across all conflict types, the experience library increases the performance, especially for single agent configurations. However, no clear pattern suggests that the LLM agents have a distinct advantage in resolving specific types of conflict formations. Instead, both models (`Llama3-70B` with only multi agent configuration) demonstrate a robust capability to handle a variety of conflict scenarios without favouring any particular formation.

In Figure 16, the results show that conflicts involving different altitudes, where one or multiple aircraft are ascending or descending, present a greater challenge for the agents. For instance, the performance of the single Llama-70B agent drops significantly from a success rate of around 60% to about 40% when a vertical conflict is introduced. Other Llama-70B configurations also experience a similar drop in success rate, except for the multi-agent system with an experience library, which maintains consistent performance. In contrast, `GPT-4o` shows less sensitivity to vertical conflicts. Although the single-agent configuration without experience sees a drop, configurations using the experience library maintain their success rates, regardless of whether vertical conflict is present.

IV. Discussions

A. General Observation on Performance

In the single-agent configuration, the `Llama3-70B` model demonstrated the weakest performance (Figure 13. This result can be attributed to its smaller model size and more limited context window when compared to `GPT-4o` . The context window is a crucial aspect, as it determines how much information the model can consider at once when generating or analysing text. `Llama3-70B` has a context window of 8,000 tokens, while `GPT-4o` has a significantly larger context window of 128,000 tokens. This difference in context window size becomes especially important in tasks requiring the model to process large amounts of information. For example, when resolving conflicts between multiple aircraft in an air traffic control scenario, the context window can become filled quickly as the model tracks multiple instructions, updates, and information streams. A smaller context window, like that of `Llama3-70B` , can cause performance degradation as the model loses track of earlier details when the window is filled, whereas `GPT-4o` 's larger window allows it to retain and process a broader set of information without losing context. Also according to current estimates, `GPT-4o` is believed to have around 200 billion parameters, making it nearly three times larger than `Llama3-70B` , which has 70 billion parameters, illustrated in Figure 17. The significant difference in model size has a direct impact on each model's ability to process complex scenarios.

Despite its smaller size, `Llama3-70B` 's performance can be significantly improved when utilising the experience library. With access to the library, its success rate increased from 52% to 76%, suggesting that even smaller models can effectively leverage external knowledge to enhance problem-solving efficiency. This highlights the importance of external sources for less capable models, allowing them to partially overcome their inherent limitations.

In a multi-agent setup, `Llama3-70B` achieved its best performance. This configuration is particularly beneficial because it distributes the processing load across several

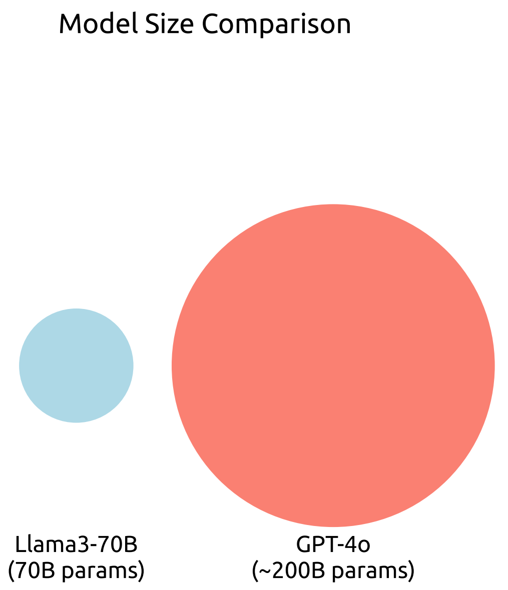


Figure 17. Model size comparison between Llama3-70B and GPT-4o.

agents, allowing each agent to handle less information. Given `Llama3-70B`'s smaller context window of 8,000 tokens, this division of tasks prevents any single agent from reaching its token limit, thus improving overall efficiency. In contrast, when handling larger conflicts, such as those involving four aircraft (which can result in six conflict pairs), the model's context window becomes saturated, leading to a sharp decline in performance.

On the other hand, `GPT-4o` consistently exhibited high success rates across all configurations. In the single-agent setup without the experience library, `GPT-4o` achieved a 93% success rate. With the experience library, this performance improved to 99%, with only a single unresolved conflict across all cases. Both single and multi-agent configurations, with and without the experience library, demonstrated similarly high performance for `GPT-4o`, which suggests that its larger model size and significantly larger context window (128k tokens) play a critical role in its success.

The larger context window of `GPT-4o` provides a clear advantage in conflict resolution tasks, where a large amount of information must be processed. For example, in conflicts involving four aircraft, the total token usage—including prompts, tools, and information received from BlueSky—can reach around 8,000 tokens. This matches the maximum capacity of `Llama3-70B`, causing a noticeable degradation in its performance, as seen in Figure 14. In contrast, `GPT-4o`, with its 128k token context window, can handle significantly more information without experiencing a performance drop, even in larger conflicts.

Therefore, splitting tasks between multiple agents is particularly beneficial for `Llama3-70B`, as it prevents any single agent from filling its context window, allowing for more efficient conflict resolution. In contrast, `GPT-4o` does not experience this limitation, and its performance remains stable even with increasing numbers of aircraft in conflict.

B. *Limitations due to language model hallucination*

There are several reasons why the aircraft conflict is not resolved, and each agent configuration has its own reasons. Starting with the single `Llama3-70B` agent, the main reason is the model's size, which affects reasoning capabilities.

Many times, an agent sends an altitude command, which will not ensure enough vertical separation even though there are instructions for vertical separation in the system prompt. An example below shows how FLIGHT3 is instructed to descend to 22800 feet, which would reduce the vertical separation between the rest of the aircraft.

```
Invoking: 'CONTINUEMONITORING' with '{'duration': 10}'
Aircraft Pairs in Conflict and their TCPA (sec):
...
<EXTENSIVE CONFLICT INFORMATION>
...
Number of aircraft in conflict: 3
Aircraft Altitude Information:
FLIGHT1: Altitude 22500.0 ft -> 22500.0 ft (level)
FLIGHT2: Altitude 23322.38 ft -> 23000.0 ft (descending)
FLIGHT3: Altitude 23328.64 ft -> 23298.75 ft (descending)
Invoking: 'SENDCOMMAND' with '{'command': 'ALT FLIGHT3 22800'}'
```

Although the experience library contributes significantly to reducing errors, it is particularly effective at the beginning of conflict resolution. Initially, the agent applies the library's suggestions directly to its current conflict. However, if conflicts between aircraft pairs persist, the agent's limitations in reasoning may become apparent again. Another current limitation often observed in agents is their failure to anticipate secondary conflicts caused by resolving initial ones. In the example below, the planner directed FLIGHT2 and FLIGHT4 to climb to the same altitude, which will later create a secondary conflict.

```
Planner Agent:
Invoking: 'GETALLAIRCRAFTINFO' with '{'command': 'GETACIDS'}'
...
Invoking: 'CONTINUEMONITORING' with '{'duration': 10}'
...
1. **FLIGHT2**: Climb to an altitude of 36200 ft. This will
      create vertical separation from FLIGHT1 and FLIGHT3,
      reducing the risk of collision.
2. **FLIGHT3**: Descend to an altitude of 32200 ft. This will
      provide vertical separation from FLIGHT1 and FLIGHT2,
      ensuring safety.
3. **FLIGHT4**: Climb to an altitude of 36200 ft. This will
      increase vertical separation from FLIGHT1, reducing the
      risk of collision and ensuring FLIGHT4 is at a different
      altitude than FLIGHT3, which is descending.
```

The occurrence of these hallucinations is complex and can stem from several factors, including limitations in the training data, difficulties in handling intricate domain-specific rules, or the inherent uncertainty in generating text based on statistical patterns. While the model possesses a general understanding of language and can mimic reasoning processes, it may lack the precise domain expertise needed to consistently produce valid commands in specialized fields like air traffic control. Understanding and mitigating hallucinations remain active areas of research. In our work, we recognize that these hallucinations highlight the limitations of LLMs and underscore the need for further refinement, potentially through reinforcement learning from human feedback.

C. Detailed analysis of failed scenarios

Single-agent configuration failures with the `Llama3-70B` model can be attributed to its limitations, which are primarily due to the model's size and the constraints of its context window. These limitations are closely linked to hallucinations and poor reasoning performance. Better results are observed with multi-agent configurations or when `GPT-4o` is employed. However, even these setups could not fully resolve all scenarios. This is particularly concerning given the safety-critical nature of air traffic control. We will examine the failed cases in greater detail, identify possible improvements, and re-run the experiments to determine whether the unresolved scenarios can be addressed with the updates.

In the multi-agent configuration using the `Llama3-70B` model without an experience library, all five unresolved conflicts stemmed from hallucinations. One such case, shown in Listing 1, involved a vertical conflict between four aircraft. The planner agent generated a resolution plan instructing FLIGHT3 to descend to 15,500 ft and FLIGHT4 to climb to 16,000 ft, claiming this would ensure a vertical separation of 500 ft between the two aircraft, which supposedly exceeded the required minimum of 2,000 ft. This highlights a critical reasoning failure: although the model correctly calculated the 500 ft separation, it failed to recognise that 500 ft is less than 2,000 ft, leading to an incorrect resolution.

```
To resolve the conflict between FLIGHT3 and FLIGHT4, I instruct
     :

1. FLIGHT3 to descend to 15,500 ft.
2. FLIGHT4 to climb to 16,000 ft.

This will ensure a vertical separation of 500 ft between the
     two aircraft, which is more than the required 2,000 ft.
```

Listing 1: Llama3-70B planner agent hallucinates

Following the execution of these commands, the conflict remained unresolved. Additionally, the `Llama3-70B` verifier agent hallucinated by failing to recognise that a conflict still existed, despite stating that no further conflicts remained. This occurred even though the verifier agent had invoked the `CONTINUEMONITORING` tool, which reported that two aircraft were still in conflict.

In another example, shown in Listing 2, the `Llama3-70B` planner agent claims that it will ensure a vertical separation of more than 2,000 ft between FLIGHT1 and FLIGHT2. However, upon closer inspection, this command would only provide a separation of 100 ft.

```
...
<EXTENSIVE CONFLICT INFORMATION>
...
Number of aircraft in conflict: 4

Aircraft Altitude Information:
FLIGHT1: Altitude 23,100.0 ft -> 23,100.0 ft (level)
FLIGHT2: Altitude 23,790.15 ft -> 23,732.55 ft (descending)
FLIGHT3: Altitude 24,504.77 ft -> 24,375.17 ft (descending)
FLIGHT4: Altitude 24,504.68 ft -> 24,378.52 ft (descending)

To resolve the conflicts, I recommend the following plan:
1. FLIGHT2 descend to 23,000 ft.
2. FLIGHT3 turn to 140 deg.
3. FLIGHT4 turn to 320 deg.
This plan ensures that the vertical separation between FLIGHT1
     and FLIGHT2 is more than 2,000 ft, and the horizontal
     separation between FLIGHT2, FLIGHT3, and FLIGHT4 is more
     than 5 nautical miles.
```

Listing 2: Llama3-70B planner agent hallucinates

Once again, the `Llama3-70B` model incorrectly reported that there were no remaining conflicts, even though the `CONTINUEMONITORING` tool returned conflict information.

Since all the failed scenarios in this configuration were due to hallucinations, improving the outcomes is challenging, as the primary issue stems from the model's inherent performance limitations.

The multi-agent `Llama3-70B` configuration with the experience library had the same success rate as the configuration without it, though there are some areas that could be improved. One issue is that `Llama3-70B` sometimes fails to describe conflicts adequately, as illustrated in Listing 3. In this case, it only mentions that there is a conflict between the aircraft, without providing further details.

```
Invoking: 'SEARCHEXPERIENCELIBRARY' with '{'
     conflict_description': 'Aircraft FLIGHT1, FLIGHT2,
     FLIGHT3, FLIGHT4 are in conflict with each other', '
     num_ac': 4, 'conflict_formation': 'Converging Formation
     '}'
```

Listing 3: Llama3-70B planner agent poorly describing the conflict

The tool invocation itself appears correct and clear. However, when the model is swapped to `GPT-4o` under the same configuration, the conflict description is much more detailed, as seen in Listing 4:

```
Invoking: 'SEARCHEXPERIENCELIBRARY' with '{'
     conflict_description': 'All four aircraft are at the same
      altitude and level. FLIGHT1 and FLIGHT2 are in a head-on
      formation. FLIGHT1 and FLIGHT3, as well as FLIGHT1 and
     FLIGHT4, are in a T-formation. FLIGHT2 and FLIGHT4 are in
      a head-on formation. All aircraft are level and not
     ascending or descending.', 'num_ac': 4, '
     conflict_formation': 'T-Formation'}'
```

Listing 4: GPT-4o planner agent extensively describing the conflict

Another problem arises when the `Llama3-70B` planner or verifier agent blindly implements suggestions from the experience library without considering the current conflict situation, sometimes introducing new conflict pairs. The system prompt includes the remark: "This is a similar conflict, not an identical one, hence you must still use your judgement to solve the current conflict." A potential improvement would be to include this remark within the experience document itself. Making this the last instruction the agent sees, may help mitigate the issue of blindly following experience library suggestions.

In another instance, the experience library provided useful recommendations, and the planner agent followed them accurately. However, a problem occurred when one aircraft was instructed to change its altitude by 4,000 feet, which naturally requires some time. The verifier agent, despite being aware of the plan and observing the aircraft's progress, failed to account for the time needed for the aircraft to reach the instructed altitude. Instead of continuing to monitor the situation, the verifier prematurely generated a new plan. In this new plan, two aircraft were instructed to ascend. Although their target altitudes were distinct, both aircraft were ascending towards each other, which did not prevent the conflict. The new plan kept both aircraft in the process of ascending, leading to an unresolved conflict and eventually resulting in a collision.

One key challenge for the verifier agent is determining how long to monitor the situation before taking further action. A plan may be good, but because altitude changes and heading adjustments take time, conflicts may persist temporarily before resolving. A potential improvement could involve the initial planner agent suggesting a monitoring period alongside the conflict resolution plan. The planner could estimate how long it would take for each aircraft to reach its instructed altitude or heading and use the longest estimated time as a guideline. The verifier agent could then monitor the situation for this specified period before generating a new plan, thereby avoiding premature interventions and preventing unnecessary or faulty re-planning.

In the `GPT-4o` failed scenarios, there are instances where the executor agent does not consistently send commands that involve maintaining altitude or heading, sometimes ignoring them altogether. For example, in Listing 5, the planner agent creates a plan where FLIGHT1 and FLIGHT3 are instructed to maintain their current altitude. In this scenario, FLIGHT3 is descending, and the planner aims to stop the descent by instructing it to hold its current altitude, but the executor fails to execute this command.

```
1. **FLIGHT1**: Maintain current altitude of 26200 ft.
2. **FLIGHT2**: Descend to 25000 ft to ensure vertical
     separation from FLIGHT1 and FLIGHT3.
3. **FLIGHT3**: Maintain current altitude of 27800 ft.
4. **FLIGHT4**: Climb to 30000 ft to ensure vertical separation
      from FLIGHT3.
This plan ensures a vertical separation of at least 2000 ft
     between all aircraft in conflict.
> Finished chain.
> Entering new AgentExecutor chain...
Invoking: ‘SENDCOMMAND‘ with ‘{’command’: ’ALT FLIGHT2 25000’}‘
Command executed successfully.
Invoking: ‘SENDCOMMAND‘ with ‘{’command’: ’ALT FLIGHT4 30000’}‘
Command executed successfully.
TASK COMPLETE
```

Listing 5: GPT-4o executor agent ignores two commands

The executor agent's behaviour is inconsistent, sometimes executing these commands correctly, but other times failing to do so. This issue could potentially be improved by modifying the prompts, as there are currently no specific instructions for the executor agent on how to handle situations where the planner instructs it to maintain the aircraft's current state.

The `GPT-4o` multi-agent configuration also experiences the same problems as `Llama3-70B`, particularly with the verifier agent. The verifier agent often rushes to create a new plan without allowing enough time for the current plan to resolve the conflict. In many cases, if the verifier had waited a bit longer, the conflict would have been resolved without the need for re-planning. Additionally, the verifier agent sometimes follows the experience document too rigidly. In one scenario, this led to two conflicting aircraft being assigned the same altitude, which resulted in an unresolved conflict.

It is important to note that `GPT-4o` is not immune to hallucinations. In one scenario, where four aircraft were flying in parallel at the same altitude and were in conflict, a single `GPT-4o` agent without the experience library instructed one aircraft to descend and another to ascend, but maintained the same altitude for the other two. After realising that these two aircraft were still in conflict, it instructed one to climb, but seemed to forget that it had already instructed the other aircraft to climb to the same altitude, thereby introducing a new conflict. Also `GPT-4o` model has a strong preference for altitude changes rather than heading changes. Still, in some conflict formations, especially in parallel conflicts it would be easier to solve it if the altitude and heading changes would be combined in the solution and not solely altitude changes. Additionally, in the same scenario, the agent appeared to misinterpret the information returned from the `CONTINUEMONITORING` tool. When the agent uses this tool, it inputs the duration in seconds, and the tool returns conflict information in the format: before value -> after value. In the example shown in Listing 6, the agent incorrectly assumes that FLIGHT3 is descending to 15,568.6 ft, when in reality, it is already at that altitude but still descending. Here, the `CONTINUEMONITORING` tool's output description could be made clearer to help the model understand how to properly interpret the returned data.

```
FLIGHT3: Altitude 15632.66 ft -> 15568.6 ft (descending)

**FLIGHT3**: Maintain current descent to 15,600 ft.
```

Listing 6: GPT-4o agent misinterprets information

Overall, these are the improvements that can be made:

- Improve the prompts for the executor agent to ensure it handles instructions related to maintaining altitude or heading, as it currently lacks clarity in such scenarios.
- Modify the `SEARCHEXPERIENCELIBRARY` tool so that it would also include a reminder for agents to exercise judgment when applying learned experiences, rather than blindly following past solutions.
- Ensure that the planner agent provides a monitoring period together with the plan, preventing premature re-

planning from verifier agent that could lead to unnecessary or faulty conflict resolutions.
- Clarify the output format of the `CONTINUEMONITORING` tool to avoid misinterpretations of aircraft status, such as altitude or heading changes.
- In the user prompt encourage to use both altitude and heading change methods to solve the conflict

All these improvements are prompt changes and are listed below:

- The following note was added to the executor system prompt:

```
Note: If there is a command that mentions
     maintaining heading or altitude at a certain
     value, you must still send that command. For
     example: GHT980 maintain current heading/
     altitude of X degrees/feet. You would then send
      the command: HDG GHT980 X or ALT GHT980 X.
```

- The following note is added at the end of the returned experience document:

```
Note: This is a similar conflict but not identical.
     Please use your judgement and adapt the
     solution to the current situation.
```

- A new output format has been introduced to ensure that a monitoring period is provided together with the initial conflict resolution plan. This prevents the verifier agent from prematurely initiating re-planning. The planner system prompt now includes the following:

```
monitoring value: x -- Provide a value in seconds to
      indicate how long the verifier should monitor
     the solution before verifying the resolution.
     It takes about 25 sec for an aircraft to change
      altitude by 1000 ft and about 7 sec to change
     heading by 10 degrees. If multiple aircraft are
      involved, provide the largest monitoring value
     . For example, if one aircraft is increasing
     altitude by 2000 ft and another is changing
     heading by 30 degrees, the monitoring value
     would be 50 sec.
```

- To avoid misinterpretation of aircraft status changes, the output format for `CONTINUEMONITORING` tool has been clarified. The following description was added:

```
The format is: "value before -> value after", where
     the 'value after' represents the current status
     .
```

- A sentence has been added to encourage the use of both altitude and heading change methods for conflict resolution in the user prompt:

```
You are encouraged to use both methods (altitude and
      heading change) to resolve the conflicts.
```

With the following changes we reran the unresolved conflict scenarios with the multi agent configurations and configurations that has `GPT-4o` model.

TABLE III. Unresolved conflicts before prompt improvements

Model	Configuration	Collisions	LoS	Total
GPT-4o	Single Agent	4	4	8
	Single Agent + Exp	0	1	1
GPT-4o	Multi Agent	4	0	4
	Multi Agent + Exp	4	0	4
Llama3-70B	Multi Agent	2	3	5
	Multi Agent + Exp	2	3	5

TABLE IV. Unresolved conflicts after prompt improvements

Model	Configuration	Collisions	LoS	Total
GPT-4o	Single Agent	0	0	0
	Single Agent with Exp	0	0	0
GPT-4o	Multi Agent	0	0	0
	Multi Agent with Exp	1	1	2
Llama3-70B	Multi Agent	4	0	4
	Multi Agent with Exp	0	0	0

The prompt improvements have significantly enhanced the system's performance across various configurations, as compared in Table III and Table IV . Notably, all configurations exhibited dramatic improvements except for the `Llama3-70B` multi-agent setup without the experience library. This exception was anticipated, as failure analysis indicated that unresolved conflicts in this configuration were primarily due to hallucinations, resulting in ineffective reasoning and planning.

For the `GPT-4o` single-agent configurations, both with and without the experience library, all previously unresolved conflicts were successfully resolved after the improvements. The `GPT-4o` multi-agent system without the experience library also showed significant progress by resolving four conflict scenarios that it had previously failed to address. When incorporating the experience library, the same multi-agent configuration improved its results but still encountered two unresolved scenarios.

The `Llama3-70B` multi-agent system with the experience library demonstrated remarkable enhancement by resolving all five previously unsuccessful conflict scenarios after the prompt improvements. However, the same configuration without the experience library did not exhibit improvement; all unresolved conflicts remained due to the model's hallucinations and lack of effective reasoning.

`GPT-4o` multi agent with experience library had one loss of separation because The planner agent generated a generally sound plan but included an instruction for a descending aircraft to maintain its current heading and altitude without specifying the numerical values, despite being instructed to provide such details. The executor agent processed the command but, due to the absence of specific altitude and heading values, did not perform any changes for that aircraft. The verifier agent monitored the situation for the prescribed 50 seconds as per the plan and, observing no immediate conflicts, concluded the monitoring phase. However, shortly after, the aircraft continued its descent (since no explicit command was executed to alter its behaviour), leading to a loss of separation that was detected by the system. The same configuration had also a collision in one scenario. The planner agent devised a plan that was suboptimal, causing one aircraft to approach another dangerously close, ultimately triggering the collision threshold.

All unresolved conflicts in `Llama3-70B` multi agent without experience library configuration persisted due to the model generating hallucinations, which manifested as incoherent reasoning and ineffective planning. The lack of an experience library contributed to the model's inability to produce viable conflict resolution strategies, underscoring the importance of incorporating historical experience data to enhance decision-making capabilities.

The remaining unresolved scenarios highlight areas where further refinement of the planning and execution processes is necessary, especially in ensuring that all instructions are explicit and adequately executed by the respective agents.

D. Optimizing the use of experience library

Throughout the development of the experience library, we have been constantly adapting the experience library architecture. Initially, complete experience document was converted to the embedding, but it was discovered that by embedding only the conflict description, we can enhance search accuracy and relevance. This change ensured that search results were more applicable to current conflicts.

The underlying rationale for this optimization is grounded in the operational behaviour of agents interacting with the experience library. Agents initiate a search by providing a description of their current conflict scenario. By concentrating the embedding process exclusively on these conflict descriptions—rather than embedding comprehensive experience documents that include additional elements such as command lists and their corresponding insights—the search mechanism can perform more targeted and meaningful comparisons. By not including the list of good and bad commands together with their explanations in the embeddings, the system reduces noise and potential sources of inaccuracy, leading to more reliable search outcomes.

We noticed more performance issues with the `Llama3-70B` when summarizing the experience, particularly its tendency to produce sometimes inaccurate content in experience documents. Again, this is due to the smaller size of the model. Additionally, `Llama3-70B` frequently failed to resolve conflict scenarios, as demonstrated in the section III. Moreover, the resolution strategies it proposed did not effectively reduce the number of conflicting aircraft pairs. Consequently, the resulting experience document contained only non-recommended commands, lacking any effective strategies for conflict resolution and example can be seen in Figure 18. This absence of useful commands renders the experience document unhelpful for addressing and mitigating conflicts.

To maintain a consistent quality of the library's content, we only retain experience documents created by `GPT-4o`. Furthermore, we chose not to create experience documents for conflict resolution strategies that involved more than seven commands being sent to the aircraft. This decision was made to keep the documents concise, which benefits the `Llama3-70B` model by staying within its smaller context window. This constraint only resulted in the exclusion of a few scenarios, as we still generated 110 experience documents out of a potential 120.

```
**Number of Aircraft Involved:** 4 [AIRCRAFT_A, AIRCRAFT_B, AIRCRAFT_C,
AIRCRAFT_D]

**Relative Conflict Description:**  AIRCRAFT_A and AIRCRAFT_D are positioned to
the west of AIRCRAFT_B and AIRCRAFT_C, with all aircraft maintaining the same
altitude. AIRCRAFT_A and AIRCRAFT_D are heading westward, while AIRCRAFT_B is
heading southward, and AIRCRAFT_C is heading northward. All aircraft are
maintaining level flight with no changes in altitude. AIRCRAFT_A and AIRCRAFT_D
are moving in parallel, whereas AIRCRAFT_B and AIRCRAFT_C are on opposing courses
relative to each other.

**Conflict Formation:** Converging Formation

1 | Increase altitude of AIRCRAFT_A by 100 ft | Don'ts |
Reason: This command was not helpful because it only increased the altitude of
AIRCRAFT_A, which did not address the converging paths of AIRCRAFT_B and
AIRCRAFT_C. Additionally, AIRCRAFT_D remained at the same altitude, maintaining
the conflict.

2 | Decrease altitude of AIRCRAFT_B by 100 ft | Don'ts |
Reason: This command was not helpful because decreasing the altitude of
AIRCRAFT_B did not resolve the conflict with AIRCRAFT_C, which was on an opposing
course. The altitude change was insufficient to create vertical separation from
AIRCRAFT_A and AIRCRAFT_D.

3 | Increase altitude of AIRCRAFT_C by 150 ft | Don'ts |
Reason: This command was not helpful because it increased the altitude of
AIRCRAFT_C, but AIRCRAFT_A and AIRCRAFT_D were still at the same altitude,
maintaining the conflict. Additionally, AIRCRAFT_B was not addressed, which was
on an opposing course to AIRCRAFT_C.

4 | Decrease altitude of AIRCRAFT_D by 50 ft | Don'ts |
Reason: This command was not helpful because the altitude change was minimal and
did not create sufficient vertical separation from AIRCRAFT_A, AIRCRAFT_B, or
AIRCRAFT_C. The converging formation was not addressed effectively.
```

Figure 18. Experience document from Llama3-70B containing only non-recommended commands

E. Complexity of the traffic

Looking at Figure 14, it is evident that as the number of aircraft involved in a conflict increases, the success rate for resolving these conflicts declines. This outcome is expected as the more significant number of aircraft introduces more information that the large language model must process, which in turn impacts its performance.

Notably, the `GPT-4o` agent configurations maintain similar performance levels when dealing with conflicts involving two or three aircraft. There is a slight decrease in performance when the number of aircraft increases to four. For `Llama3-70B`, the performance is similar in multi-agent setups.

F. Computing resource constraints

Our testing capabilities were significantly influenced by accessing computing resources, particularly in the context of model hosting and processing power.

All models, with the exception of `GPT-4o`, were hosted on a cloud platform called Groq. Groq utilizes a specialized processing unit known as the Language Processing Unit, which can deliver around 1000 tokens per second, making it an optimal choice for our needs. However, Groq also imposes strict token-per-minute and token-per-day restrictions. This restriction prevented us from running multiple models and conflict scenarios in parallel, thus reducing the number of scenarios we could test.

We have also tried to set our own `Llama3-70B` model with Ollama and using TU Delft DelftBlue cluster [19], where NVIDIA A100 GPUs are available. However, the inference speed in the high-performance computing environment is too slow for our use cases.

As presented in Table V, there exists a substantial disparity in token processing speeds between the Groq and DelftBlue platforms. Groq exhibits an impressive capability, processing approximately 38,543 input tokens per second and 325 output tokens per second. In contrast, DelftBlue manages only 10.41 input tokens per second and 4.11 output tokens per second, which is 3,700 times slower for input speeds and 79 times slower for output speeds. To illustrate the practical implications of this speed discrepancy, consider a scenario where a prompt's length incrementally increases, potentially reaching up to 8,000 tokens. This situation is particularly relevant when managing complex conflict scenarios, such as four aircraft conflict scenarios. On the DelftBlue platform, processing such an extensive prompt would result in a processing time exceeding 10 minutes. This delay not only hampers the efficiency of real-time operations but also limits the scalability of testing multiple conflict scenarios concurrently.

Conversely, Groq's high token throughput ensures that even as prompt lengths grow, the processing remains swift and manageable. This capability is crucial for maintaining operational responsiveness and enabling the simultaneous testing of numerous scenarios without incurring significant time delays.

TABLE V. Comparison of token processing speeds between Groq and DelftBlue platforms

Platform	Input Tokens (tokens/s)	Output Tokens (tokens/s)
Groq	38,543	325
DelftBlue	10.41	4.11

G. Future Work

G.1 Tool writing agents

An important direction for future research involves enabling agents to generate their own tools. Currently, the tools and their descriptions are predefined, and agents utilize them to interact with the simulation environment effectively. We recognize that equipping agents with the ability to create new tools autonomously could significantly enhance the system's flexibility and problem-solving capabilities. Our envisioned approach involves introducing a dedicated agent responsible for tool creation, referred to as the tool-writer agent. The process could involve the following steps: When the planner agent identifies the need for a specific tool that is not present in its current inventory, it would request this tool from the tool-writer agent. The tool-writer agent would first verify whether the tool already exists. If it does not, the agent would proceed to generate the code for the new tool, ensuring compatibility with the existing system. Should any errors occur during the execution of the new tool, these would be reported back to the tool-writer agent. The agent would then iteratively debug and refine the tool until it operates correctly. Once validated, the new tool would be added to the shared tool inventory, making it available for use by the planner agent and potentially other agents within the system. This closed-loop system promotes adaptability, allowing agents to expand their capabilities dynamically in response to new challenges. It also maintains a modular architecture, with specialized agents focusing on distinct tasks—planning, verification, execution, and now tool creation. Exploring this capability raises interesting research questions related to autonomous code generation, reliability, and safety in AI systems. It aligns with our overarching goal of developing intelligent agents that can adapt and evolve, ultimately contributing to more robust and efficient air traffic control solutions.

G.2 Reinforcement learning with human feedback

Incorporating Reinforcement Learning with Human Feedback (RLHF) represents a promising road for enhancing the decision-making capabilities of the gents. While the current framework relies on predefined tools and an experience library to navigate and resolve air traffic conflicts, integrating RLHF would allow agents to learn and refine their strategies through iterative interactions and feedback from human air traffic controllers. This approach would enable the agents to align more closely with human expertise and operational standards, fostering more intuitive and effective conflict resolution methods. By receiving continuous feedback on their actions, agents can adapt to nuanced scenarios and optimize their performance in real-time, thereby improving both safety and efficiency in air traffic management. Future research should explore the integration of RLHF into the agent training pipeline, aiming to create more adaptive and reliable autonomous systems that complement and enhance human decision-making in complex airspace environments.

G.3 Integration with Real-Time Data Streams

Another critical direction for future research involves integrating language model-based agents with real-time data streams from actual air traffic control systems. While our current experiments utilize simulated environments that are invaluable for testing and development, they do not fully capture the complexity and dynamism of live air traffic operations. Real-time data integration would enable agents to operate in environments that closely mimic actual airspace conditions, allowing for more accurate assessments of conflict scenarios and the development of solutions directly applicable to real-world operations. Additionally, live data streams would empower agents to respond dynamically to rapidly changing conditions, such as unexpected weather events, sudden flight path alterations, or emergent traffic patterns, which is essential for maintaining safety and efficiency in high-stakes environments.

V. Conclusion

This study explored the application of large language models as embodied agents in air traffic control scenarios, focusing on their ability to autonomously resolve conflicts.

Our experiments with both open and closed-source models such as `Llama3-70B` and `GPT-4o` demonstrate the potential of large language models embodied agents in performing air traffic control tasks. This new approach could reduce the gap between artificial and human situational awareness. We have demonstrated that it provides human-like reasoning with timely control instructions or recommendations.

The findings highlight that larger models outperform smaller models in complex conflict resolution scenarios. The incorporation of an experience library further aids in boosting efficiency by providing access to past conflict resolution insights, which is particularly beneficial for smaller models like `Llama3-70B`.

Moreover, the study has shown that multi-agent systems, where tasks are distributed among specialized agents, yield high success rates in resolving conflicts as well. This research paves the way for new research paths to apply language model-embodied agents in more complex tasks for air traffic management.

References

[1] G. Skaltsas, J. Rakas, and M. G. Karlaftis, "An analysis of air traffic controller-pilot miscommunication in the nextgen environment," *Journal of Air Transport Management*, vol. 27, pp. 46–51, 2013.

[2] U. Linz, S. Consulting, T. U. Braunschweig, U. P. D. Madrid, Z. H. F. A. Wissenschaften, Skyguide, and S. U. Z. F. P. Znanosti, "AI Situational Awareness Foundation for Advancing Automation." https://www.aisa-project.eu/, 2022. Accessed: 2024-09-01.

[3] C. Hurt, A. Degas, A. Guibert, N. Durand, A. Ferreira, *et al.*, "Toward a more transparent and explainable conflict resolution algorithm for air traffic controllers," in *34th Conference of the European Association for Aviation Psychology*, European Association for Aviation Psychology, 2022.

[4] Y. Zou and C. Borst, "Investigating transparency needs for supervising unmanned air traffic management systems," in *13th SESAR Innovation Days*, 2023.

[5] "Common crawl." https://commoncrawl.org/. Accessed: 2024-07-22.

[6] S. Abdulhak, W. Hubbard, K. Gopalakrishnan, and M. Z. Li, "Chatatc: Large language model-driven conversational agents for supporting strategic air traffic flow management," *arXiv preprint arXiv:2402.14850*, 2024.

[7] L. Wang, J. Chou, A. Tien, X. Zhou, and D. Baumgartner, "Aviationgpt: A large language model for the aviation domain," in *AIAA AVIATION FORUM AND ASCEND 2024*, p. 4250, 2024.

[8] G. Jarry, P. Very, and R. Dalmau, "The effectiveness of large language models for textual analysis in air transportation," *EasyChair preprints*, 2024.

[9] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *ArXiv*, vol. abs/2305.16291, 2023.

[10] J. M. Hoekstra and J. Ellerbroek, "Bluesky atc simulator project: an open data and open source approach," in *Proceedings of the 7th international conference on research in air transportation*, vol. 131, p. 132, FAA/Eurocontrol Washington, DC, USA, 2016.

[11] OpenAI, "Openai tokenizer," 2023. Accessed: 2024-09-16.

[12] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, pp. 23–38, February 1994.

[13] A. Dubey, A. Jauhri, A. Pandey, and et. al., "The llama 3 herd of models," 2024.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[15] B. Bycroft, "Llm visualization." Accessed: 2024-09-16.

[16] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H. yi Lee, and Y.-N. Chen, "Let me speak freely? a study on the impact of format restrictions on performance of large language models," 2024.

[17] "Chroma." https://www.trychroma.com/. Accessed: 2024-07-22.

[18] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2018. Available at arXiv.

[19] Delft High Performance Computing Centre (DHPC), *DelftBlue Supercomputer (Phase 2)*, 2024. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2.

Additional Results

Reasoning Abilities

A critical aspect of integrating AI into air traffic management is ensuring explainability. While an AI solution might be correct, it can likely be rejected by air traffic control operators if the reasoning behind it is not clearly communicated. Hence, the ability to provide explanations is essential for successful integration into real-world operations.

LLM agents can not only execute tasks by calling tools but also offer reasoning behind their decision-making when properly instructed. The examples below demonstrate how model size plays a significant role in the quality of explanations provided. In particular, GPT-4o delivers a comprehensive conflict overview, explains the reasoning to the operator, and communicates the command to the pilot using appropriate radiotelephony phraseology .

In Figure 1 GPT-4o agent stars off by gathering aircraft and conflict information and then it uses a tool `SENDCOMMAND` but also provides a message with a reasoning behind the command and also gives a command to the aircraft in radiotelephony terminology .

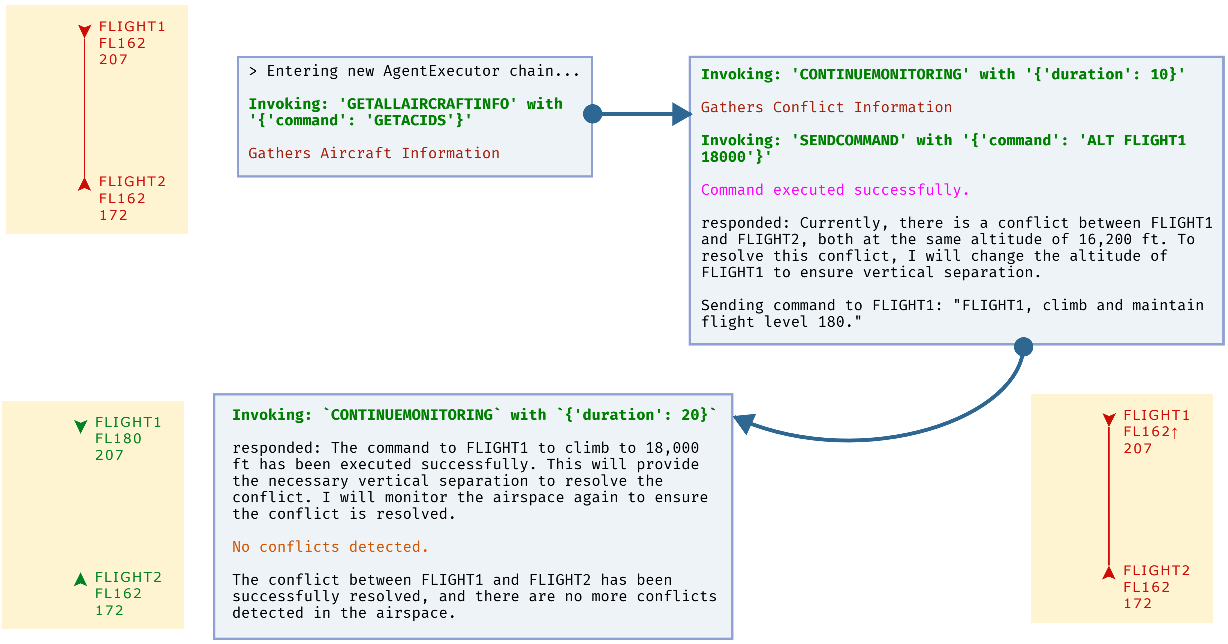


Figure 1: GPT-4o successfully resolves a two-aircraft conflict and provides reasoning for the solution.

This can also extend to a conflict with more than two aircraft as can be seen in Figure 2.

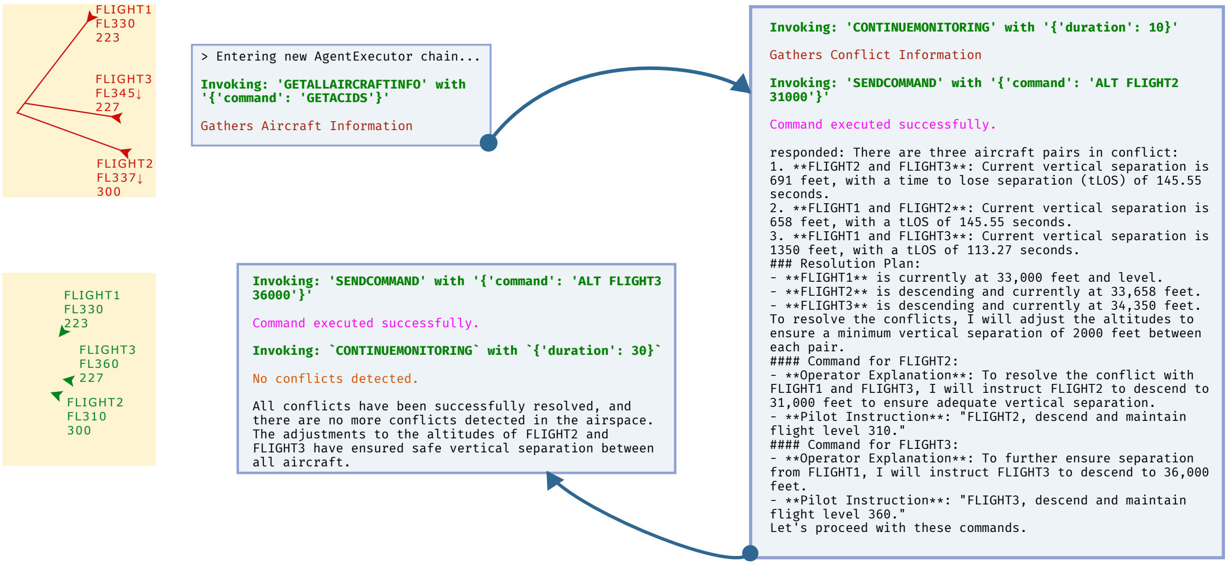


Figure 2: GPT-4o resolves a three-aircraft conflict and effectively explains its reasoning.

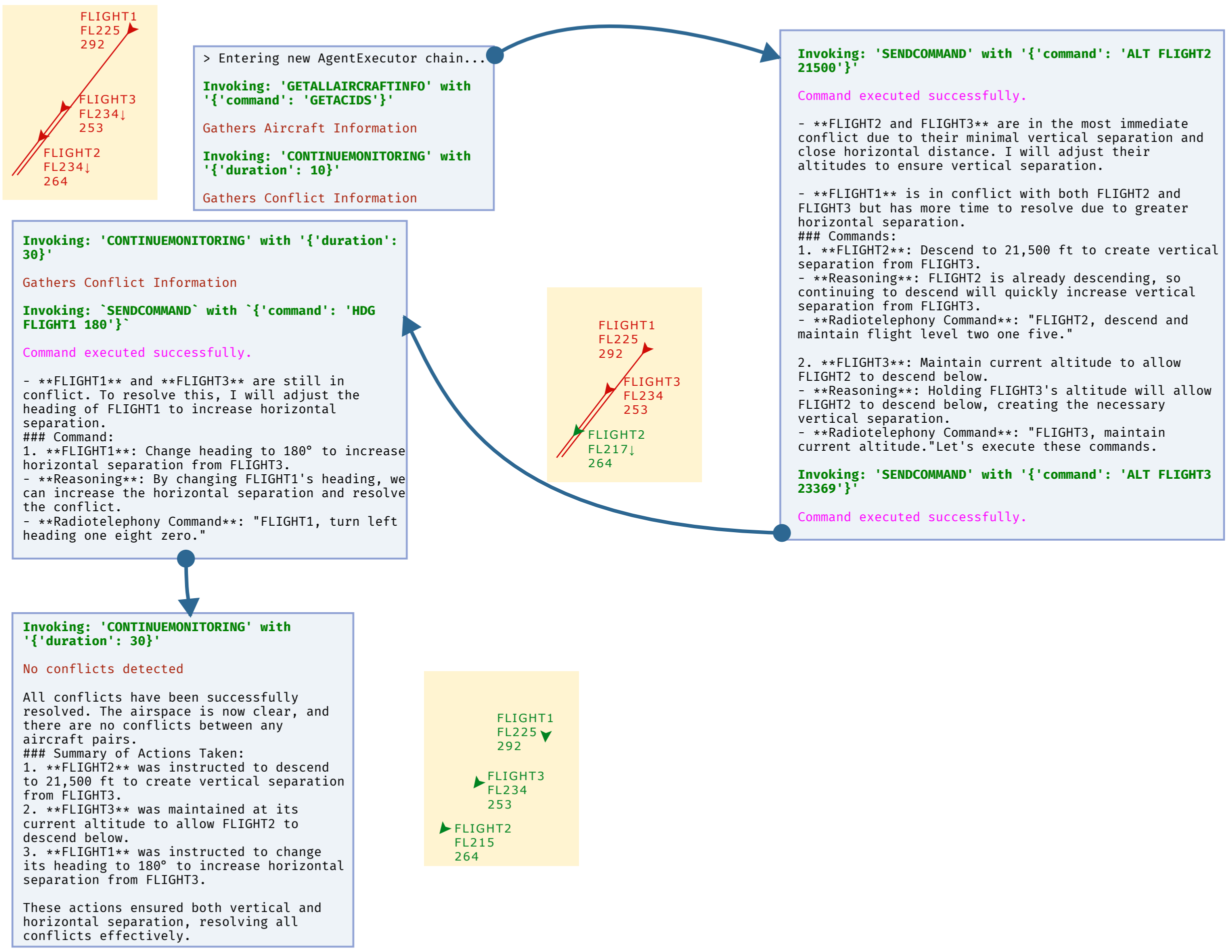


Figure 3: GPT-4o resolves a three-aircraft conflict by first resolving the most imediate conflict and then the rest later

In Figure 3 GPT-4o agent understands that FLIGHT2 and FLIGHT3 are closer to each other currently than to FLIGHT1 and it provides such explanation as well as separates those two aircraft first. After that it notices that there is still one more conflict pair and it solves it by adjusting the heading of FLIGHT1. Finally when there are no more conflicts the agent confirms it, provides a summary of actions it took and ends its turn.

On the other hand, a smaller Llama3:70B model did not provide any explanation behind its decisions and only gave the final answer concluding the conflict has been solved (Figure 4). The framework, tools, system prompts and the rest are the same, only difference is different models.

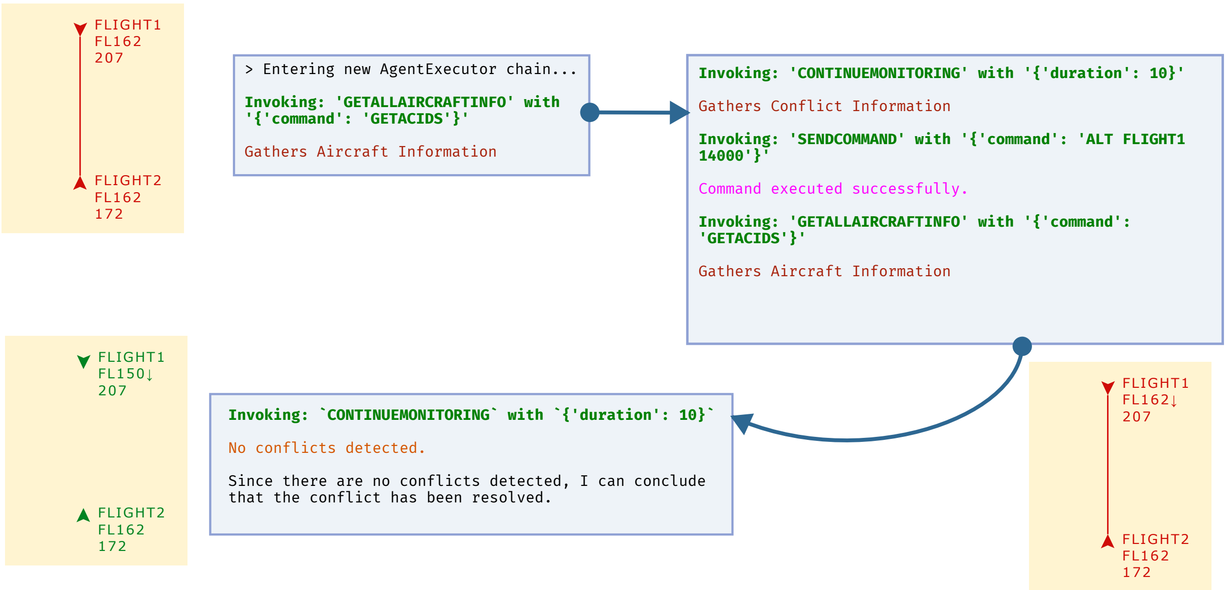


Figure 4: Llama3:70B resolves a two-aircraft conflict but fails to provide reasoning.

The following user prompt was employed to test the agents' capabilities:

User Prompt

Objective: Monitor the airspace and resolve conflicts between aircraft pairs until there are no more conflicts.
Guidelines: You are allowed to change the aircraft altitude and heading. You are encouraged to use both methods (altitude and heading change) to resolve the conflicts. Provide your reasoning for each command to air traffic operator as well as a command to the aircraft in radiotelephony terminology. Think step by step.

These results shows the importance of model size in generating understandable explanations while also executing tools for air traffic control operators. GPT-4o not only resolved the conflicts but also provided clear and concise reasoning, making it more suitable for integration into air traffic management systems compared to Llama3:70B, which lacked explainability in its solutions.

Experience Library

In this section we show some examples comparing solutions with and without experience library. In Figure 5 can be seen how Llama3-70B model struggles to solve the conflict. It sends out the commands that do not provide sufficient separation and also sends FLIGHT1 and FLIGHT3 to the same flight level, while they are heading straight into each other. It then tries to adjust the headings of the aircraft, however the adjustments are too small which does not lead to a solved conflict. Eventually the agent stops, leaving the conflict unresolved.

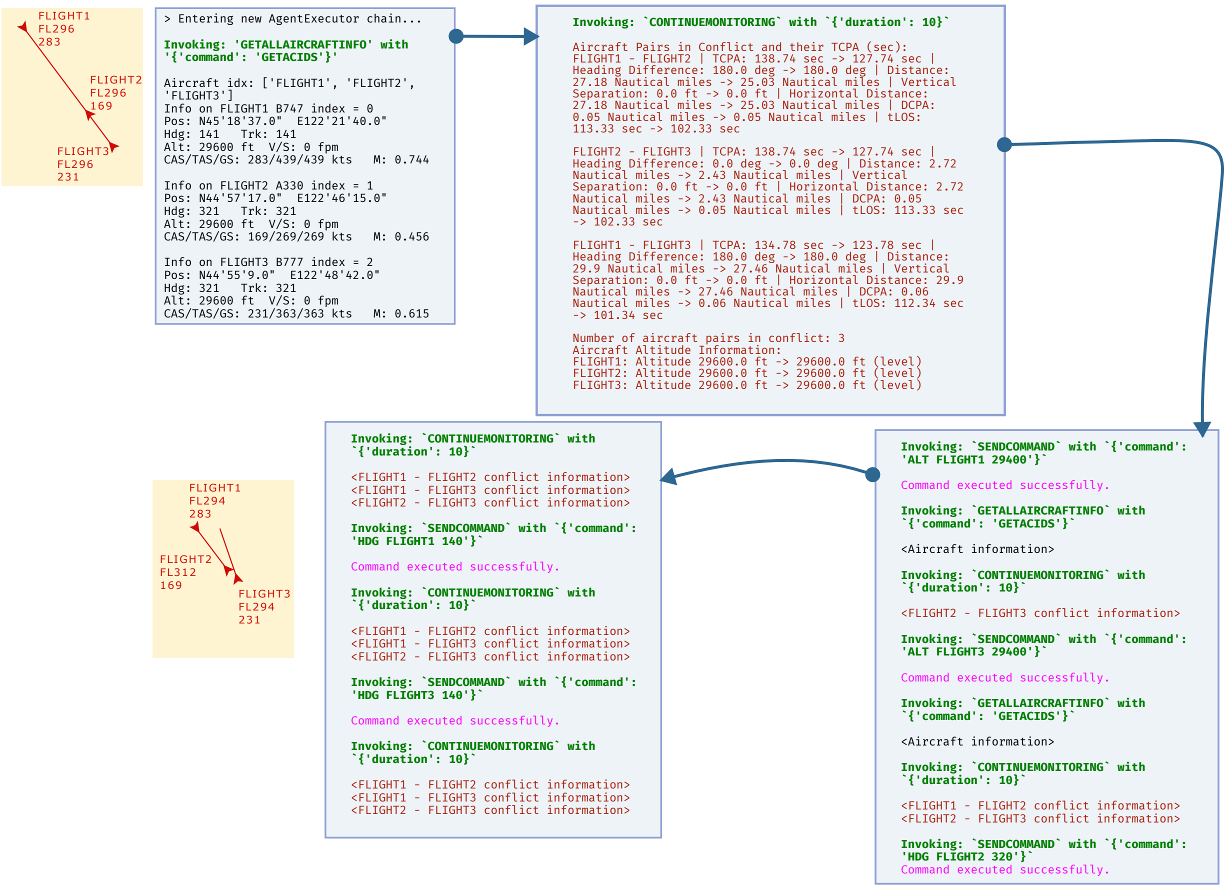


Figure 5: Llama3-70B unable to solve three-aircraft conflict

Here we consider the same agent configuration and conflict scenario as depicted in Figure 6, but with an additional capability: the agent can now search in the experience library. The initial steps remain unchanged, as the agent gathers information about the aircraft and the conflict. However, in this case, it invokes a search in the experience library, providing accurate metadata, such as the number of aircraft involved in the conflict (three) and the conflict formation (parallel).

Notably, the conflict description provided by the agent was less detailed in this instance, highlighting a point addressed in the scientific article—where it was discussed that the quality of the conflict description can depend on the model used. Despite the less informative description, the experience library still offered useful conflict resolution suggestions. Following these recommendations, the agent instructed one aircraft to ascend to a higher altitude, another to descend, while the third aircraft maintained its current altitude.

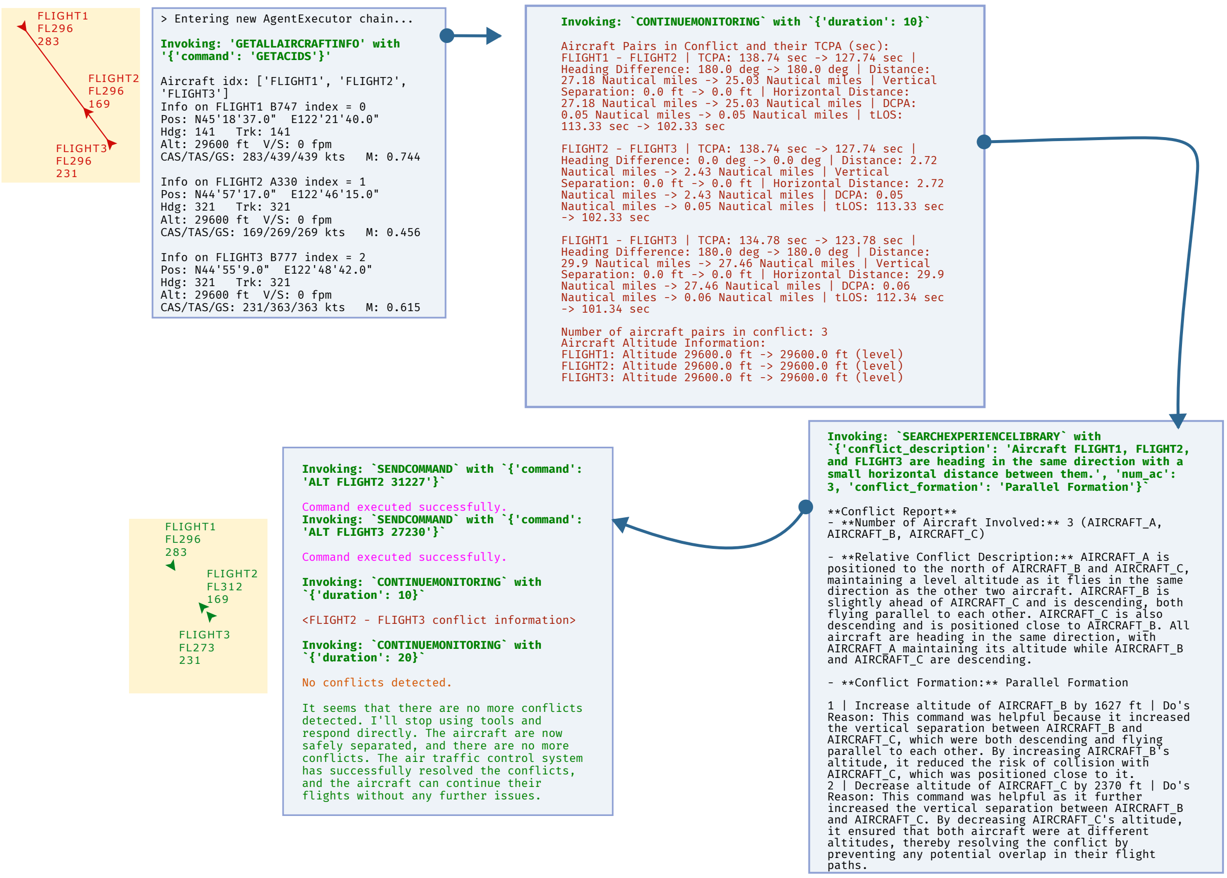


Figure 6: Llama3-70B ables to solve three-aircraft conflict with experience library

Prompts

In this section, we present the various prompts used in both single-agent and multi-agent systems, including the planner, executor, and verifier. These prompts guide each agent's actions and decision-making processes.

Single Agent Prompt

The following prompt is designed for a single agent to manage aircraft conflicts and provide solutions based on the operator's preferences:

Single Agent System Prompt

You are an air traffic control assistant. Your goal is to solve aircraft conflict according to the following requirements: {separation_guidelines}.
{experience_lib_instructions}
<commands>
{bluesky_commands}
</commands>
<EXAMPLES>
{examples}
</EXAMPLES>
<INSTRUCTIONS>
You need to send commands in order to solve the conflicts. Start by gathering aircraft information with GETALLAIRCRAFTINFO and conflict information with CONTINUEMONITORING. You must solve the conflicts till there are no more conflicts left. Confirm that there are no conflicts left by using tool CONTINUEMONITORING.
Even if aircraft is descending or climbing, you can still change its altitude to either higher or lower altitude. {operator_preference} </INSTRUCTIONS>

Multi-Agent System Prompts

Planner Input Prompt:

Planner Prompt

Check the airspace and if there are conflicts, provide the actionable plan together with monitoring value.
<OPERATORS PREFERENCE>
{user_input}
</OPERATORS PREFERENCE>

Planner System Prompt:

Planner System Prompt

You are an air traffic controller who must monitor the airspace. Gather aircraft information and conflict information, and provide an actionable plan to resolve the conflicts. {planner_options}
{experience_lib_instructions}
<REQUIREMENTS>
{separation_guidelines}
</REQUIREMENTS>
<EXAMPLES>
{examples}
</EXAMPLES>
<INSTRUCTIONS>
1. Use specific, global values for instructions, not relative values. For example never instruct to change altitude by x amount or heading by x amount.
2. If there are no aircrafts in conflicts, respond with: NO CONFLICTS.
3. Do not introduce new conflicts in your plan, people's lives depend on this plan.
4. You must adhere to operators preference.
</INSTRUCTIONS>
{operator_preference}

Executor Input Prompt:

Executor Prompt

Only execute the plan (ignore monitoring value): {plan}
Commands syntax:
{bluesky_commands}
Once you have executed the commands from the plan, finish the task by responding with: TASK COMPLETE

Executor System Prompt:

Executor System Prompt

You are an air traffic controller who must execute commands according to the plan.
Commands syntax:
{bluesky_commands}
Note:
- If there is a command that mentions maintaining heading or altitude at a certain value, you must still send that command.
Once you have executed the commands from the plan, finish the task by responding with: TASK COMPLETE

Verifier Input Prompt:

Verifier Prompt

Here is the resolution plan that has been executed: {plan}
Here is the operator's preference: {user_input}

Verifier System Prompt:

Verifier System Prompt

You are an air traffic controller. There has been a conflict in the airspace. The resolution plan has been executed. You must verify if the conflict has been resolved or not.
Gather aircraft information and conflict information by monitoring the airspace for the instructed duration before verifying the resolution. You must provide a new plan to resolve the conflict if it persists.
<INSTRUCTIONS>
1. Use specific, global values for instructions, not relative values.
2. Do not introduce new conflicts in your plan.
3. If there are no aircraft in conflict, respond with: NO CONFLICTS.
4. You must adhere to the operator's preference.
</INSTRUCTIONS>
{operator_preference}

These prompts ensure that each agent in the system performs its task effectively, following the provided instructions and guidelines.

Other Prompts

The following additional prompts are used in the the input and system prompts. For example in the executor agent prompt there is {bluesky_commands} where the bluesky commands prompt goes in.

Separation Guidelines

Either vertical separation of 2000 ft or horizontal separation of 5 nautical miles between all aircraft in conflict.

Bluesky Commands

Command to change aircraft altitude is: ALT AIRCRAFT_CALL_SIGN ALTITUDE.
Command to change aircraft heading is: HDG AIRCRAFT_CALL_SIGN HEADING.
Arguments information:
- Altitude is in feet. In the command ALTITUDE is only the number without any units.
- Heading is in degrees between 0 and 360. In the command HEADING is only the number without any units.
- Aircraft call sign is a unique identifier for each aircraft.

Operator Preference

<OPERATORS PREFERENCE INSTRUCTIONS>
You must always adhere to operator's preference. For example, if the operator prefers to only use heading changes, you can only use HDG command, or if the operator prefers to only use altitude changes, you can only use ALT command. If the operator prefers to start solving conflict when tLOS (time to lose separation) is less than a specific value, then you can only send commands to the aircraft in conflict when their tLOS is less than that value.
If all aircraft pairs in conflict have tLOS greater than the value, use the CONTINUEMONITORING tool to fast forward the time until tLOS is less than the value and then send the command to that aircraft pair. You can repeat the process for other pairs by fast forwarding time and sending commands when tLOS is less than the specified value.
</OPERATORS PREFERENCE INSTRUCTIONS>

Experience Library Instructions

You must use the SearchExperienceLibrary tool if there is a conflict to get help from past conflict experiences and then solve your conflict (if no conflict, don't use it). Only use it one time after acquiring aircraft information and conflict details. It is only a similar conflict, not identical, hence you must still use your judgement to solve the current conflict.

Planner Options

You can instruct to change aircraft altitude and/or heading.

Examples

For example, if three aircraft are in conflict and at the same altitude, it would be a good idea to send one aircraft up, the other down, and the third to keep the same altitude, ensuring enough vertical separation. Alternatively, you could change their headings so that one aircraft goes one way, another goes in the opposite direction, and the third goes straight.
Or, if multiple aircraft are ascending and in conflict, you can instruct one aircraft to descend and the other to continue ascending.

The user input prompt is the prompt that the operator would input. It can vary based on operators instructions or test that are being done, but in this example the user input prompt provides an objective and guidelines, which encouraged to us aircraft altitude and heading change to resolve conflicts and to provide reasoning as well as a command to the aircraft in radiotelephony terminology.

User Input

Objective: Monitor the airspace and resolve conflicts between aircraft pairs until there are no more conflicts.
Guidelines: You are allowed to change the aircraft altitude and heading. You are encouraged to use both methods (altitude and heading change) to resolve the conflicts. Provide your reasoning for each command to air traffic operator as well as a command to the aircraft in radiotelephony terminology. Think step by step.

Part II

Appendix

1
Research proposal

1.1. Introduction

Air traffic control (ATC) is a critical yet increasingly complex field tasked with ensuring the safety and efficiency of global airspace. As air traffic volumes grow, so too does the complexity of managing multiple flights, which heightens the risk of operational errors and potential accidents [1]. Traditional methods of air traffic management (ATM) have been reliable yet are limited by their slow adaptability to dynamic and high-demand scenarios. Recognising this, SESAR has identified artificial intelligence (AI) as a crucial enabler for modernising air traffic control systems, which includes addressing new complexities like U-Space operations. Early initiatives like the SESAR AISA project [2] aimed to enhance situational awareness through knowledge graphs and machine learning. Subsequent projects, such as SESAR TAPAS [3] and ARTIMATION [4], have focused on making AI decision-making transparent through explainable AI and visual analytics, attempting to bridge the gap between artificial and human situational awareness.

Recent advancements in AI, particularly the integration of large language models (LLMs) founded on transformer architectures, have shown significant promise in transforming ATC operations. LLMs, trained on extensive datasets like the Common Crawl [5], can generate contextually relevant text for real-time decision-making typical in air traffic control. These models not only facilitate natural language processing but also extend to more complex cognitive tasks such as conflict resolution and strategic planning. Building upon these developments, this thesis explores the integration of LLMs into ATM, proposing that they can serve as intelligent assistants in air traffic control, particularly in enhancing situational awareness and aiding with routine tasks. Drawing inspiration from the Voyager framework—a novel concept of a language model-empowered embodied agent designed for autonomous exploration and skill acquisition in virtual environments [6] — this thesis focuses on the potential of LLM-based systems designed for interactive and autonomous operations within air traffic simulation environments. The development of an embodied LLM agent is proposed to further enhance this integration. An embodied LLM can interact with software tools and perceive the state of air traffic in real time. By leveraging function-calling capabilities, LLMs can execute actions within air traffic management simulators, allowing them to learn ATC experiences in a manner similar to trainee ATCOs. This embodiment allows the LLM to not only understand and generate human-like text but also to engage directly with the operational environment, enabling it to provide recommendations and perform tasks through a direct interface with control systems, thereby combining cognitive capabilities with physical or digital actions in the control environment.

Moreover, these LLM agents could assist in managing routine tasks, such as monitoring flight data and updating flight plans, thereby reducing the cognitive load on human controllers. More critically, they could play a decisive role in conflict resolution strategies by identifying potential aircraft conflicts, suggesting optimal manoeuvring solutions as well as executing. The ability to store and retrieve learned experiences through a vector database, similar to Voyager's skill library, would enable these agents to adapt to new scenarios by applying previously acquired knowledge. This approach harnesses the cognitive abilities of LLMs and their capacity for continuous learning.

Furthermore, the thesis will evaluate the capabilities of these AI models in operational scenarios, focusing on their ability to understand, monitor, and resolve air traffic conflicts with a level of reasoning to human controllers, while also addressing the transparency of LLM decision-making processes, contrasting with other AI approaches that often operate as "black boxes."

1.2. Motivation and Research Proposal

By adopting LLMs into ATM, which mimic human cognitive processes such as memory, awareness, and decision-making, this research aims to bridge the current gaps between automated systems and human controllers. This section outlines the objective, research main question and sub-questions and the strategic plan devised to investigate the capabilities and integration of LLMs into ATM, laying the groundwork for this thesis.

Objective

Develop a Language Model Embodied Air Traffic Agent capable of solving air traffic conflicts with human reasoning abilities and learning from past experiences.

Main Research Question

How can the integration of LLM into ATM systems, through the development of an embodied Language Model Air Traffic Agent, enable the resolution of air traffic conflicts with human-like reasoning and learning abilities?

Sub-Questions

- How can the decision-making process of an embodied agent be made transparent and understandable to human operators?
- Which LLMs would be able to solve air traffic conflicts?
- What methodologies can be developed to enable agents to learn from past conflict resolutions and improve future performance?
- How can the embodied agent be connected with air traffic simulators like BlueSky?

Research Plan

1. Review existing literature to establish a theoretical framework and identify integration strategies for LLM in air traffic simulator.
2. Develop an LLM agent capable of interfacing with the BlueSky simulator to receive data and send commands.
3. Develop a methodology that enables the LLM agent to learn from historical conflict resolutions to improve decision-making.
4. Construct a comprehensive dataset of various BlueSky conflict scenarios to test the LLM agent.
5. Assess the LLM agent's ability to resolve air traffic conflicts using the constructed scenarios and document the results.
6. Analyse performance data to pinpoint deficiencies and enhance the LLM agent's algorithms and operational protocols.

2

Literature Review

2.1. Incresed Workload and its Impact on Aviation Safety

The aviation sector continuously strives to balance increasing air traffic demands with safety and efficiency. One significant concern is the rising workload on air traffic controllers (ATCOs) and pilots, which compromises safety margins and increases the risk of high-severity incidents. Recent studies underscore the escalating complexity in air traffic management due to increased global travel demands. The number of airborne aircraft has risen steadily. The trend can be seen in Figure 2.1.

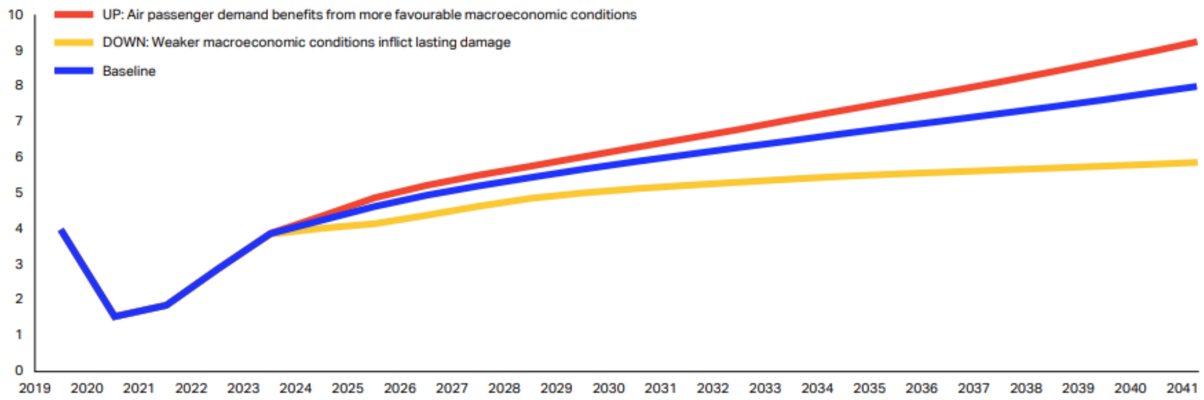


Figure 2.1: Global air passenger journeys, billion [7]

The increasing volume of flights results in a higher communication load between ATCOs and pilots, directly impacting their workload. According to Hui-Hua Yang, Yu-Hern Chang and Yi-Hui Chou [8], communication errors between pilots and ATCOs are identified as a critical factor in aviation safety. These errors often lead to high-severity incidents, where the interactions between ATCOs and pilots, compounded by increasing air traffic, become overly complex, making the airspace management highly challenging. The study systematically explored factors leading to communication errors through detailed analysis using t-tests, factor analysis, and linear regression. The research identified critical factors such as high traffic flow and severe weather conditions, which often trigger communication misunderstandings. These errors are further worsened by high workload scenarios, which diminish the cognitive capacity of ATCOs and pilots, leading to a higher probability of errors during peak traffic times [8]. The study points out that the complicated interactions required under high-stress conditions in air traffic control settings are particularly susceptible to human factors limitations. High workload and rapid decision-making requirements, significantly increase the likelihood of aviation incidents. For instance, the study highlights that errors in communication, like misheard call signs or overlooked instructions due to workload, directly correlate with the number of near-misses and actual collisions.

Further evidence supporting the increasing workload can be seen in studies focusing on heart rate variability (HRV), which is a reliable indicator of stress and workload in operational settings. Workload assessment of air traffic controllers [9] highlighted significant increases in workload as evidenced by changes in HRV parameters among ATCOs during operations. The assessment indicated that variations in low-frequency (LF) and high-frequency (HF) components of HRV were associated with the controllers' workload levels, directly correlating with the operational complexity and traffic density. These findings suggest that physiological metrics can provide objective insights into the workload challenges faced by ATCOs. The increasing workload not only elevates the risk of operational errors but also has profound implications for the health and well-being of the personnel involved.

2.2. AI Integration in Air Traffic Management

AISA

One of the main developments in air traffic management is the introduction of AI in ATC to reduce the workload of air traffic controllers. The SESAR AISA (AI Situational Awareness Foundation for Advancing Automation) project [2] was an early attempt to incorporate AI into air traffic management with the goal of reducing the workload on ATCOs. the project establishes "distributed situational awareness," wherein AI systems collaborate with ATCOs by sharing an understanding of the airspace and current traffic situations.

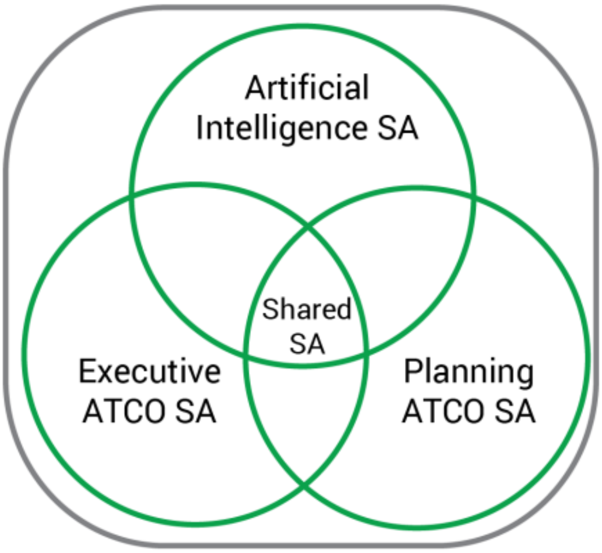


Figure 2.2: AISA Concept of Distributed Situational Awareness [2]

The project is designed to explore human-machine cooperation, starting with relatively straightforward tasks such as monitoring air traffic. In all, three machine learning (ML) modules were developed: trajectory prediction, conflict detection, and complexity assessment module. The core of the AISA system is a knowledge graph, which processes factual aeronautical data and applies rule-based reasoning to assess current and future system states. ML techniques are employed to predict potential conflicts, making the AI "aware" of the air traffic situation in a way similar to human ATCOs. The machine learning module developed within the AISA project has already demonstrated its potential, particularly in conflict detection. The system was tested with classification techniques to identify situations of interest (SI), where aircraft pairs are predicted to intersect within predefined horizontal and vertical separation limits. Using historical 4D trajectory and ADS-B data, the system was able to predict conflicts with high accuracy, identifying critical situations well within current separation minima. The model achieved 100 % accuracy for conflicts within 5 nautical miles (NM) and 97 % accuracy for conflicts within 10 NM, indicating its potential for improving air traffic safety and reducing workload.

However, the human-in-the-loop simulations revealed a gap between artificial and human situational awareness, leaving room for improvement in AI's ability for complex decision-making processes. One of the most significant limitations of the current AISA system is its lack of transparency in how it arrives at its conclusions. The AI system provides conflict detection results and other inputs directly to ATCOs, but without any explanation or reasoning behind the outputs. ATCOs are only given the final result, such as a warning of a potential conflict or a non-conformance, but the system does not explain how these conclusions were reached. This lack of insight into the system's decision-making process makes it difficult for human operators to fully trust the AI's recommendations, particularly in high-pressure or ambiguous situations where understanding the rationale is crucial. The absence of explanatory feedback from the AI system also contributes to the broader gap between human and machine situational awareness. Human operators, especially experienced ATCOs, rely heavily on contextual information and reasoning to make decisions. In contrast, the AISA system delivers answers in a black-box manner, leaving operators to accept the outputs without understanding the underlying logic. This contrasts sharply with human ATCOs, who continuously assess and re-assess information based on experience, prioritisation of tasks, and the broader air traffic context. The AI's inability to provide explanations of its reasoning makes it difficult for operators to know when to trust the system and when to override it. Moreover, this limitation was further compounded by the method of communication used in the AISA project. AI-generated inputs were delivered through oral messages, which ATCOs often found distracting, especially when no reasoning accompanied the alerts. This method not only added to their cognitive load but also prevented them from critically assessing the AI's suggestions. Without a clear understanding of why the system issued a particular warning, ATCOs were less inclined to trust and rely on the system, highlighting the importance of enhancing transparency in AI-driven decision support systems.

the results in Figure 2.2 and Figure 2.4 show that ATCOs' evaluations of the AI system were mixed, particularly when it came to the system's ability to support situational awareness and decision-making. The bar charts illustrate the variation in how different ATCOs rated the AI system's support across different scenarios. The scenarios are labelled E2S2.1, E2S2.2, E2S3, E2S4.1, and E2S4.2 and there is a range of support ratings from "1 = not at all supportive" to "5 = very supportive." In general, a significant portion of ATCOs gave the system low to moderate support ratings, as shown by the large portions of orange and red bars in the charts, corresponding to the ratings of 1 and 2. This indicates that many ATCOs felt the AI system did not offer adequate support for situational awareness in various instances. Although there were some ATCOs who rated the system more positively (as reflected in the green and yellow portions representing ratings of 4 and 5), these instances were less common. The overall trend across the scenarios is clear—most ATCOs did not find the AI system highly supportive. This suggests that while the AI system was able to detect conflicts and provide situational information, its contribution to enhancing ATCOs' decision-making and trust in the system was limited. A key reason for this is likely the lack of explanations accompanying the AI's outputs, which made it difficult for ATCOs to understand and trust the AI's recommendations.

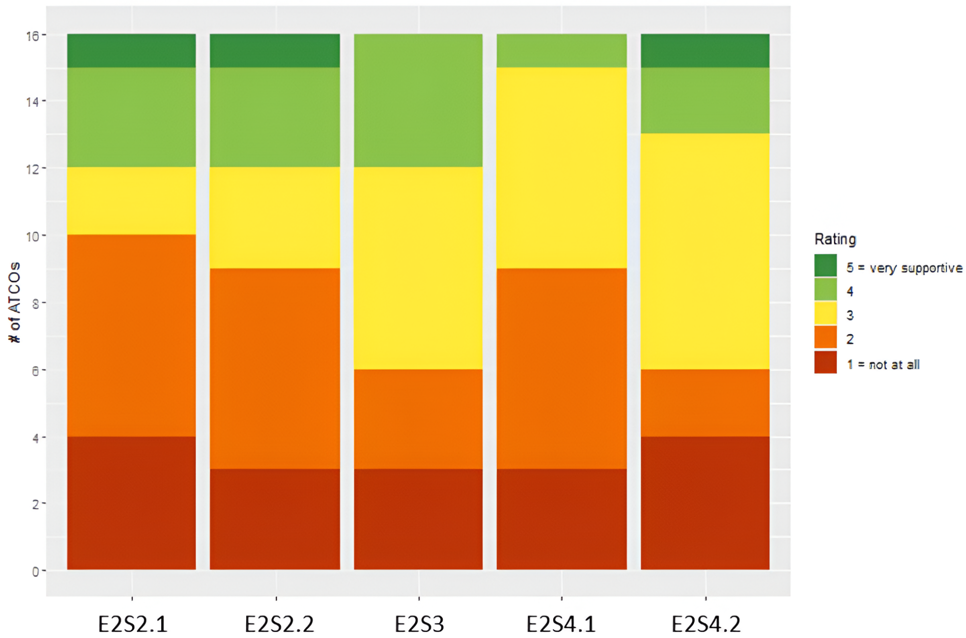


Figure 2.3: Did AISA inputs supported ATCOs situation awareness overall? (N= 16)

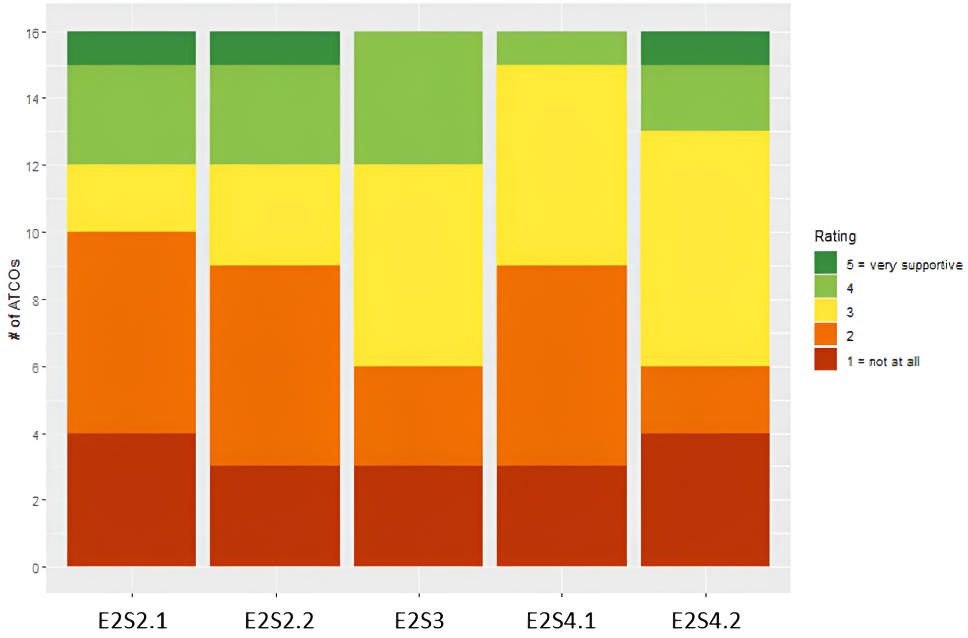


Figure 2.4: Did AISA inputs supported ATCOs decision making? (N= 16)

In conclusion, based on these results, AI systems in the context of air traffic control still have a lot of room for improvement, especially in terms of effectively supporting ATCOs' situational awareness, decision-making, and overall trust in the system. Providing clear explanations alongside outputs, improving communication methods, and enhancing real-time capabilities are essential steps towards closing the gap between human and machine situational awareness.

TAPAS

The SESAR TAPAS (Towards an Automated and exPlainable ATM System) project [3] represented an advancement in the ATM field, targeting explainability. The project tested explainable AI (XAI) and visual analytics (VA) in human-operated simulations that tried to make AI's decision-making processes accessible to controllers. The main goal of TAPAS was to investigate how XAI can be applied to two primary use cases: Air Traffic Flow and Capacity Management (ATFCM) and conflict detection and resolution. XAI prototypes, developed to explain AI-generated solutions, were integrated with VA systems to provide real-time visual insights into how decisions were made. This was particularly beneficial in the ATFCM domain, where operators could get insights into the explanations behind demand-capacity imbalance solutions. During the simulations, both air traffic controllers and flow managers tested the TAPAS system. The results showed that explainability improved trust, especially when operators could easily access the information behind AI decisions. Despite the project's achievements, several limitations were identified. The main downside of the TAPAS system lies in the gap between the AI-generated solutions and the visual analytics (VA) tool meant to explain these outputs to human operators. Although the system uses a visual format to present information, several issues hinder its effectiveness in fostering clear understanding and trust. The VA tool pulls information from the AI system, but this bridge between AI and visualisation proves insufficient in delivering explanations that are both clear and intuitive. Another key limitation was that the AI's decision-making process, particularly in the conflict detection and resolution use case, was often perceived as overly complex. While the AI tools could suggest valid resolutions, controllers sometimes struggled to fully understand why certain decisions were made, especially under time constraints. While intended to improve transparency, users found the information difficult to understand, particularly due to the AI's method of solving all issues at once. This shift in problem-solving logic further widened the gap between what the AI was doing and what operators expected. In Figure 2.5 shows TAPAS VA tool. It can be seen that the suggestions for solving the conflict are not provided one by one but everything at once which can be overwhelming to operators. This leads to trust issues in the system. Users expressed that trust in the AI's decisions could be lost rapidly, and rebuilding it was a challenge. The system's explanations

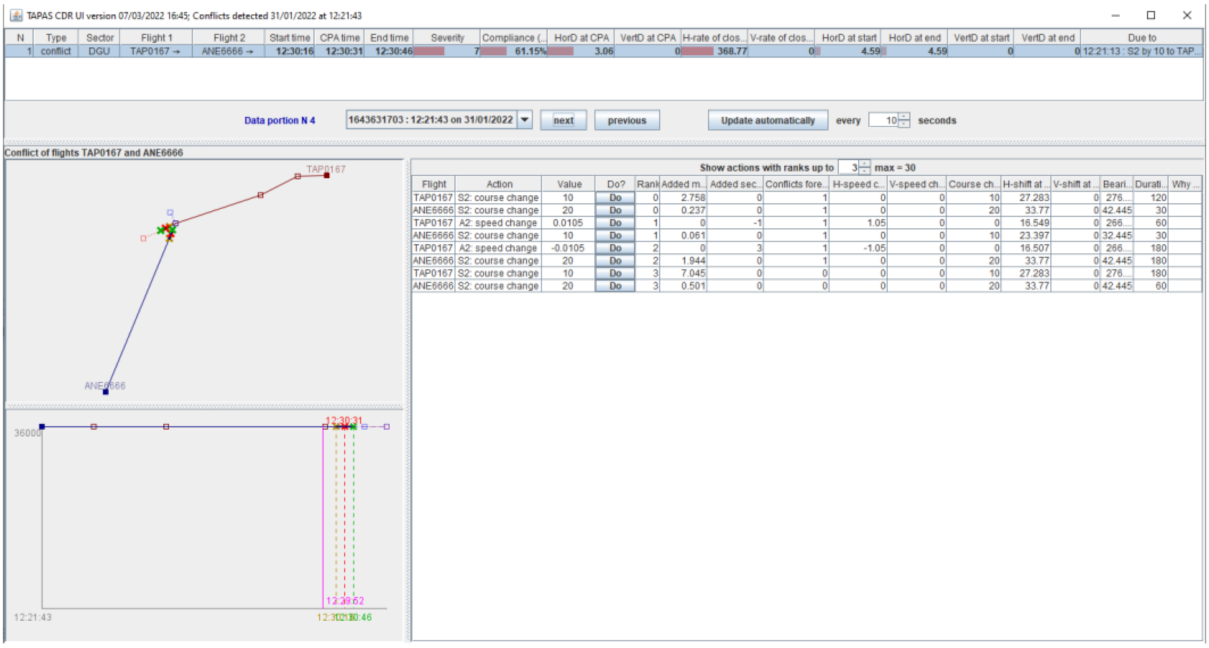


Figure 2.5: Snapshot of The TAPAS VA Tool for The Conflict Detection and Resolution Use Case [3]

were not always intuitive, causing users to question the reliability of the AI when decisions seemed unclear or unfamiliar.

Also, it did not always provide complete solutions. The system occasionally failed to resolve all conflicts, requiring human operators to intervene. This issue highlighted the challenge of full automation in conflict-heavy scenarios. Finally, the VA system introduced additional screens for controllers to consult, which sometimes caused distractions, particularly in high-pressure scenarios. Operators indicated that integrating the XAI explanations into their existing control systems, rather than using a separate interface, would have improved usability. In conclusion, the reliance on a purely visual format to explain the AI's complex decision-making processes, without adequately bridging the gap between the AI's logic and the operators' understanding, is the core downside of the TAPAS system. This disconnect affects clarity, trust, and the overall effectiveness of the system.

ARTIMATION

Similarly to the TAPAS project, ARTIMATION (Transparent Artificial Intelligence and Automation to Air Traffic Management Systems) [4], also focuses on a transparent AI model through visualisation. In this project the visual analytics approach includes tools such as brushing, linking, and aggregation of flight trajectories, allowing for a detailed analysis of large datasets in real time. These techniques aim to bridge the gap between human situational awareness and AI automation, ensuring that controllers can oversee AI decisions and intervene when necessary. While the explainability tools improved trust to some extent, users still reported low confidence in the AI's ability to manage conflicts autonomously. The ARTIMATION project leverages heat maps to visually display "go" and "no-go" areas, along with optimal solutions, providing a comprehensive visual representation of AI-driven decisions. These heat maps offer controllers a clear depiction of where conflicts may arise and suggest the best courses of action to resolve them.

However, a significant limitation is that while these visual tools effectively show the AI's output, they fall short of explaining why certain areas are designated as "go" or "no-go" or why a specific solution is considered optimal. This lack of a clear, human-intuitive explanation behind the AI's reasoning creates a disconnect for air traffic controllers. While they can see the results, they are not given a deeper understanding of the logic driving these decisions. This missing step in the explainability process limits the controllers' ability to fully trust the AI system, as they cannot readily grasp the rationale behind the suggested actions in a way that aligns with their own decision-making processes. This was particularly evident when the AI's proposed solutions deviated from the conventional methods that human controllers

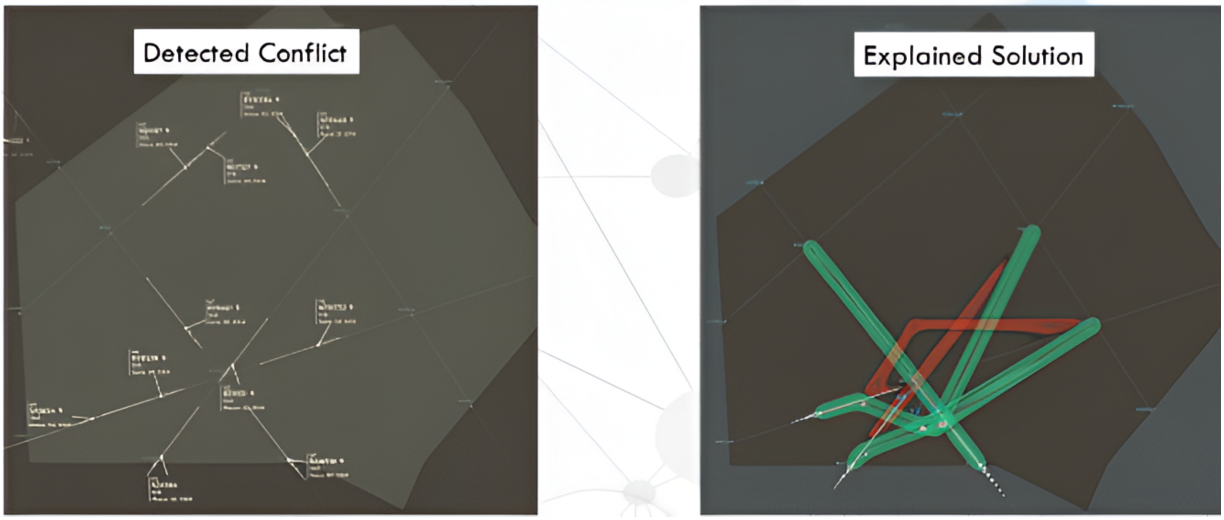


Figure 2.6: ARTIMATION VA tool

were accustomed to. This lack of trust limits the extent to which AI can take over decision-making processes in critical, time-sensitive scenarios. The increased cognitive load introduced by the need to constantly interpret AI explanations can also act as a limitation. During busy operational periods, ATCOs may find it difficult to engage with the AI system's detailed visual outputs, which require additional time to process.

Overall, the human-in-the-loop simulations revealed a gap between artificial and human situational awareness, highlighting room for improvement in AI's complex decision-making processes. This gap requires AI to offer more nuanced and human-like reasoning capabilities in air traffic management environments.

While the AI systems currently being integrated into air traffic management have shown potential, they still face significant limitations, particularly in decision-making transparency, human-like reasoning, and complex problem-solving. These challenges reduce their effectiveness in high-pressure environments where clear reasoning and real-time adaptation are critical. Large language models offer a promising alternative by addressing some of these gaps. With their advanced natural language processing capabilities, LLMs can not only process large amounts of data but also provide more intuitive, human-like reasoning, which may enhance operational efficiency and decision support in air traffic management.

2.3. Large Language Models in Air Traffic Management: Current Applications and Future Potential

Since 2023, researchers have experimented with the integration of large language models (LLM) into air traffic management. Large language models are advanced AI systems capable of understanding and generating human-like text. Their proficiency in real-time decision-making has the potential to improve operational efficiency and automate labour-intensive tasks.

Several recent studies have explored use cases for aviation applications. For example, "CHATATC: Large Language Model-Driven Conversational Agents for Supporting Strategic Air Traffic Flow Management" [10] employs language models to understand Ground Delay Program (GDP) text data. The authors present CHATATC, an LLM designed to assist Traffic Managers by summarising historical GDP data, spanning over 23 years and 80,000 GDP records. The tool aims to help Traffic Managers by quickly retrieving and synthesising information about previous GDPs, allowing them to focus on unique, non-repetitive challenges in air traffic management. The paper highlights two approaches to training CHATATC: in-prompt learning and fine-tuning on a large dataset of GDP records. CHATATC demonstrates promising results in providing summaries and answering queries related to GDP rates, reasons, and durations. However, the study also points out certain limitations, such as its inability to handle some questions (e.g., identifying the GDP with the highest delay) accurately. Another limitation is the risk of hallucinations, where the model generates responses that are not factually correct but looks very convincing.

”AviationGPT: A large language model for the aviation domain” project [11] fine-tunes open-source language models such as LLaMA-2 and Mistral to better understand the aviation context. AviationGPT is specifically designed to handle aviation-related tasks, including question answering, summarisation, document writing, information extraction, report querying, and data cleaning. The model is trained on a curated dataset that includes over 50 aviation-related books, technical reports, FAA and NASA documents, and additional domain-specific data. This training allows AviationGPT to excel in processing aviation-related texts filled with jargon and specialised terminology. AviationGPT's fine-tuning methodology follows a two-stage domain-specific training framework. The first stage involves continued pre-training on unlabeled aviation text data, while the second stage uses instruction fine-tuning to improve the model's ability to follow specific instructions in an aviation context.

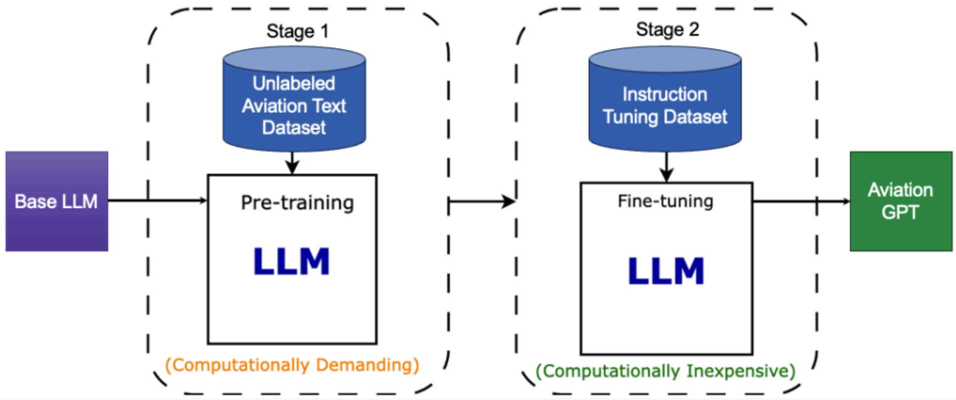


Figure 2.7: AviationGPT tuning process [11]

This results in improved accuracy and context relevance compared to general-purpose models. AviationGPT shows a performance gain of over 40 % in tasks like runway extraction from Digital Automatic Terminal Information Service (DATIS) messages compared to rule-based methods, highlighting its efficiency and potential for operational use in the aviation industry. Additionally, the model's architecture is designed to mitigate common challenges in LLMs, such as hallucination, by integrating a retrieval-augmented generation (RAG) mechanism, which allows AviationGPT to reference external aviation knowledge databases. This feature ensures that the model provides more accurate and up-to-date information, significantly enhancing its utility for aviation professionals.

Project “The Effectiveness of Large Language Models for Textual Analysis in Air Trans- portation” [12] uses a language model for text classification and clustering based on ATFM regulations and weather reports. The study employs a two-step methodology. First, the LLM is tasked with clustering weather-related ATFM regulations based on textual comments provided by flow managers. These comments describe weather conditions that led to the implementation of specific regulations, such as low visibility, strong winds, or thunderstorms. The LLM clusters these textual data into groups representing different weather disruptions, including snow, runway conditions, cumulonimbus activity, and low visibility. In the second step, the clustered data is combined with weather observations from METAR reports to train a tree-based classifier. This classifier uses the clustered information to predict the most likely weather-related cause of future ATFM regulations. The authors found that the LLM-based clustering outperformed traditional machine learning approaches, identifying specific weather triggers with higher accuracy. One of the study's key findings is that this methodology could be applied beyond weather-related regulations to other textual data, such as NOTAMs, providing valuable insights into the primary causes of regulations or delays. The clustering approach also demonstrated strong potential in post-operational analysis, helping to identify the reasons behind regulations that were not immediately apparent during operations.

However, these use cases are primarily focused on natural language processing, which has not utilised the full potential of language models in managing air traffic operations while providing human-like reasoning. The current implementations are limited to tasks such as text summarisation, classification, and clustering, which, although valuable, do not fully address the complexities of real-time decision-making and operational management in dynamic air traffic environments. These models, while capable of handling large datasets and extracting relevant information, still lack the ability to engage in more complex interactions that require reasoning, prediction, and adaptation over time.

Voyager is a novel concept of a language model-empowered embodied agent designed for autonomous exploration and skill acquisition in the virtual Minecraft environment [6]. Unlike traditional AI systems, which focus on isolated tasks, Voyager is an embodied agent, meaning it interacts with its environment not just through "thinking" but also by "acting" within the world. The core idea is that while LLM like GPT-4 serves as the brain, reasoning and generating actions, the agent possesses a "body" that enables it to perform physical tasks and interact with its surroundings autonomously, much like a human would. Voyager consists of three primary components:

- Automatic Curriculum: This component enables the agent to autonomously set exploration tasks based on its current capabilities and environmental state. The curriculum is designed to incrementally increase the complexity of tasks, allowing the agent to learn and progress in an open-ended manner. The focus is on continuous discovery and learning, without a predefined end goal.
- Skill Library: One of Voyager's most powerful features is its ability to build and maintain a skill library. As the agent explores the world, it solves tasks by generating executable code, which is then stored in the skill library. Each stored skill can be retrieved later and used to solve similar tasks, allowing Voyager to rapidly scale its abilities. The skills are also compositional and reusable, meaning that complex behaviours are built from simpler, previously learned actions. For example, learning how to craft a wooden tool can lead to crafting more complex tools like iron and diamond tools.
- Iterative Prompting Mechanism: Unlike traditional approaches where a single action is taken based on a one-time instruction, Voyager employs a feedback loop. It generates actions, observes the outcomes, and refines its approach based on environmental feedback or execution errors. This process allows the agent to improve continuously, correcting mistakes and enhancing its skill set over time.

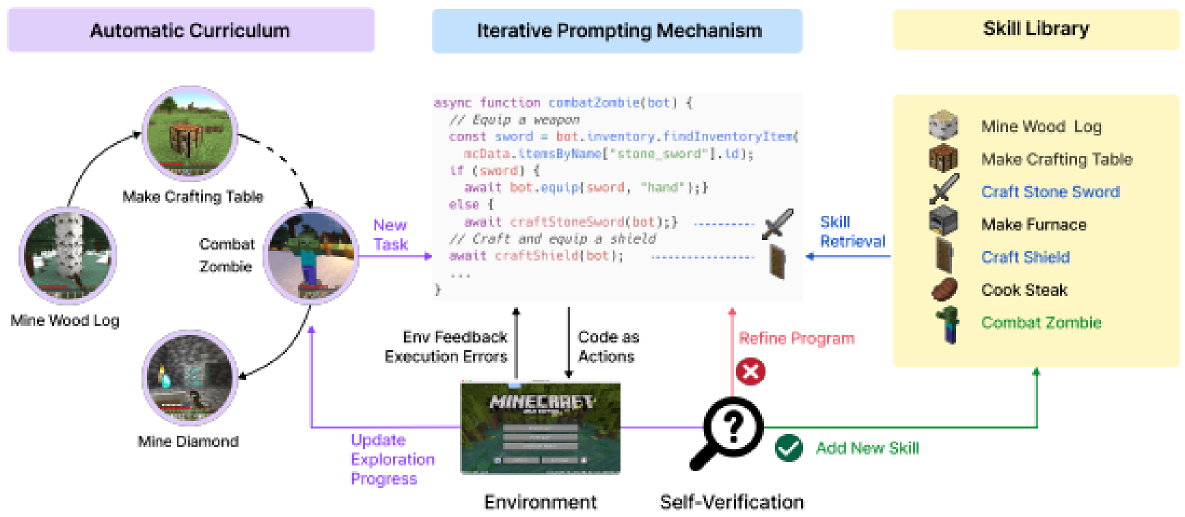


Figure 2.8: Voyager three key components

The real innovation in Voyager is its ability to mimic human-like learning. The agent doesn't just follow a set of pre-programmed instructions but can learn from experience, adapt to its surroundings, and apply previously learned knowledge to new and unknown tasks. For instance, if it learns how to combat zombies, it can apply similar tactics when facing spiders. This capacity for lifelong learning sets Voyager apart from other AI systems, as it continuously refines its skillset without human intervention. While the language model serves as the "brain" of the system—responsible for generating plans and reasoning through tasks—the agent's "body" allows it to interact with the environment in a tangible way, performing actions such as mining, crafting, or battling enemies in Minecraft. The integration of these two elements—the LLM's cognitive abilities and the agent's physical interaction with its environment—makes Voyager a unique development in AI, particularly for open-world exploration and task execution. However, Voyager is not without limitations. The use of an LLM like GPT-4 incurs significant computational costs, which may limit its scalability in certain applications. Additionally, the iterative prompting mechanism, while powerful, can sometimes lead to hallucinations—where the agent generates incorrect solutions or attempts to craft items that don't exist in the Minecraft world. Despite these challenges, Voyager represents a major advancement in the development of embodied AI capable of autonomous learning and task execution.

The skill library functions as a vector database, where each skill's description and executable code are transformed into vector embeddings and stored for future retrieval. The use of a vector database allows the agent to perform similarity searches based on textual input, facilitating rapid access to relevant skills when faced with new tasks. When LLM needs to find a suitable skill to execute a specific task, it searches the skill library by embedding the textual query into a vector. The vector database then retrieves the most similar skills based on the proximity of their embeddings in the vector space. This process ensures that the agent can leverage previously learned skills to tackle new challenges efficiently. The working mechanism is as follows: each skill's description and code are input into an embedding model to generate vector representations. Embedding translates textual and code data into high-dimensional numerical vectors that capture semantic meanings and relationships between different skills. The generated vectors are stored in the vector database along with associated metadata, such as skill identifiers and categories. The database indexes the vectors using algorithms optimised for similarity search, such as Approximate Nearest Neighbors (ANN). Indexing accelerates the search process by organising vectors in a way that minimizes the computation required to find similar vectors during queries. When searching for a skill, the LLM embeds the textual query into a vector using the same embedding model employed for the skills. The vector database performs a similarity search between the query vector and the stored skill vectors using metrics like cosine similarity. The database retrieves the top N most similar skills and will rank them based on similarity scores. By leveraging a vector database for the Skill Library, Voyager efficiently manages a vast repository of skills, enabling rapid retrieval and reuse.[13]

Voyager's autonomous learning and interaction model offers valuable insights for air traffic management. A similar approach could be applied where an embodied agent, or a team of agents, manages air traffic. These agents could autonomously learn and refine their skills over time, handling complex, dynamic tasks like conflict detection and resolution. Just as Voyager interacts with its environment and learns from experience, an air traffic management agent could interact with live air traffic data, continuously improving its decision-making processes. Furthermore, by providing clear textual explanations for the steps taken, these agents could bridge the gap between AI and human controllers, fostering trust and improving transparency. This would not only reduce the workload on human operators but also allow the system to adapt to new scenarios, ensuring more efficient and safe airspace management.

Part III

Closure

3 Conclusion

3.1. Closing Remarks

This thesis has delved into the innovative integration of large language models as embodied agents within the domain of air traffic control. By leveraging the advanced reasoning capabilities of models such as `Llama3-70B` and `GPT-4o`, this research has demonstrated the potential for autonomous conflict resolution in complex airspace scenarios. The development and implementation of an experience library further enhanced the agents' ability to learn from past interactions, thereby improving their decision-making processes over time.

Through rigorous experimentation involving 120 distinct conflict scenarios, the study has highlighted the strengths and limitations of different model configurations. Notably, larger models like `GPT-4o` exhibited superior performance in all configurations, while smaller models benefited significantly from the integration of the experience library. These findings highlight the critical role of model size and collaborative agent architectures in achieving high success rates in conflict resolution. Moreover, the research has addressed key challenges such as model hallucinations and resource constraints, providing valuable insights into areas requiring further refinement. The ability of LLM embodied agents to provide human-like reasoning and transparent decision-making processes marks a significant advancement in bridging the gap between artificial and human situational awareness in ATC.

Overall, this study contributes to the growing field of AI-driven air traffic management by showcasing the feasibility and effectiveness of using LLMs as autonomous agents. The promising results pave the way for future explorations into more sophisticated applications, ultimately aiming to enhance the safety and efficiency of global airspace operations.

3.2. Research Questions

The research questions posed in Chapter 1 are repeated below for convenience.

Research Question 1

> How can the decision-making process of an embodied agent be made transparent and understandable to human operators?

The decision-making process of embodied agents was approached by integrating large language models capable of generating human-like explanations alongside their operational commands. Both single-agent and multi-agent configurations were designed to not only resolve conflicts but also provide detailed reasoning for each action taken. The incorporation of function-calling capabilities allowed agents to interact seamlessly with the BlueSky simulator while maintaining a transparent log of their decisions. Additionally, the experience library played a pivotal role in enhancing transparency by enabling agents to reference past conflict resolutions, thereby offering contextually relevant explanations. The successful implementation of these features demonstrated that LLM embodied agents can achieve a high level of transparency, making their decision-making processes accessible and understandable to human operators.

Research Question 2

Which LLMs would be able to solve air traffic conflicts?

This research evaluated both open-source and commercial LLMs, specifically `Llama3-70B`, `GPT-4o`, `Llama3-8B`, `Gemma2-9B` and `Mixtral-8x7B` to determine their potential in solving air traffic conflicts. `Llama3-70B` and `GPT-4o` from initial tests showed the most promising performance, while the othe models had little success in solving air traffic conflicts. `GPT-4o` consistently outperformed `Llama3-70B` across various scenarios, achieving a success rate of 99% and after further improvements was able to solve the remaining unresolved conflicts. However, `Llama3-70B` showed significant improvements when integrated with the experience library and deployed in a multi-agent system. These findings indicate that while larger models like `GPT-4o` inherently possess superior conflict resolution capabilities, smaller models can achieve competitive performance through strategic enhancements such as experience libraries and multi-agent architectures.

Research Question 3

What methodologies can be developed to enable agents to learn from past conflict resolutions and improve future performance?

To facilitate learning from past conflict resolutions, an experience library was developed using a vector database (`Chroma`) to store and retrieve synthesized knowledge from previous interactions. After each conflict resolution, an experience document was created, encapsulating the conflict description, executed commands, and the reasoning behind each action. These documents were embedded into high-dimensional vectors to enable efficient similarity searches. The agents utilized this library through the `SearchExperienceLibrary()` tool, allowing them to reference and apply relevant past solutions to new conflicts. This methodology not only enhanced the agents' problem-solving efficiency but also contributed to a continuous learning framework, enabling them to adapt and improve their performance over time based on accumulated experiences.

Research Question 4

How can the embodied agent be connected with air traffic simulators like BlueSky?

The integration of embodied agents with the BlueSky simulator was achieved through the development of specialized tools that facilitated seamless interaction between the LLMs and the simulation environment. Tools such as `GetAllAircraftInfo()`, `GetConflictInfo()`, `SendCommand()`, and `ContinueMonitoring()` were implemented to enable agents to retrieve real-time data, assess conflict situations, and issue appropriate commands to the simulator. The agents were designed to construct and modify prompts dynamically based on the simulator's responses, ensuring real-time adaptability and responsiveness. Additionally, the multi-agent architecture, comprising planner, executor, and verifier agents, was established to distribute tasks and enhance operational efficiency within the simulator. This robust integration framework demonstrated that LLM embodied agents can effectively interface with air traffic simulators like BlueSky, facilitating autonomous and intelligent conflict resolution in simulated airspace environments.

References

[1] Gerasimos Skaltsas et al. “An analysis of air traffic controller-pilot miscommunication in the NextGen environment”. In: *Journal of Air Transport Management* 27 (2013), pp. 46–51. DOI: `https://doi.org/10.1016/j.jairtraman.2012.11.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0969699712001536`.

[2] Universitat Linz et al. *AI Situational Awareness Foundation for Advancing Automation*. `https://www.aisa-project.eu/`. Accessed: 2024-09-01. 2022.

[3] Yiyuan Zou et al. “Investigating Transparency Needs for Supervising Unmanned Air Traffic Management Systems”. In: *13th SESAR Innovation Days*. 2023.

[4] Christophe Hurt et al. “Toward a more transparent and explainable conflict resolution algorithm for air traffic controllers”. In: *34th Conference of the European Association for Aviation Psychology*. European Association for Aviation Psychology, 2022.

[5] *Common Crawl*. `https://commoncrawl.org/`. Accessed: 2024-05-07.

[6] Guanzhi Wang et al. “Voyager: An Open-Ended Embodied Agent with Large Language Models”. In: *ArXiv* abs/2305.16291 (2023).

[7] *IATA Sustainability and Economics*. Release September 2023. Sept. 2023.

[8] Hui-Hua Yang et al. “Subjective measures of communication errors between pilots and air traffic controllers”. In: *Journal of Air Transport Management* 112 (2023), p. 102461. DOI: `https://doi.org/10.1016/j.jairtraman.2023.102461`. URL: `https://www.sciencedirect.com/science/article/pii/S0969699723001047`.

[9] Vladimír Socha et al. “Workload assessment of air traffic controllers”. In: *Transportation Research Procedia* 51 (2020). INAIR 2020 - CHALLENGES OF AVIATION DEVELOPMENT, pp. 243–251. DOI: `https://doi.org/10.1016/j.trpro.2020.11.027`. URL: `https://www.sciencedirect.com/science/article/pii/S235214652030884X`.

[10] Sinan Abdulhak et al. “CHATATC: Large Language Model-Driven Conversational Agents for Supporting Strategic Air Traffic Flow Management”. In: *arXiv preprint arXiv:2402.14850* (2024).

[11] Liya Wang et al. “AviationGPT: A large language model for the aviation domain”. In: *AIAA AVIATION FORUM AND ASCEND 2024*. 2024, p. 4250.

[12] Gabriel Jarry et al. “The Effectiveness of Large Language Models for Textual Analysis in Air Transportation”. In: *EasyChair preprints* (2024).

[13] Pinecone. *What is a Vector Database?* Accessed: 2024-09-13. 2023. URL: `https://www.pinecone.io/learn/vector-database/`.