Politecnico di Torino

Master's Degree in Aerospace Engineering

A.y. 2025/2026

Graduation Session March/April 2026

Automated Generation of Concept of Operations (ConOps) Through a LLM-Based Approach

Supervisors:
Elisa Capello
Gianni Pecora
Daniel Talavera Jiménez
Samuel Reskala Eguiarte

Candidate:
Sergiu Florea

Abstract

Concepts of Operations (ConOps) define the structured configuration through which complex systems execute operational objectives. In challenging operational environments, such as in the aerospace sector, the creation of a valid ConOps requires deep knowledge of subsystem interactions and strict syntactic rules.
The manual translation of high-level operational requirements into machine-readable configuration files is therefore a time-consuming, expensive process. This requires one or more highly skilled and certified engineers, and so it also introduces the risk of human error. As the system complexity increases, this process can become a significant bottleneck that can stall operations, which in the aerospace sector can cause significant costs.

In other sectors, advances in the use of Large Language Models (LLMs) suggest the possibility of partially automating this process, especially in helping to generating structured configurations from natural language requirements.
This thesis investigates the feasibility of employing a LLM-based approach to bridge the gap between high-level user intent and formally structured ConOps representation.

For this work, a stepped LLM architecture was developed, in which an operational description for a spacecraft mission in natural language is translated into a structured XML schema. This model is guided through a highly constrained prompt designed to enforce the correct structure of the XML. To improve syntactic comprehension of the user input and semantic correctness of the XML, Retrieval Augmented Generation (RAG) was added to provide contextual examples.
The validated XML is subsequently translated into executable, Python code, that strictly follows a predefined template, through the use of a deterministic, hardcoded module.

The final product is presented as a Python-based proof of concept and evaluated in terms of structural correctness and computational time, analyzing which models are best suited for this process.
Ultimately, this work establishes a baseline architecture for LLM-assisted configuration generation and provides a foundation for future AI assistant applications.

Table of Contents

Acronyms IV

List of Tables V

List of Figures VI

1 Introduction 1

1.1 Motivation . . . 1
1.2 Overview . . . 2
1.2.1 360™ . . . 2
1.2.2 Artificial Intelligence . . . 4
1.3 Outline . . . 7

2 Problem Breakdown 8

2.1 Mission . . . 8
2.2 Problem Statement . . . 9
2.3 Architecture Trade-off . . . 12
2.3.1 Monolithic Approach . . . 12
2.3.2 Stepped Approach . . . 13
2.3.3 Trade-Off . . . 14
2.4 Solution Implementation . . . 16
2.4.1 Syntax Recognition . . . 16
2.4.2 Code Generation . . . 19

3 System Architecture and Implementation 21

3.1 Syntax Recognition: First Architecture . . . 24
3.2 Syntax Recognition: Second Architecture . . . 31
3.3 Syntax Recognition: Final Architecture . . . 35
3.4 Syntax Recognition: Recap . . . 40
3.5 Code Generator . . . 42

4 Testing and Validation 45
4.1 Validation Objectives 45
4.2 Syntax Recognition 47
4.2.1 LLM Model Trade-Off 49
4.2.2 RAG Parameter Trade-Off 53
4.3 Code Generator 54

5 Discussion of Results 56
5.1 Project Outcome 56
5.2 Future Improvements 59

6 Conclusion 61
6.1 Overview 61
6.2 Future Work 62

Bibliography 65

Acronyms

ConOps
: Concept of Operations

AI
: Artificial Intelligence

ML
: Machine Learning

NLP
: Natural Language Processing

LLM
: Large Language Model

RAG
: Retrieval-Augmented Generation

CPU
: Central Processing Unit

GPU
: Graphics Processing Unit

RAM
: Random-Access Memory

List of Tables

4.1 Model performance comparison 50
4.2 RAG parameter performance comparison 53

List of Figures

1.1 Operation Structure . . . 2
1.2 Mode Structure . . . 3
1.3 Event Structure . . . 4
1.4 AI structure . . . 6

2.1 Operations Map . . . 10
2.2 Monolithic Architecture . . . 12
2.3 Stepped Architecture . . . 13
2.4 Syntax Recognition . . . 16
2.5 XML Schema pipeline . . . 17
2.6 Examples pipeline . . . 19
2.7 Code Generator . . . 20

3.1 Modular design of the Tool . . . 22
3.2 JSON Schema pipeline . . . 25
3.3 JSON examples pipeline enhanced by RAG . . . 26
3.4 Full JSON Schema pipeline . . . 27
3.5 JSON Schema pipeline enhanced by RAG . . . 29
3.6 Full JSON Schema pipeline enhanced by RAG . . . 30
3.7 XML Schema pipeline enhanced by RAG . . . 32
3.8 XML Examples pipeline enhanced by RAG . . . 33
3.9 XML Schema pipeline enhanced by RAG . . . 34
3.10 XML Schema pipeline . . . 37
3.11 XML Examples pipeline . . . 38
3.12 Full XML Schema pipeline . . . 38
3.13 XML Schema pipeline . . . 40
3.14 XML Examples pipeline enhanced by RAG . . . 41
3.15 Full Final XML Schema pipeline . . . 41
3.16 Full Final XML Schema pipeline . . . 42
3.17 Hard-Coded Code Generator Function pipeline . . . 43
3.18 XML to code translation . . . 43

Chapter 1

Introduction

1.1 Motivation

IENAI Space[1] is a startup located in Madrid, Spain, focused on building space mobility products and services for the next generation of small satellites, empowering a sustainable new space economy.
The company developed ATHENA, a highly compact electric propulsion family based on their proprietary electro-spray technology, uniquely engineered to meet the challenges of propelling spacecraft, further, for longer, and a suite of mobility software tools, 360™[2], that is used to provide advanced mission analyses.

IENAI Space is interested in rendering their proprietary software as user friendly as possible, streamlining workflows and cutting overheads and inefficiencies.
To do so they have spent considerable time developing several accessibility functions to streamline the development of spacecraft simulations.

This is the context in which this thesis was envisioned: it was decided to explore the feasibility of implementing AI assistants to help the user correctly generate the code used in the application, allowing the user to focus on the design of the simulation. This would increase the efficiency of the simulation stage of any mission, lowering costs and time needed to move from the design stage to the production stage.
For this thesis it was decided to focus on one of the more complex features of 360™, the behavior modeling for spacecraft.
In this thesis, the proof-of-concept developed will be indicated with the name of "Tool".

1.2 Overview

To better understand the content of this thesis, a short exploration of the development environment and the tools used is displayed in this section.

1.2.1 360™

360™ is an advanced mission analysis tool with cutting-edge space mobility analysis capability. It enables high-fidelity propagation of: Flight and Attitude dynamics, Spacecraft and System States and Operations. Coupled to a heuristics optimization algorithm it allows for a wide range of concurrent engineering and design functionalities at all stages of mission design.
360™ is purposely packaged as a python library, ensuring easy accessibility and seamless integration into diverse workflows, other industry-standard libraries and internal developments. The tool provides a robust set of mission and space mobility analysis and design features.
While the application focuses on a wide range of functionality, for the explicit purposes of this project, the main focus will be in exploring the behavior modeling that 360™ provides: the OpMachine[2].

OpMachine

The OpMachine, or operation machine, coordinates multiple operations to correctly model spacecraft behavior.
Each operation object is composed of a mode object and one or more event objects. This structure can be observed in Figure 1.1.

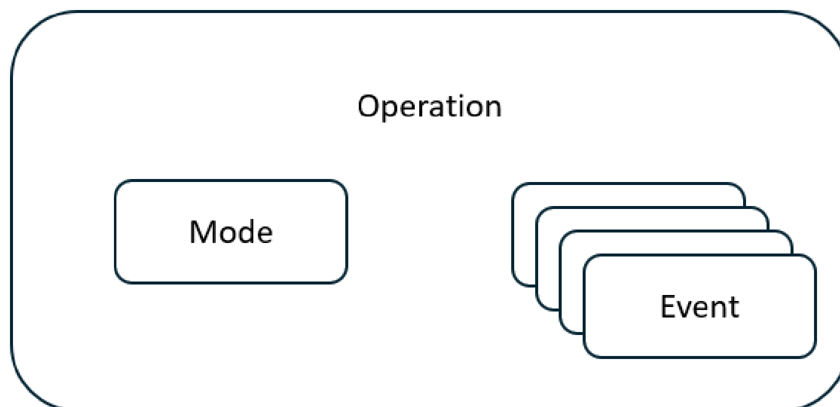

Figure 1.1: Operation Structure

The operation encompasses the behavior of the spacecraft during a specific time period of the mission. The behavior of the spacecraft, or the "what", is structured inside the mode object, while the specific time period of the mission, or the "when", is structured inside of the event.
The mode encompasses various sub-functionalities, but the most important for this project are the pointing function, the subsystem mode function and the thruster mode function.
The structure of the mode object can be observed in Figure 1.2.

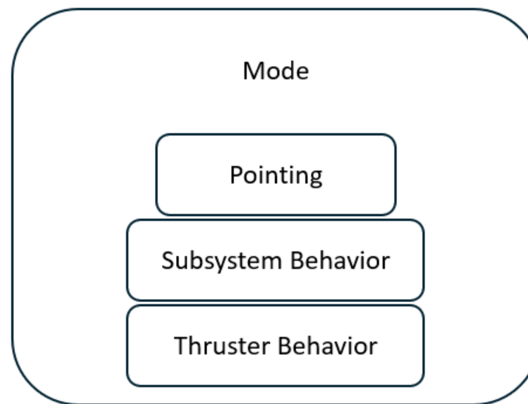

Figure 1.2: Mode Structure

The elements of the mode are functions that indicate the systematic behavior of the spacecraft:

- pointing: this function is divided in two main parts, the main pointing which points the pointer of the spacecraft, which can be either one of the body axis of the spacecraft or a specific subsystem, towards a specified target, and the secondary pointing, which tries to maximize a secondary direction throughout the orbit of the spacecraft without affecting the primary pointing. As it currently stands, there are five possible target functions and two possible secondary pointing functions.

- subsystem modes: this function applies to any subsystem on the spacecraft with the capability of consuming power. Currently, any subsystem of the spacecraft has three possible behaviors: it can either be on, off or idle.

- thruster modes: this function applies to the thruster subsystem of the spacecraft. Currently it has five available behaviors.

The event represents the logical operator that defines what conditions have to occur

to have a specific outcome.
The structure of the event object can be observed in Figure 1.3.

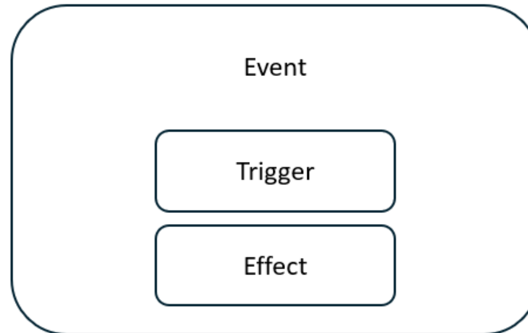

Figure 1.3: Event Structure

The event is often used as the switch between different operations, allowing the OpMachine to function correctly.
Its main elements are:

- trigger: defines the condition that must be met to achieve a particular outcome. Currently there are twenty five available triggers in the OpMachine.
- effect: defines the outcome of a specific trigger. Currently there are six available effects in the OpMachine. This function allows the jump between operations or the termination of the simulation.

Having briefly defined the structure of the OpMachine, it is clearly observable how complex it is. The presence of deep nesting and the high number of available functions renders this object the best suited to perform the research displayed in this thesis.

1.2.2 Artificial Intelligence

Artificial intelligence[3][4], or AI, is the ability of a computer system to simulate human thought, such as thinking, learning, comprehending and problem solving.
These inherent human abilities designed in the framework of a machine have made it possible to automate complex operations: with the ability of these systems to reason, recognize patterns and then make decisions, their inclusion in many sectors was quickly adopted.
For a use case such as the one explored in this thesis, the use of AI systems was the most appropriate. Dealing with the complexity of spacecraft operation is currently a bottleneck in many space missions, since it requires multiple highly trained

and experienced engineers that must correctly understand mission requirements, properly structure them in correct Concepts of Operations and finally translate those ConOps into finalized simulations to check their viability. The use of an AI system would greatly decrease the time needed to go from initial high level requirements to functional simulations, while also mitigating human error in the process.
Unfortunately not any AI system can accomplish this complex task, and since AI is an umbrella term that encompasses a huge variety of systems, it is important to narrow down the system family more applicable for this project.

The most important feature that the AI system must have to be compatible with the requirements of this project is the ability to learn from already existing data. This is required since simple systems that only follow programmed rules are not powerful enough to understand complex operations and extract high level requirements from simple natural language statements.
Therefore the focus shifts from the broad AI paradigm to the narrower Machine Learning paradigm.

Machine Learning systems[5][6] include every system capable of understanding expected behavior from existing data, and therefore able to correctly predict how to handle new data without the need for strict hard-coded instructions.
Given the impossibility of creating a rule-set for every possible natural language description of spacecraft operation, the employment of these systems was the correct choice. Another interesting quality is their ability to quickly adapt to new scenarios if properly conditioned.
Going even deeper, the strictest requirement for this project now is the capability of the system in understanding human language, strictly in written form, therefore the attention focuses on the narrower sub-sect of Natural Language Processing.

Natural Language Processing systems[7][8] use machine learning to properly parse natural language inputs and give coherent, understandable outputs. These systems are capable in extracting intent from natural language, covering the project requirement of extracting high level requirements and an early structure of the ConOps from detailed descriptions of the operations of the spacecraft.
Unfortunately, these systems often misinterpret the inaccuracies of human language, and given the context of this thesis, a misunderstanding in implied context from the user input could lead to the generation of a incomplete or incorrect ConOps.
To correctly generate the ConOps from a simple natural language input, a new system that could bridge the gap of NLP imprecision with ambiguous language.
Thankfully, with the advancements in AI development, a solution was found: the Large Language Models.

Large Language Models

Large Language Models[9][10], often referred to as LLMs, are the latest advancement in NLP. They are transformer-based models that are trained on large sets of data. While training these models, the huge sets of data is broken down in tokens, machine readable units that often represent single words or parts of words. Being made up of transformers, these models are also capable of self-attention: this means that they are capable of detecting dependencies and correspondences between different segments of the input[11]. Combining the self-attention with the large amount of data processed, these models are capable of understanding deeper connections between words, often being able to understand implied meaning in the user input. This characteristic lead to their huge popularity, and it is also the reason why they were chosen for this project.
To better understand the journey from generic AI system to LLM, their dependencies are displayed in Figure 1.3.

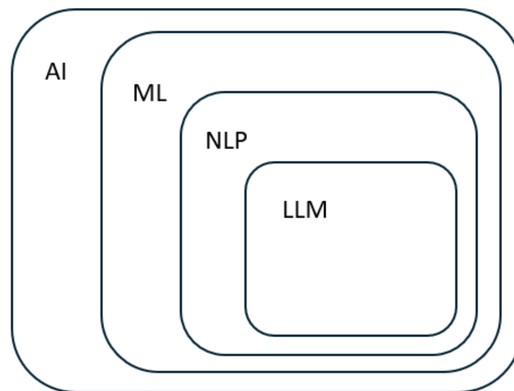

Figure 1.4: AI structure

Combining the intent understanding of the LLM with their ability to generate structured output inherited from NLP, the LLM becomes the best available AI system to use for this thesis. After choosing this paradigm, it was then decided to use pre-trained models to focus more on the design and development of the architecture of the Tool. The main strength of using a pre-trained model is their availability and ease of integration, allowing for fine-tuning to further increase reliability.
A downside of using LLMs is that, on low-end hardware, they struggle often with maintaining attention on the prompt. This problem derives from the unavailability of RAM, and the slower computational speed of CPUs compared to GPUs, which requires the LLM to reduce their precision to function. This loss of attention is noticeable in highly complex prompt, which clutter the available memory for the

model and often leave it unable of correctly parsing all of the user input or any rules loaded into the prompt.
To alleviate this problem a reduction of the prompt would be necessary, to free up the memory and allow the model to function properly, but most of the time, this is not possible, since vital rules and examples often are included in the prompt to direct the LLM in its generation.
To reduce prompt clutter without risking the incorrect generation of the output because of missing context, the implementation of Retrieval-Augmented Generation was explored.

Retrieval-Augmented Generation

Retrieval-Augmented Generation[12], often referred to as RAG, supports the LLM by providing it with external documentation. Before injecting any extra documentation in the prompt, the RAG generate a vector space and fill it up with different snippets of the documentation, then they compare each snippet with the user input and select a set amount to be injected into the prompt. This enhanced pipeline is capable of reducing the clutter in the prompt, especially from long examples that might not always be contextual with the user input, allowing the LLM to better focus its attention on the user input, and therefore on generating correct ConOps.

Having defined both the operational environment of the Tool and the models that will be used to build it, it is now time to explore the Tool in all fo its aspects.

1.3 Outline

The thesis is organized as follows:

- **Chapter 2:** declaration of the problem statement and initial architecture trade-off. This chapter establishes the starting point of the project.
- **Chapter 3:** a full exploration of the design history of the project, with the implementation of the final version of the Tool.
- **Chapter 4:** an exploration of the validation strategies applied to test the Tool.
- **Chapter 5:** a discussion of the findings from the tests with a look at possible future improvements to turn this proof-of-concept in a full-fledged application.
- **Chapter 6:** closing remarks.

Chapter 2

Problem Breakdown

2.1 Mission

The configuration and definition of a Concept of Operations (ConOps) for complex operations is a time-consuming, manual process that requires deep domain expertise. This process is often a bottleneck in deploying or re-tasking systems efficiently.

This thesis proposes the development of a novel system that leverages a Large Language Model (LLM) to bridge the gap between high-level user requirements, expressed in *natural language*, and the structured, machine-readable ConOps in the 360™ environment. By training or coordinating the LLM on a corpus of existing tutorials, system documentation, and operational examples, the proposed system will act as an intelligent assistant, automatically generating the required ConOps.

This thesis will explore the feasibility of this approach using Python and will outline a path from a foundational proof-of-concept to a more advance, fine-tuned implementation.

Modern operational machines, from industrial robots to complex software systems, rely on precisely defined ConOps to function correctly.
The manual authoring of these ConOps documents or configuration is a significant challenge: it demands specialized knowledge of the machine's subsystems and their specific command syntax. An engineer must manually translate the abstract operational goals into a series of exact, low-level commands.
This process is not only slow but also prone to human error, which can lead to inefficient operations or even system failure.

For the specific environment of 360™, the creation of a ConOps requires, in

addition, the knowledge of its framework and the correct use of its functions, which can further lengthen the time required to create a functioning file, and increase the chance of human error.

2.2 Problem Statement

This thesis aims at the automation of the ConOps by creating a Python-based application that uses a LLM.
The system will work as follows:

- **Requirement Acquisition**: an operator or engineer provides the desired operational requirements in simple, *natural language*.
- **Contextual Mapping**: the system interprets these requirements in the context of the specific operational machine.
- **Model Synthesis**: the LLM, primed with knowledge of the subsystems of the spacecraft and command structures. generates the complete, syntactically correct ConOps.
- **Semantic Parsing:** This ConOps is then translated into Python code ready to plug into 360™ and use.

To illustrate the complexity of this translation, a representative mission scenario, moving from a natural language prompt to a 360™-compatible Python script, is analyzed as follows.

User prompt:

*"The spacecraft shall keep its orbit is semi-major axis higher than 6780 Km through the use of its thrusters.
The simulation shall end on 01/02/2030 at 12:00."*

Considering how 360™ handles operations by dividing them into Modes and Events, the LLM should be able to identify this structure in the prompt, just like the following:

*"The spacecraft shall keep its orbit is semi-major axis higher than 6780 Km through the use of its thrusters.
The simulation shall end on 01/02/2030 at 12:00."*

Where Modes are expressed in Green while Events in Blue.
The LLM must be able to contextualize and understand the user's needs: a simple dual-state configuration.
This configuration accounts for a Nominal (Idle) state and a Thrusting (Active) state.
The transition between these two states is governed by the Event, which establishes a link between the semi-major axis of the orbit and the required operational state.
This capability to synthesize implicit states from explicit goals is the core functional requirement of the proposed tool.

This mapping can be visualized in Figure 2.1, which follows the same color pattern.

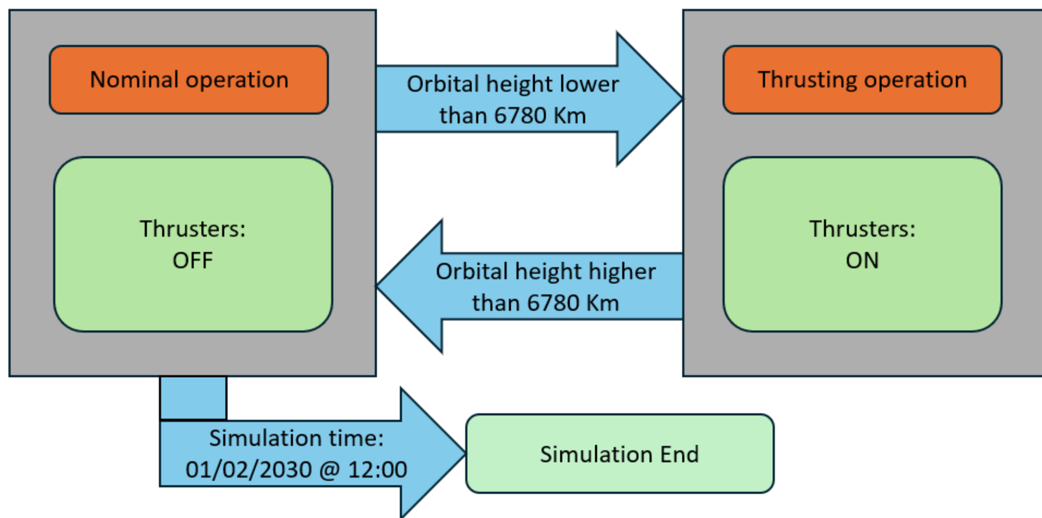

Figure 2.1: Operations Map

From this mapping, the LLM must create a machine-readable model, to help itself in the later process.

Finally the Model shall be translated in Python code compatible with the 360™ environment, such as the following:

```
#Nominal Operation

nominal_mode = Mode(systems_mode=[sc.sc_sys("PROP").Off()])
nominal_event = Event(OnSemiMajorAxis("<" 6780*km), ToOp("Thrusting"))
epoch_event = Event(AtEpoch("2030-02-01T12:00:00"), TerminateSimulation())

nominal_operation = Operation("Nominal", nominal_mode, [nominal_event,
    epoch_event])

#Thrusting Operation

thrusting_mode = Mode(systems_mode=[sc.sc_sys("PROP").Thrust()])
thrusting_event = Event(OnSemiMajorAxis(">" 6800*km), ToOp("Thrusting"))

thrusting_operation = Operation("Thrusting", thrusting_mode, [thrusting_event])

#Building the ConOps

om = OpMachine(nominal_operation, thrusting_operation)
```

It is advisable for the LLM to be capable not only of understanding the user's intent, but also of identifying possible instabilities in the model and fixing them.

This can be clearly seen in the part of the user prompt that states *"semi-major axis higher than 6780 Km"*: this creates an instability in the trigger that switches between the operations, since, if both operations had the same threshold in their Event, the simulation would likely encounter a high-frequency oscillation between the states that can lead to buffering or simulation instability.

This problem can be solved by adding a buffer to the user input, where the LLM predicts the instability and corrects it accordingly, such as in this case where a buffer of 20 km was added to the trigger of the Thrusting Event.

Ultimately, the challenge lies in ensuring that the LLM maintains adherence to the structure of 360™ while accurately understanding even the nuances of natural language.
This requirement must be explored starting with the choice of the LLM architecture.

2.3 Architecture Trade-off

Having established the necessity of generating a Python output, structured to be compatible with 360™, from the user's natural language input, the architectural strategy for this translation must be evaluated.
This can be done considering two possible paradigms:

- A **monolithic approach**, in which the input is translated directly into Python code.
- A **stepped approach**, in which the input is first mapped to an intermediate structured representation and later transforms it into Python code.

2.3.1 Monolithic Approach

The Monolithic Approach treats the LLM as a black-box system.
It can be best described using Figure 2.2.

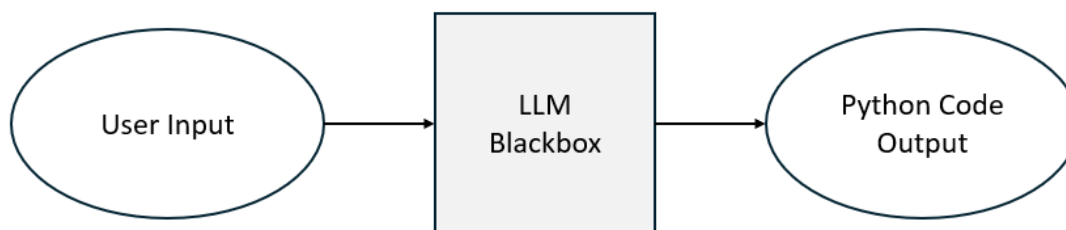

Figure 2.2: Monolithic Architecture

This architecture centralizes both syntax recognition and code generation within a single probabilistic component. Consequently structural correctness and formatting constraints rely entirely on the LLM.

Advantages

- **Simpler structure**: the design being self-contained makes system creation and maintenance straightforward.
- **Higher flexibility**: without a strict schema to follow, other than the allowed and required Python functions of 360™, the LLM can interpret the user input much more freely and handle edge cases better.
- **Reduced latency**: the absence of intermediate validation layers reduces processing time, minimizing user wait time.

Disadvantages

- **Harder debugging**: the design being self-contained makes it difficult to determine whether failures originate in the syntax recognition or in the code generation, complicating development.
- **Inconsistency**: without a strict schema to follow, the likelihood of LLM-generated syntax errors or inconsistencies in handling unusual input increases.
- **Reduced validation**: the absence of intermediate validation layers renders verifying correctness challenging, increasing the complexity of future updates.

2.3.2 Stepped Approach

The Stepped Approach divides the system in multiple connected sub-systems.
It can be best described using Figure 2.3.

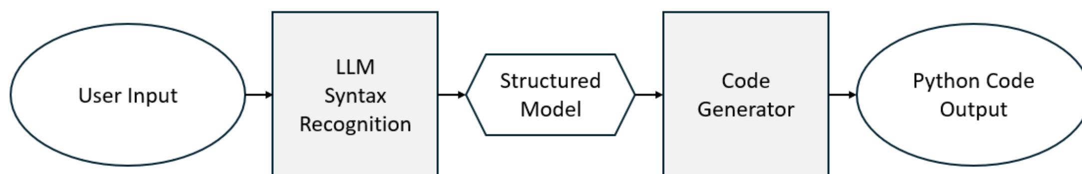

Figure 2.3: Stepped Architecture

This architecture introduces a clear separation between syntax recognition and code generation. The intermediate structured representation acts as a controlled interface, allowing structural correctness and formatting constraints to be enforced deterministically rather than relying only on the LLM.

Advantages

- **Increased validation**: having intermediate validation layers allows easier verification of the system, decreasing the complexity of future updates.
- **Easier debugging**: separating the syntax recognition from the code generation allows the better understanding of the origin of a bug, reducing debug complexity and time.
- **Higher consistency**: having a strict schema to follow, the LLM is less likely to hallucinate and generate incorrect structures.

- **Easier updates**: having an independent, deterministic code generator allows for easier updates and overhauling without changing the core functionality of the LLM.

Disadvantages

- **Higher complexity**: adding a strict schema increases the complexity of the system prompt of the LLM, while rendering the code generation to be deterministic increases development time.
- **Higher latency**: the presence of an extra checkpoint increases processing time.
- **Restricted flexibility**: with a strict schema to follow, the model can be too rigid and find difficulties handling complex user input and edge cases.

2.3.3 Trade-Off

At a first glance, the monolithic approach seems more intuitive and its simplicity is quite alluring.
Its single-stage architecture reduces development time and cost, and in a scenario where output correctness and consistency is loosely constrained, such a design might actually be the best choice.

Unfortunately, considering the importance of reliability for this thesis, this model does not allow for direct control over intermediate determinations and nullifies the verification.
This means that the model might recognize the correct structure of the Python output as a mere suggestion instead of an enforced rule.

This risk cannot be ignored, since incorrect Python code will not compile in the 360™ environment, going against the purpose of this thesis.
Structurally correct outputs, with incorrect variables are even more dangerous.
In the aerospace sector, even an incorrect command can cause failures, if not the total loss of the spacecraft in worst-case scenarios.

The stepped approach addresses this concern by introducing a checkpoint, which acts as an extra constraint between the natural language input and the Python code output.
By separating syntax recognition from code generation, the LLM is "lightened", since it does not have to transition between two languages that have lose boundaries, and it can simply fill a structured schema.

This also affects the code generation, as it becomes fully deterministic, lowering the testing and validation difficulties and corresponding development costs.
Lastly the checkpoint can be easily tested and validated, lowering future development cost.

Unfortunately these benefits come with increased complexity, which would increase development time, and the need for a heavily structured prompt requires innovative prompt additions to give the LLM the necessary flexibility.
The presence of the checkpoint also increases computing time and can have negative effects for user comfort.

This trade-off can be then viewed as a tension between flexibility and controllability.
For simple systems which do not require heavily structured outputs, the monolithic approach shines through, while for more strictly structured outputs, combined with the necessity for consistency and reliability, the stepped approach is the correct choice.

Considering, again, the strict requirements of ordered structure of 360™, and the need for reliability and consistency in the final output, the choice becomes obvious: the correct architecture for this project is the **stepped approach**.

2.4 Solution Implementation

Ultimately, an exploration of the final configuration chosen for the application, a deep dive into its main components (the Syntax Recognition and the Code Generation) will allow for better insight on the decisions made throughout this work.

This section will also help in better understanding the stepper architecture.

2.4.1 Syntax Recognition

The syntax recognition segment of the application is responsible for interpreting the user input and extracting the context for the modes and events, that get later translated into a structured XML object that is compatible with those created in the 360™ environment.

This segment also contains the LLM enhanced with RAG, rendering it the stochastic element of the application, and therefore harder to develop and maintain.
The architecture for the syntax recognition is displayed in Figure 2.4.

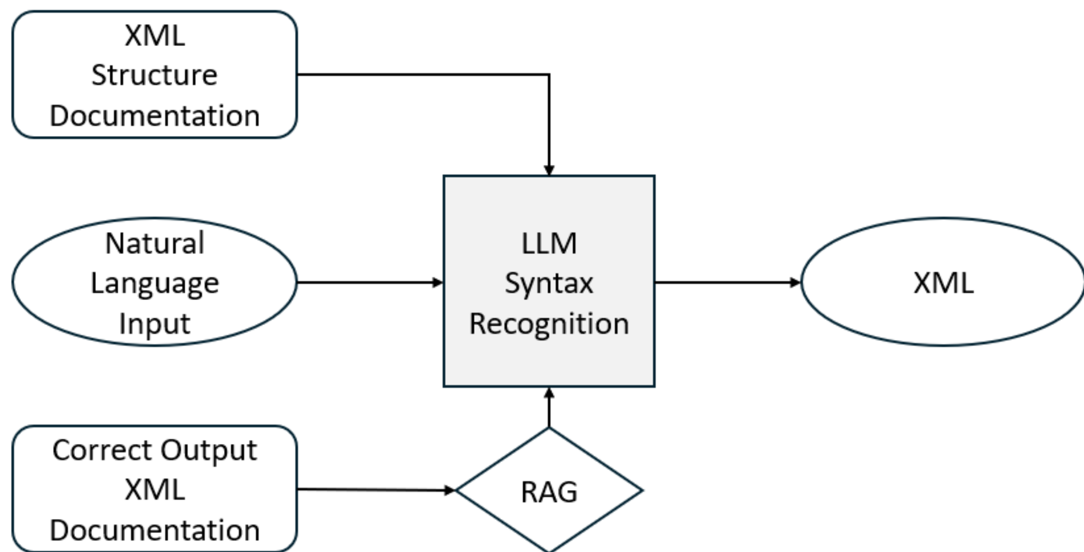

Figure 2.4: Syntax Recognition

For this thesis the chosen structure of the output was XML, to be compatible with the XML generated in 360™.

This made it possible to have a reliable set of correct XML objects readily available to test and validate the output of the LLM, with the downside of highly restricting its flexibility and complicating the development of the tool.
Having a 360™-familiar output is further advantageous, since any developer at the company IENAI Space will be able to readily understand the output and quickly attend to any possible problems in the future.

The structured XML schema is delivered to the LLM through both its system prompt and the user input prompt: the global rules, which govern the base behaviors of the LLM and suggest it to analyze the natural language input and turn it into an XML object, are delivered through the system prompt, while the ordered structure of the XML schema is injected in the user prompt before it is delivered to the LLM (Figure 2.5).

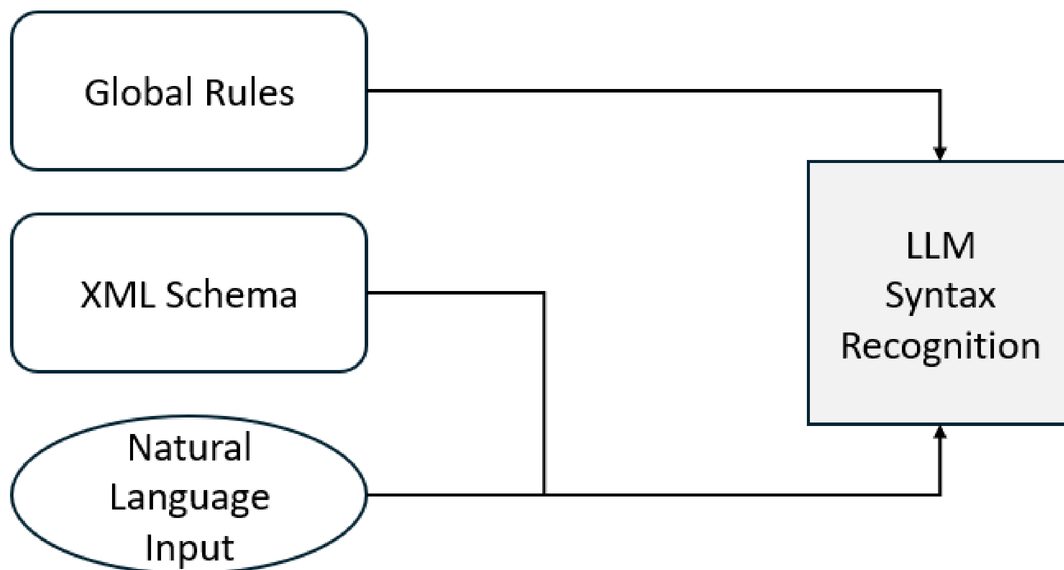

Figure 2.5: XML Schema pipeline

Doing this ensures that the LLM always considers the full context of the XML schema when generating the correct XML from the user input, since if the XML schema were put in the system prompt the LLM would follow it more as a mere suggestion rather than an enforced rule.

This behavior was observed through experimentation, in which different LLMs were loaded with different prompt configuration.
The most notable outcome was the "lack of interest" in the structure of the XML

and in certain XML tags by the LLM whenever the schema was loaded in the system prompt. This can be explained by the complexity of the XML structural schema. Given that most of the models used were small compared to readily available commercial LLMs, and thus have a limited computational capability. This is due to the available hardware, which makes the LLMs used very slow. The latency combined with the small size makes it harder for the LLM to correctly follow the system prompt while analyzing the user prompt, and therefore most of the XML schema gets "forgotten" when the LLM is looking at the examples and the user input, consequentially generating an incorrect XML object.
It was finally decided to load the schema together with the user prompt to avoid any inconsistency in the output.

This decision was allowed since the LLM is reinitialized after every call, negating the possibility of context overflow. If the system was not reinitialized, for every subsequent call the user prompt would be flooded with the same XML schema.

Having the proper XML schema helps the LLM in understanding the correct structure it must replicate, it is still missing any help in recognizing the user input, especially in understanding certain complex or technical segments.
To solve this a document filled with various correct example XML together with their natural language input was created, but the sheer size of the XML and the amount of examples rendered the simple upload of this document unadvisable, to not reach the context limit of the LLM and to not increase the computational time.

To solve this it was decided to implement a Retrieval Augmented Generation (RAG) method. It embeds the examples documents and then searches for a few examples, relevant to the user input, to help the LLM better understand edge cases and complex XML tags (Figure 2.6).

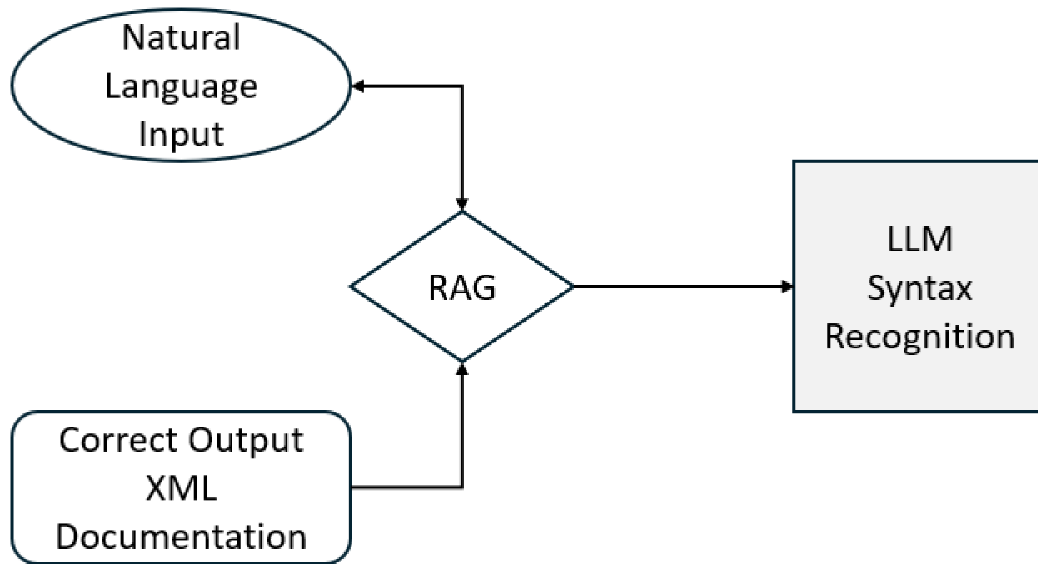

Figure 2.6: Examples pipeline

This architecture that uses external documentation is also beneficial since it renders the application modular, allowing for easier maintenance and updates: after any development in the 360™ environment, and especially after the addition of new features for the OperationMachine, any developer can simply update the XML schema document by adding the new XML tags, and also creating new examples in the example document, in which they illustrate which natural language input might require that specific new tag.

2.4.2 Code Generation

The code generation segment of the application is responsible for turning the XML from the syntax recognition into the ready to use, 360™-compatible Python code.

This segment is fully deterministic, rendering it the "unintelligent" part of the work, and therefore easier to update and maintain.
The architecture for the syntax recognition is displayed in Figure 2.7.

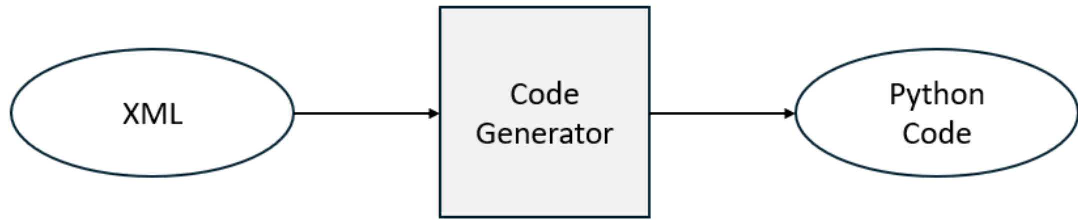

Figure 2.7: Code Generator

For this thesis, the Python output had to be compatible with 360™, and structured strictly to follow the OpMachine code schema.
Being deterministic, the testing and validation was also faster and easier.
Using both specialized code written in 360™ and the XML generated by said code, the testing phase was simply working in reverse, inputting the XML and comparing the output code with the original hand-made code.

A beneficial side effect of this work is that, being modular, this function can be used in different context throughout 360™, bridging the previous gap of 360™, the inability to regenerate correct code from an XML.

Chapter 3

System Architecture and Implementation

Having analyzed the problem and defined its complexity, it is now possible to focus more on the inner workings of the OpMachine Tool, specifically on the different stages of implementation.
Having also defined the architecture most reliable and compatible with the desired solution, it is important to differentiate between the syntax recognition and the code generator, in both their conceptual design and eventual developed structure.

Before exploring the different architectures designed and developed, it is important to define the environment in which the Tool was developed.
It was decided to use Ollama[13], an open-source application which allows for easier integration of LLMs into the python environment in which the Tool was coded.
The Ollama application allows the user to download and run locally many commercially available LLMs without the need for a subscription, and allowing for heavy customization and tinkering with the system prompt of the LLM.
Ollama was selected because of its simplicity and ease of access, allowing for a faster implementation of the Tool, and therefore allowing for more research into the viability and the usefulness of the Tool.
The Tool was also designed to be modular, allowing for easy plug-and-play functionalities. To do so it was decided to keep all documentation in the form of plain text, since it is readable by any LLM with a chat functionality.
it is clear to see the usefulness of this design: being modular, the Ollama Application block can be easily swapped with any other desired Application, whether already available or custom made, without having to rewrite the full implementation of both the structure of the ConOps and its functional example library.
The modular design of the Tool is displayed in Figure 3.1.

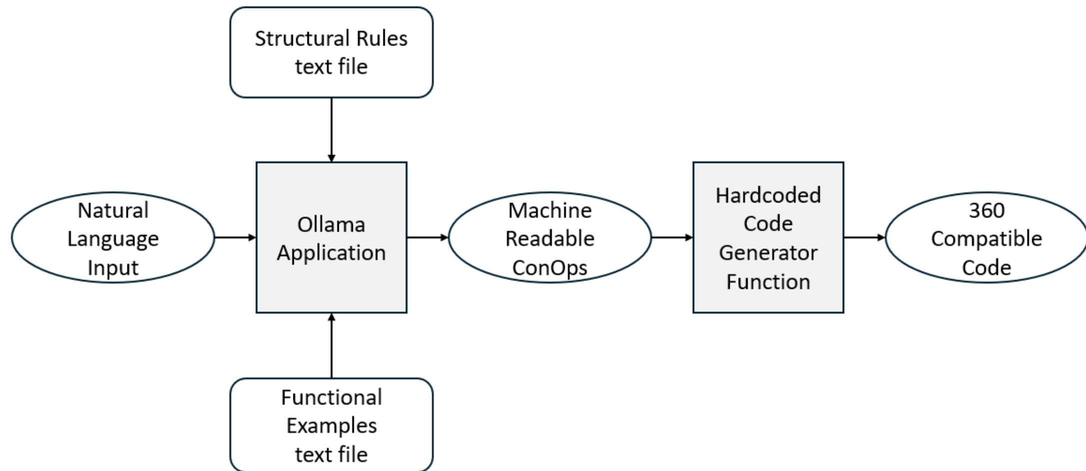

Figure 3.1: Modular design of the Tool

One of the set backs of this design is the incompatibility with any Application that does not support LLMs with a chat function, in which the text based approach no longer applies.
Another set back is the need to re-test and re-validate the Syntax Recognition block whenever a new Application is used, to make sure that the new LLM is compatible and capable of understanding the instructions written in the text files and capable of creating the correct Machine Readable ConOps. This is an important check to perform since the hard-coded Code Generation function was developed following a specific and rigid structure, and even a small deviation in the expected Machine Readable ConOps can completely void the final 360™ compatible code.
From the Ollama application it was also decided to use several different LLM models. Given the hardware availability of only CPUs and their low yield for LLMs, unfortunately only small models could actually be tested and analyzed.
The main models used ranged from 1.5b to 8b, with some attempts to use larger models of 14b.
The following is a list of the models used for this project:

- deepseek-r1:1.5b
- deepseek-r1:7b
- gemma3:4b
- qwen3:4b
- qwen2.5-coder:7b

- ministral-3:8b
- llama3.1:8b
- phi4:14b

Out of all of these, the most used one was llama3.1:8b, since it worked excellently with only CPU while yielding correct results reliably.
Most of the other models were tested but lacked the agility of llama3.1:8b when it comes with working on CPUs, answering the user query after a wait time that often peaked at one hour, whilst smaller models than 8b did not give reliable results and were often incorrect in their output, with either the structure of the ConOps or with the syntax recognition of the user input.

3.1 Syntax Recognition: First Architecture

After the first round of research it was decided to design the first iteration of the Syntax Recognition of the Tool to output the ConOps in the form of JavaScript Object Notation, or JSON. This choice was guided significantly by the built in function of the Ollama suite to request structured outputs directly from the LLM in the form of JSON files[14].
The JSON format was also very interesting to analyze because it is easy to understand for humans while also being parsed by machines with no problem. This could allow easy validation of results through it is more intuitive structure, minimizing testing time.
To better understand the structure of the JSON schema used for this design, the following is a simplified example of the OpMachine in this format:

```
{
  "OpMachine": {
    "name": "nominal",
    "mode": {
      "pointing": {
        "primary_pointing": {
          "pointer": "payload",
          "target": "nadir"
        },
        "secondary_pointer": "maximize_sunlight"
      },
      "system_mode": [
        {
          "name": "payload",
          "sys_mode": "on"
        },
        {
          "name": "comms",
          "sys_mode": "off"
        }
      ],
      "thruster_mode": {
        "name": "thruster",
        "thr_mode": "off"
      }
    },
    "event": [
      {
        "trigger": "atApoapsis",
        "effect": "terminateSimulation"
      }
    ]
  }
```

```
34 }
```

it is clear to see that the OpMachine requires a pretty nested and complex structure, leading to the creation of an equally nested and complicated JSON schema.
it is also easy to read and understand, allowing for a faster check on the correctness of the generated ConOps of the LLM.
A slight drawback of this architecture is the high restriction imposed on the LLM, causing it to lose a bit of understanding of the user input if not phrased appropriately, or even certain implied concepts, such as the presence of a standard operation called nominal or even the correct jump between operations.
The pipeline for the structured JSON schema using the Ollama structured output function is displayed in Figure 3.2.

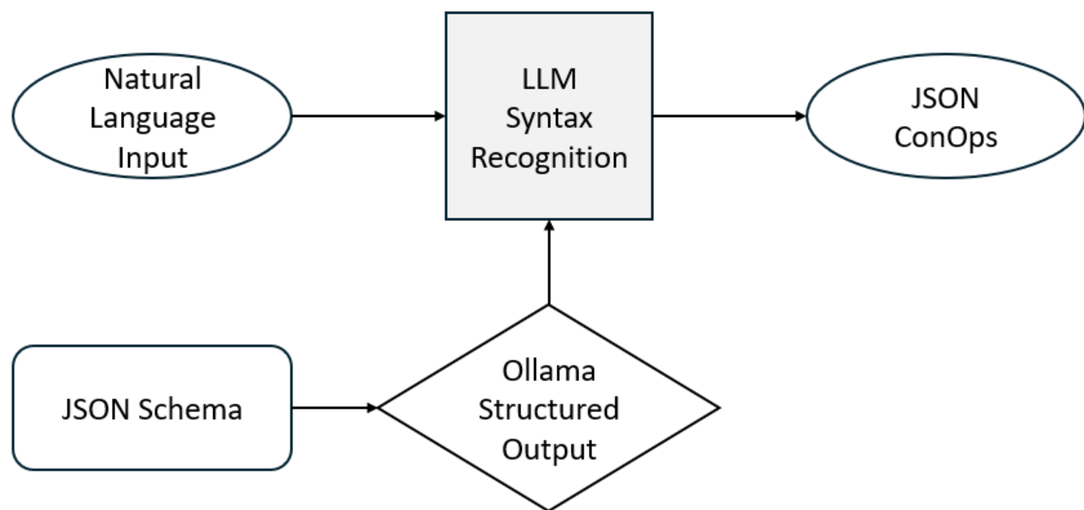

Figure 3.2: JSON Schema pipeline

To compensate the high rigidity of the a set of examples was created. These examples were composed of example user inputs matched with the desired JSON output.
Originally, these examples were going to be uploaded to the LLM through the user prompt, before the actual user input, to help the LLM in better understanding user intent while also consolidating the required JSON schema more so than the JSON schema uploaded through the structured output function of Ollama already does.

While creating this examples a new problem arose, the sheer amount of examples needed to cover all possible functionalities of the OpMachine, which could lead to the overstress of the LLM, by either consuming too many tokens, diluting the actual user input and causing the LLM to hallucinate or stitch together example outputs, and the overuse of computational power, leading to an increase of the computational time needed.
Therefore it was decided to explore the implementation of a RAG. This meant that only the examples that were relevant to the user input, and therefore only the examples that showed the correct use of desired functions, would be added to the prompt, lowering the used tokens and the computational power needed.
The pipeline for the examples delivery is seen in Figure 3.3.

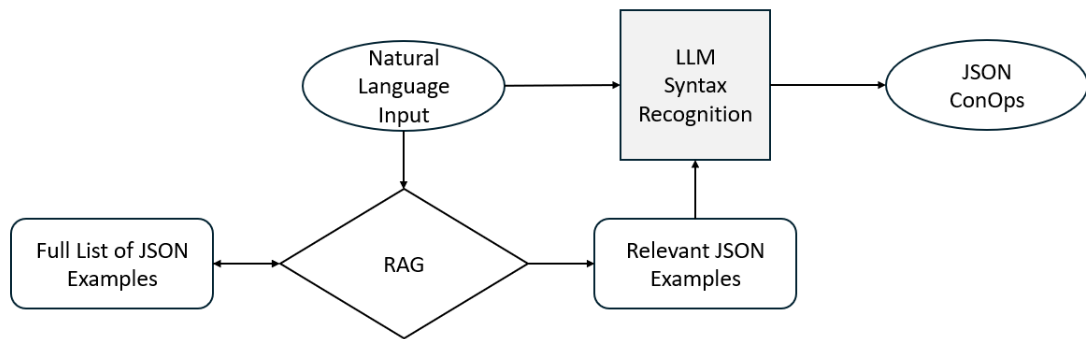

Figure 3.3: JSON examples pipeline enhanced by RAG

Having defined both elements and having developed them, after a few tests it was confirmed that the simple Ollama structured output could not work without giving the LLM some guidance in understanding user intent for more complex operations, so the two elements were merged into a single system.
The system prompt of the LLM was also designed to make it understand the role of each segment, telling it to strictly output JSON schema while referencing the example outputs to create a satisfactory ConOps.
The full pipeline for the first design iteration of the Tool can be seen in Figure 3.4.

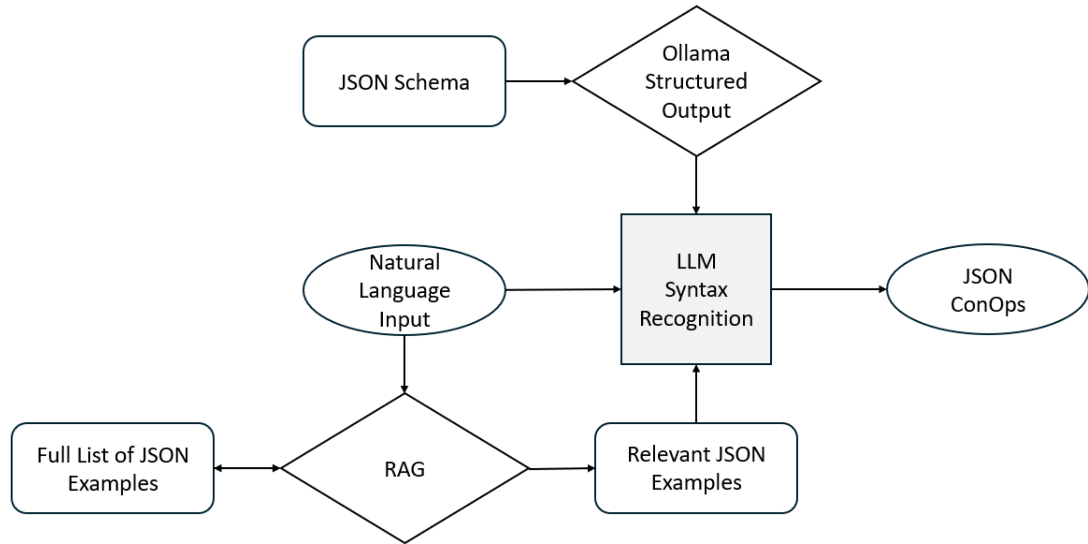

Figure 3.4: Full JSON Schema pipeline

Unfortunately, given the complex and nested structure of the OpMachine, with all the multiple allowed built-in functions, the structured output function of Ollama proved to be too restricting for the LLM, even with the support of the relevant examples.
After several tests, the computational time necessary to output the ConOps reached peaks of one hour, completely unacceptable, whilst the actual JSON file created often showed multiple errors. The LLM was not capable of fully understanding the user intent, for example with a simple user prompt such as "the spacecraft used its payload pointed towards Earth", the JSON file would often display in the pointing object either a different pointer, such as another part of the spacecraft or a vector, or display the wrong direction, seemingly not knowing that by pointing towards the Earth the required target was Nadir.
This seemed to be common between the multiple models that were tested, going from smaller models of 3b or 4b, that often created almost completely nonsensical JSON ConOps, to medium models of 14b, which still struggled with user intent and sometimes hallucinated operations when they were not mentioned, or completely skipped important operations required by the user.

The culprit of this malfunction turned out to be the structured output function, which worked perfectly fine with simple JSON schema, but could not parse correctly the required JSON structure of the OpMachine.
Looking for another way to feed the correct structure to the LLM, it was decided to implement and test a RAG approach. This would lighten the restraints on the

LLM, allowing for more creative thought, while also ensuring the correct generation of the JSON ConOps.
The need for a RAG approach mirrors the logic behind its choice in the example injection: having too many rules for such a complex structure could consume more tokens than necessary while also risking on loosing the attention of the LLM on the user input.
The choice of using RAG was also allowed by the malleability of the JSON ConOps. In fact, the JSON file must include all of the needed functionalities, such as the mode, the event and the operation name, but not in a strictly set order. Given its ease of machine parsing, it is possible to have "disordered" JSON files that are semantically correct and still be able to parse them in the Code Generation block with little to no problem.
To better understand how the RAG handles the different elements of the JSON schema in the structural schema file, the following is a short example of how each element was implemented:

```
1  {
2  ELEMENT: Mode
3  TYPE: Object
4  PARENT: Operation
5  CHILDREN: [
6      Subsystem_Mode,
7      Pointing,
8      Thruster_Mode
9      ]
10 DESCRIPTION: "It represents the Mode of the spacecraft..."
11 }
12
13 {
14 ELEMENET: Subsystem_Mode
15 TYPE: Array
16 PARENT: Mode
17 SCHEMA: [
18     {
19         "name",
20         "sys_mode":["on","off","idle"]
21     }
22     ]
23     DESCRIPTION: "It represents the Mode of the subsystems of the spacecraft...
       "
24 }
```

The RAG will assess similarity between the user prompt and the description each element to decide which ones are more relevant and therefore needed. The examples given to the LLM by the example injection pipeline will also help the LLM better understand how the different elements are linked, supporting the parent and child sub-element of every single structural rule written.

Having selected a new schema delivery other test were performed, showing improved performance in computational time needed and in result correctness, allowing for the correct creation of all the necessary elements needed in the ConOps.
It was also noticed how the order of the elements in the ConOps was not always consistent, just as theorized in the design phase of the work.
The new pipeline for the JSON Schema using RAG can be seen in Figure 3.5.

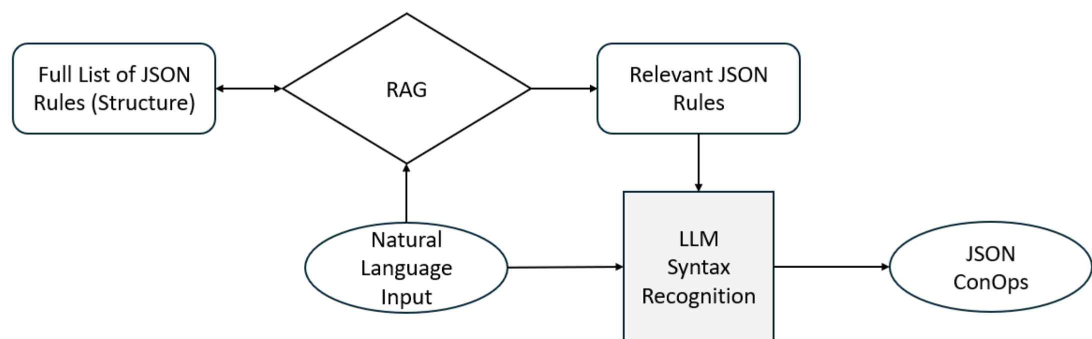

Figure 3.5: JSON Schema pipeline enhanced by RAG

Combining the new schema injection with the old example injection, the final architecture for a JSON ConOps was created.
The full pipeline for the second design iteration of the Tool can be seen in Figure 3.6.

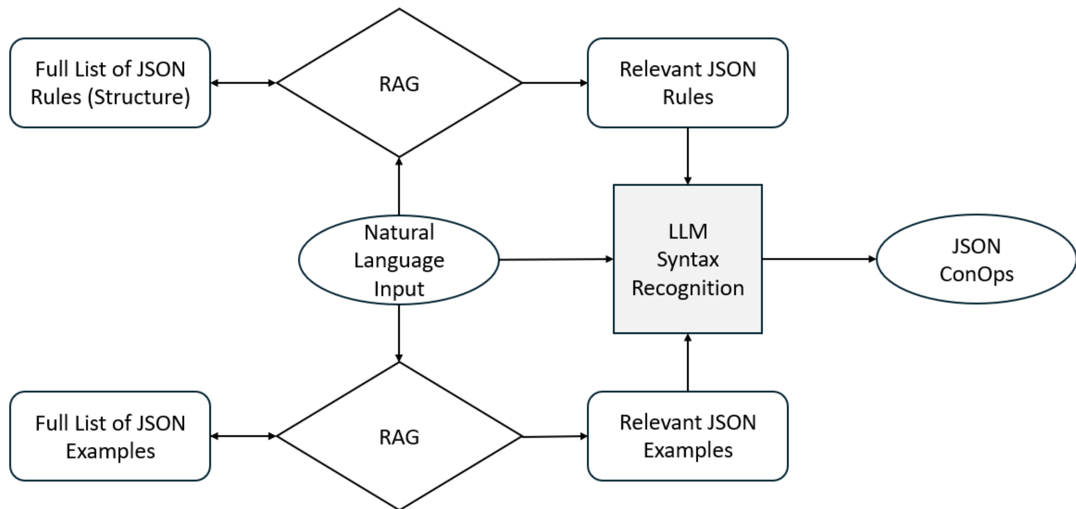

Figure 3.6: Full JSON Schema pipeline enhanced by RAG

Unfortunately, this solution was quickly abandoned, and further testing was not performed. The main outcome of the few rounds of testing were the improvement in structural correctness compared to the previous version, and a noticeable decrease in the wait time from the initial sending of the user input to the response of the LLM.
This architecture was promptly replaced by the next iteration shown in the following section.

3.2 Syntax Recognition: Second Architecture

It was decided after several meetings to generate the ConOps in the form of an XML, to match the format used in 360™. Since 360 generates an XML from Python code, it seemed more fitting to generate the ConOps in the same format, to be more standardized.
This shift did not seem too complex to accomplish, since the main difference would have been creating a text file with the definition of all the used tags for the OpMachine and also updating the examples with correct XML representations of the desired output.

Considering this, it was decided to also keep the same architecture used with the JSON file type. This choice was made because of the shift of attention from the architecture to the conversion to a different format. The process of converting both the structural rules file and the example file was quite time consuming, effectively halting the development of the Tool for quite some time.
To better understand how the structural rules are handled by the RAG, a short representative example of sample rules is shown as follows:

```
1  {
2  ELEMENT: Mode
3  XML_TAG: <Mode>
4  PARENT_TAG: <Operation>
5  CHILDREN_TAGS: [
6      <Subsystem_Mode>,
7      <Pointing>,
8      <Thruster_Mode>
9      ]
10 DESCRIPTION: "It represents the Mode of the spacecraft..."
11 RUELS:
12 1. This tag MUST contain its CHILDREN_TAGS in the set order,
13 2. This tag CANNOT be self-closing.
14 3. This tag MUST be included
15 }
```

The RAG compares the user input with the description of any element, selecting the most appropriate needed and then injecting only the correct elements in the LLM.
The XML structure pipeline using RAG is displayed in Figure 3.7.

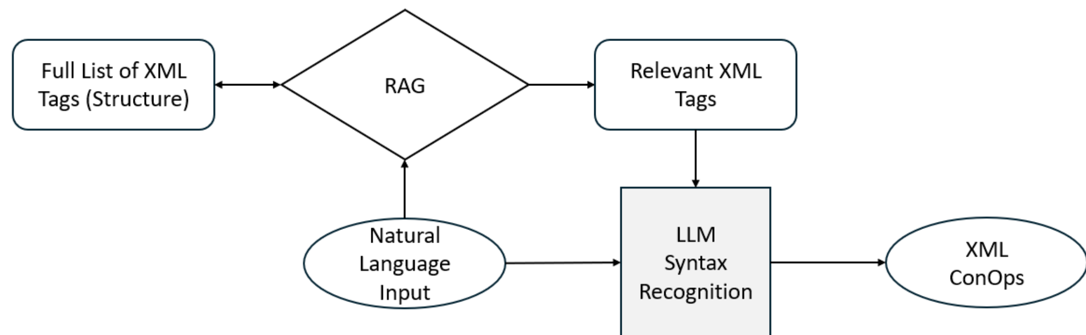

Figure 3.7: XML Schema pipeline enhanced by RAG

Even the example delivery pipeline was copied from the previous iteration, updating the example output with the correct XML structure. To better understand this the following is a simplified example sample:

```
1  INPUT: 'The spacecraft shall point its payload towards the Earth while keeping
       its comms idle. when it reaches the apoapsis the simulation must end.'
2  OUTPUT:
3  <OpMachine>
4      <Operation name="NOMINAL">
5          <Mode>
6              <Subsystem_Mode>
7                  <Command subsystem="payload">
8                      <command cmd="on"/>
9                  </Command>
10                 <Command subsystem="comms">
11                     <command cmd="idle"/>
12                 </Command>
13             </Subsystem_Mode>
14             <Pointing>
15                 <PrimaryPointing>
16                     <pointer name="payload"/>
17                     <target name="nadir"/>
18                 </PrimaryPointing>
19                 <SecondaryPointing>
20                     <MaxSunlight/>
21                 </SecondaryPointing>
22             </Pointing>
23             <Thruster_Mode thruster="thruster">
24                 <command cmd="off"/>
25             </Thruster_Mode>
26         </Mode>
27         <Event>
28             <Trigger>
```

```
29                  <AtApoapsis/>
30              </Trigger>
31              <Effect>
32                  <TerminateSimulation/>
33              </Effect>
34          </Event>
35      </Operation>
36  </OpMachine>
```

The RAG will compare the INPUT section of the example with the provided user input, to not lose meaning and tokens with the XML taking up most of the sample, and then injecting the full sample straight into the LLM.
The XML example delivery using RAG is displayed in Figure 3.8.

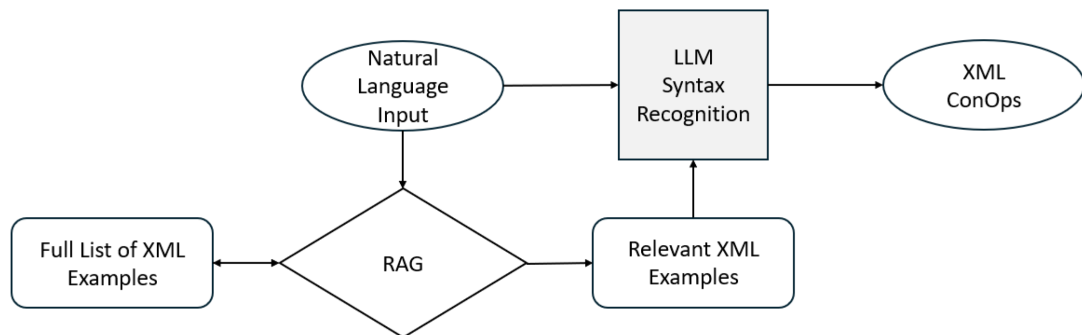

Figure 3.8: XML Examples pipeline enhanced by RAG

Combining both the structural and the example injections the first iteration of the XML pipeline was finally complete.
The full pipeline for the third design iteration, and the first one using the XML format, is displayed in Figure 3.9.

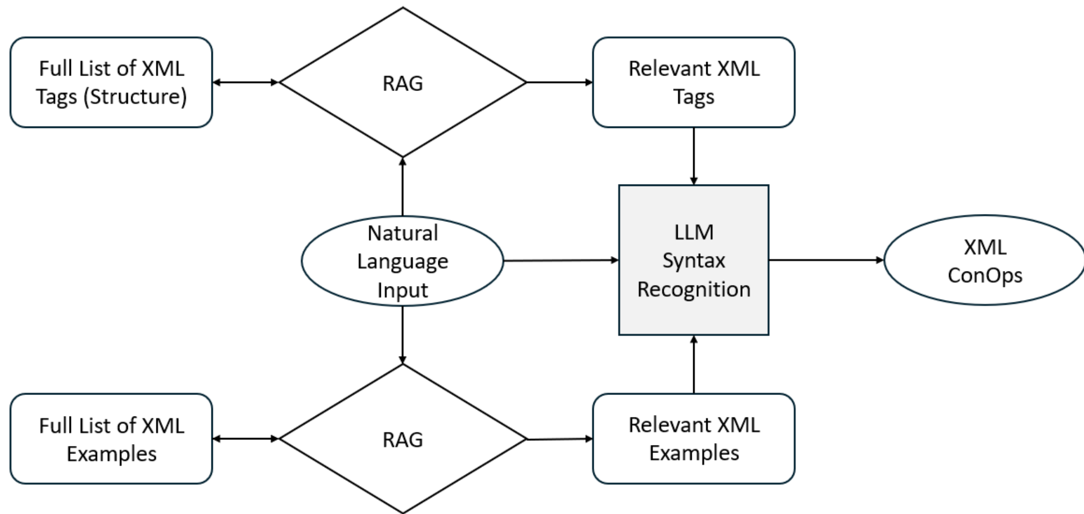

Figure 3.9: XML Schema pipeline enhanced by RAG

Although the architecture is virtually the same as the first JSON iteration, after a few rounds of testing it was evident that the output was not generated correctly, often generating XML with missing tags or with tags that the LLM hallucinated. This seemed to be a problem across all the models, from the small ones to the medium ones, and no matter how the parameters of the RAG were set, the output did not rectify.
The solution for this disastrous outcome will be shown in the following section, together with the final design iteration.

3.3 Syntax Recognition: Final Architecture

After several days of testing and debugging, the common problem with all the work XML ConOps was finally found: every ConOps, independently from the model used or the parameters of the RAG, was missing vital tags that were needed to have a complete XML, or some tags were in the wrong order.
This pointed out the culprit of the problem: the structural rule injection enhance by RAG.
Given the stochastic nature of the RAG and the ambiguous comparison between the user input and the description of every tag, most of the time tags that were strictly needed to generate a correct XML were often "forgotten" by the RAG and never injected in the LLM, leaving holes that the examples alone could not fill, and therefore creating incorrect or incomplete XML ConOps.
It was obvious that a more deterministic approach was necessary to handle the XML structure in the Tool, therefore the RAG was completely removed and replaced by a more compact definition of the XML structure injected straight into the LLM.
To better understand how this was done, the following is a sample example of an XML tree:

```
1  {
2  ELEMENT: OpMachine Tree
3  OUTPUT_TAG: <OpMachine>
4  DESCRIPTION: "This is the ROOT tree, it represents the main structure..."
5  STRUCTURE:
6      <OpMachine>
7          <Operation name="{operation_name}">
8              <Mode>
9                  <Subsystem_Mode>{SEE SUBSYSTEM MODE TREE}</Subsystem_Mode>
10                 <Pointing>{SEE POINTING TREE}</Pointing>
11                 <Thruster_Mode>{SEE THRUSTER MODE TREE}</Thruster_Mode>
12             </Mode>
13             <Event>
14                 <Trigger>{SEE TRIGGER TREE}</trigger>
15                 <Effect>{SEE EFFECT TREE}</Effect>
16             </Event>
17         </Operation>
18     </OpMachine>
19 RULES:
20 1. <OpMachine> must have 1:N <Operation> tags inside
21 2. Each <Operation> must have a unique operation_name
22 3. ...
23 }
24
25 {
26 ELEMENT: Subsystem_Mode Tree
27 OUTPUT_TAG: <Subsystem_Mode>
```

```
28 CONTEXT_PATH: OpMachine > Operation > Mode > Subsystem_Mode
29 STRUCTURE:
30     <Subsystem_Mode>
31         <Command subsystem="{subsystem_name}">
32             <command cmd="{on/off/idle}"/>
33         </Command>
34     </Subsystem_Mode>
35 OR (if no payload is mentioned):
36     <Subsystem_Mode/>
37 RULES:
38 1. ...
39 }
```

Using this new tree structure instead of the legacy single tag definition showed improved results in the correctness of the structure of the output.
Considering the current structure of the OpMachine XML in 360™, most of the sub-tags that define specific functions are either self-closing tags or simple small trees that have only few parameters and values. Given this, it was quite easy to compact the full structure of the XML inside of the Tool.
Unfortunately there were a few XML trees that were not strictly necessary to define the basic structure of the XML but were too big to compact together with the main structure of the XML in the schema file.
To solve this hurdle, it was decided to create a separate file, filled with these optional XML trees that are user dependent, and to inject them after the main structural rules.
In the system prompt of the LLM the difference between the two structural files was strictly defined, forcing the LLM to use the main structural file to generate the base XML structure, while looking at the optional structural file only when the user input matched the specific descriptions of the optional XML trees.

Given the few instances of the optional XML trees, this design was decided and implemented, but the idea of the possible architecture to implement after further development of the OpMachine in 360™ was still considered.
The current number of optional trees is four, so their inclusion in the prompt of the LLM would not consume too many tokens and cause the LLM to lose attention, but if that number grew, implementing a pipeline enhanced by RAG might be beneficial if not necessary.
The implementation of this RAG would follow the same logic as the previous iteration of the structural rules pipeline, comparing the user input with specialized descriptions of the different optional trees, allowing for fast injection of the required trees alongside the necessary structural trees, without consuming tokens and bloating the user prompt.

The XML structure pipeline minus the use of RAG is displayed in Figure 3.10.

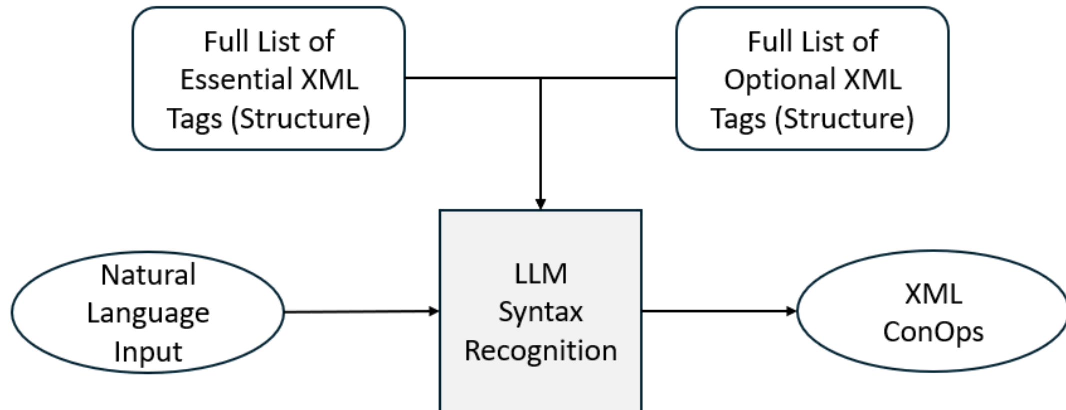

Figure 3.10: XML Schema pipeline

After overhauling the XML structure injection pipeline, a temporary overhaul of the example injection pipeline was also performed, to further test if the complete removal of the RAG would enhance the quality of the final output or decrease the computational time.
This meant injecting a set amount of examples directly in the user prompt of the XML, before the user input.
To compensate for the token weight of many XML output examples, it was decided to include only a few representative examples, enhancing how the LLM understood how to correctly follow the structure from the XML schema.
The XML examples pipeline minus the use of RAG is displayed in Figure 3.11.

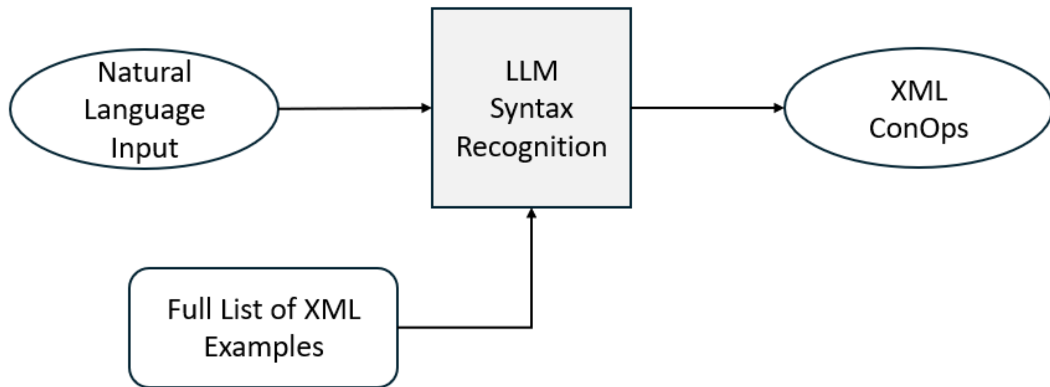

Figure 3.11: XML Examples pipeline

Combining the new overhauled pipelines, further testing was performed to confirm the improvement of the XML ConOps generation.
The full pipeline for the fourth design iteration, and the second one using the XML format, is displayed in Figure 3.12.

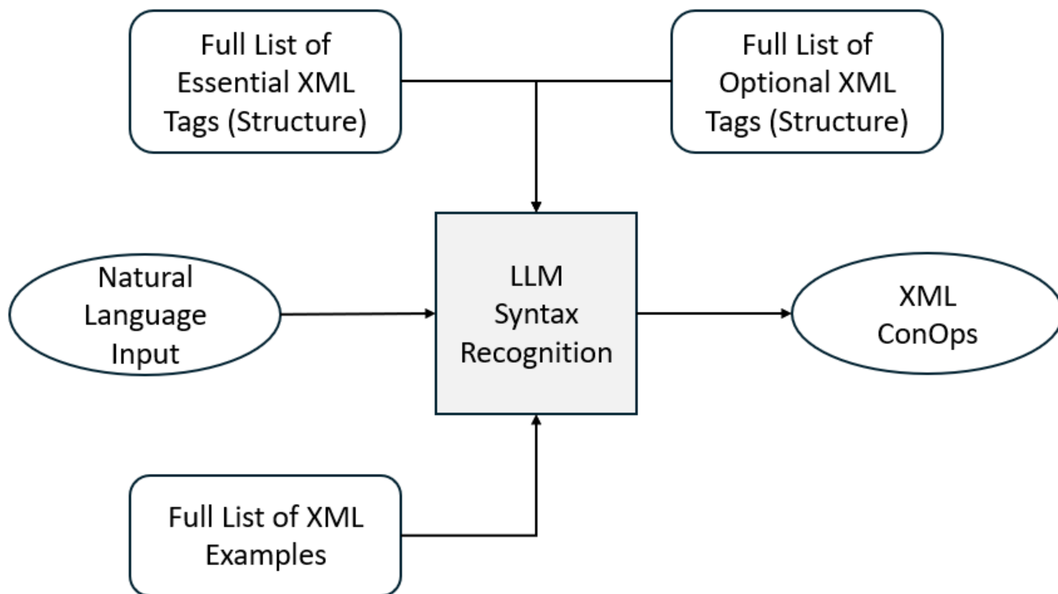

Figure 3.12: Full XML Schema pipeline

After several rounds of testing the new XML structure injection pipeline proved

successful in rectifying the mistakes present in previous iterations, allowing for the XML output to be correctly structured, and also following the correct order to be compatible with 360™.
Unfortunately, it was noted that when the test input varied and became more complex, the output started showing fatal mistakes. This was clearly from how the examples were implemented, showing how the RAG was not completely inapplicable for the Tool, and in fact was required to enhance the understanding of the user input for the LLM.
Whilst the structural rules were the necessary building blocks that the LLM needed to create the XML, the examples were the rule-book that explained not only how to properly assemble those blocks, but also how to correctly understand user intent, by matching a similar exemplary query with the actual user input.

3.4 Syntax Recognition: Recap

Having identified the best pipelines for both the XML structure and the examples, it is important to do a final recap, to see how the final architecture that was developed looks like.

Like stated in the previous section, the chosen schema pipeline is the XML structural rules injection, with the main structural rules that explain to the LLM the base required structure that the output must follow, together with the optional structural rules, that are loaded in the LLM but only used when the user calls upon them.
The final XML structure pipeline is displayed in Figure 3.13.

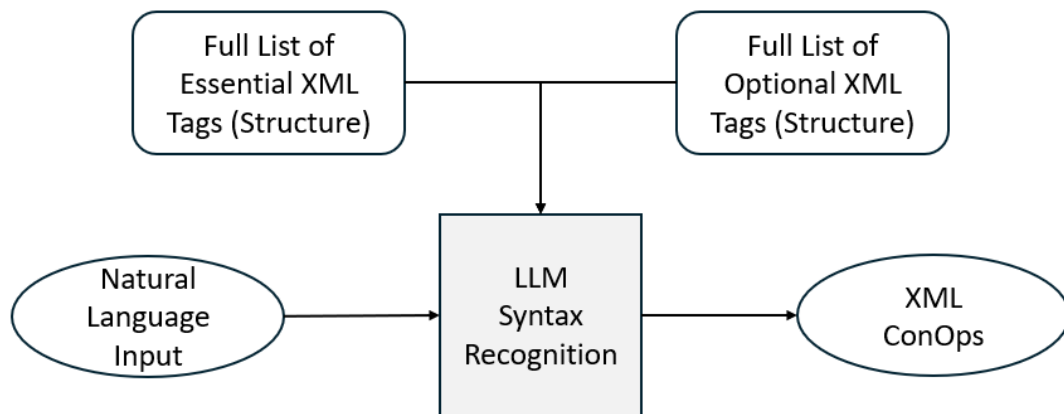

Figure 3.13: XML Schema pipeline

For the example pipeline, the best choice is the RAG enhanced injection, which compares the user input with the example inputs, to better choose examples that are the most similar both semantically and syntactically, to allow the LLM to better understand how to handle correct XML tree structure and implied user intent and edge cases.
The final XML examples pipeline is displayed in Figure 3.14.

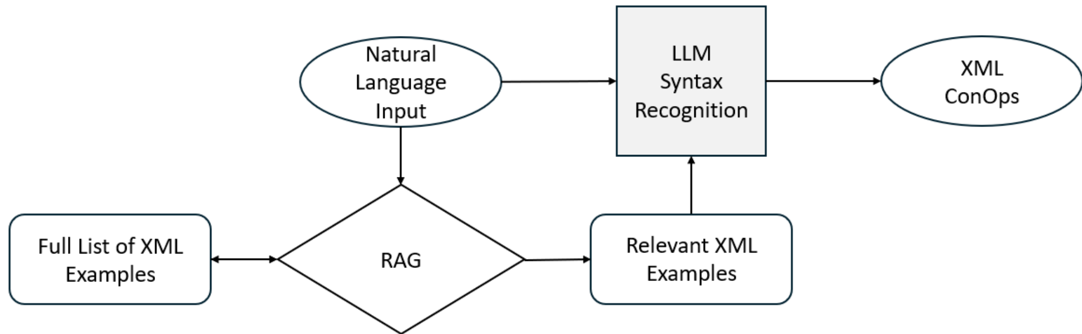

Figure 3.14: XML Examples pipeline enhanced by RAG

Combining both of these pipelines the final, full architecture for the syntax recognition is finally confirmed, and therefore fully developed and implemented. The full pipeline for the final design iteration is displayed in Figure 3.15.

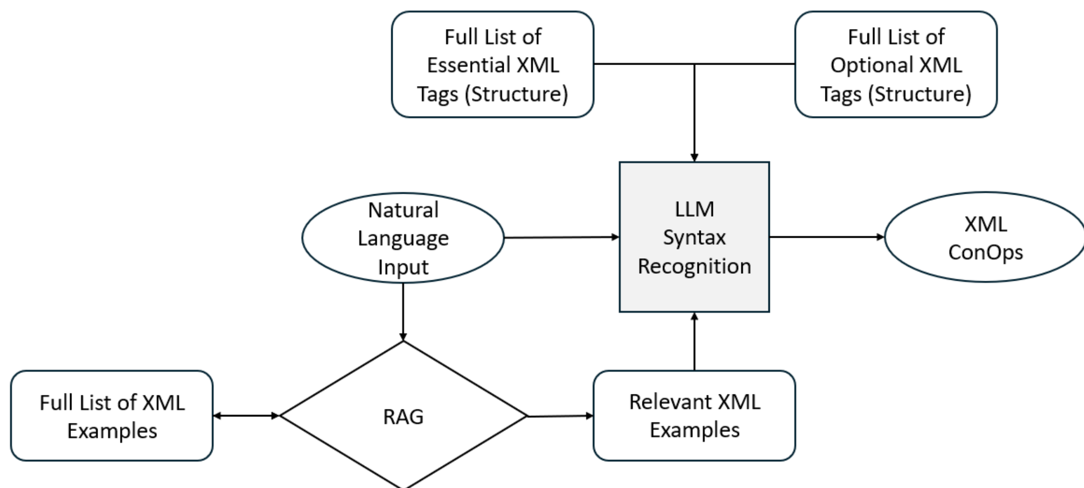

Figure 3.15: Full Final XML Schema pipeline

Having completed the first and most complex part of the Tool, it is important to take a look at the development of its second segment, the code generation, to understand how its design changed over time and to see the final implemented solution.

3.5 Code Generator

During the design development of the syntax recognition, several ideas for the code generator were laid out, but the main choices were either a stochastic approach using a LLM or a deterministic approach with a hard-coded function.
In the late stage of development of the JSON ConOps architecture for the syntax recognition, it was decided to explore the stochastic approach.
This choice was driven by the the need to handle a fairly simple and machine readable object such as the JSON file that does not strictly follow an ordered structure. It seemed appropriate and given the already developed functions created to communicate with the LLM the choice was obvious.
The base design for this LLM powered function was a simple delivery of the ConOps schema given by the syntax recognition together with the base explanation of the different correctly worded functions of the OpMachine in 360™ and some sample examples showing the correct structure of the code.
The pipeline for the LLM powered Code Generator is displayed in Figure 3.16.

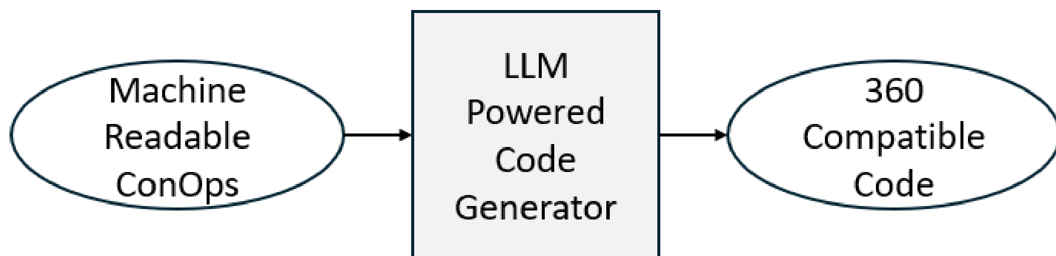

Figure 3.16: Full Final XML Schema pipeline

Although appealing at first, this architecture was quickly abandoned because of its stochastic nature. After changing the ConOps structure from JSON to XML, its increased rigidity lead to the need of a more predictable and reliable function. Thus the hard-coded function was designed and developed.
Using the ordered and strict nature of the XML it was easy to develop a function that would scan the XML for specific tags and create a string with the correct corresponding function of the OpMachine in 360™.
The pipeline for the hard-coded Code Generator is displayed in Figure 3.17.

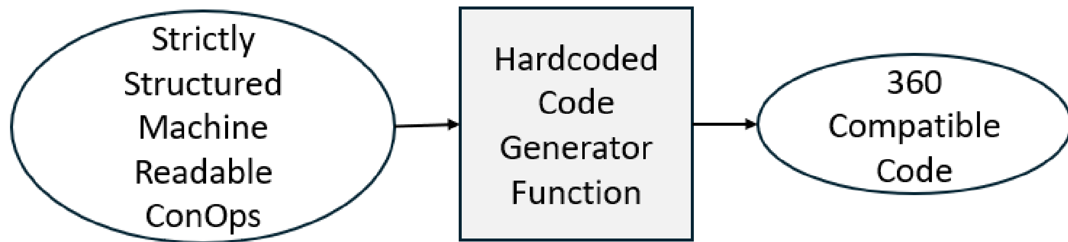

Figure 3.17: Hard-Coded Code Generator Function pipeline

To better understand how the function works, a simplified depiction of the inner workings of the function is shown as follows:

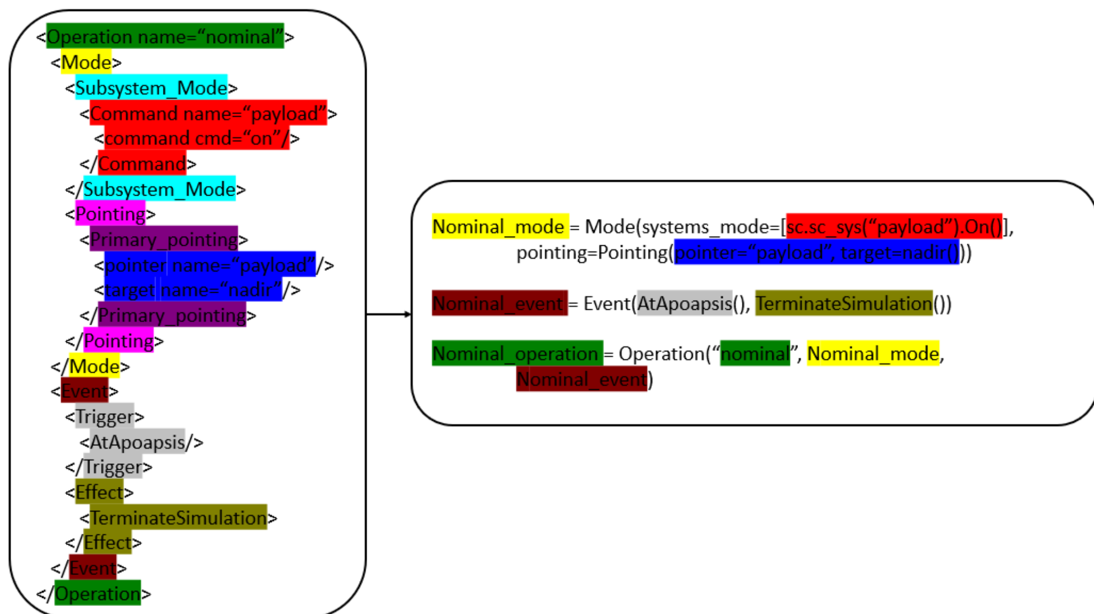

Figure 3.18: XML to code translation

it is clear to see how the tags for the mode and the event are used to create the call for the respective mode and event functions, while the sub-tags populating them are used to correctly fill the parameters of the main functions, calling upon specific functions to do so.
This scan approach made the function completely reliable, and it even added extra validation for the XML ConOps, raising an error if the specific required tag was either not present or misspelled.

Having finally decided the complete architecture of the Tool, it is important to understand how its output behaves, to check its correctness and possible improvements.

Chapter 4

Testing and Validation

Having defined the architecture for the Tool, it is important to check if it actually functions as expected.
To do so several rounds of testing were performed to validate the output of the Tool. Specifically, given its hybrid nature, it is important to check both the final Python code output and the intermediate XML object.
Having said so, this chapter will be divided in two parts: the validation of the syntax recognition, in which the stochastic nature of the LLM is studied, controlled and directed as best as possible through the manipulation of the LLM model and the RAG parameters, and the validation of the code generation function, which, being fully deterministic, proved to be easier to analyze.

4.1 Validation Objectives

To correctly perform the tests and get useful results, it is important to set some specific validation objectives, which are specific characteristics of the output that must be met by the actual output of the Tool.
These requirements, divided for validation objectives for the XML object and the Python code, are listed as follows.
For the syntax recognition:

- XML structure: this parameter checks for the correct structure of the XML, to verify if the LLM can correctly follow the structure defined in the XML schema documents and generate the XML with no missing or hallucinated tags.

- XML semantics: this parameter checks if the XML tag arguments are filled and formatted properly following the user input. This means checking if the

LLM can correctly understand the user input and derive all needed variables, that are then correctly formatted in the XML.

- Consistency: this parameter tests how consistent the LLM is in replicating the correct XML output without changing the user input.
- Computational time: this parameter checks the time from the initial query to the LLM to the final XML output. This parameter is most important for user comfort and to check if the model can efficiently run on the provided hardware.

While for the code generation:

- Script completeness: this parameter refers to the full generation of the Python code from the provided XML, without missing any vital functions.
- Function correctness: this parameter refers to the correct allocation of the XML arguments into the Python functions.
- Reliability: this parameter checks if the function can correctly generate functional python code regardless of the provided XML input.

To better understand how the results of the tests are analyzed, the following is the legend for the performance comparisons:

- Bad: whenever a validation objective is satisfied for less than 50% of all the tests performed. For example, if the XML structure generation for a model is indicated as Bad, it means that every time a test is run, the structure of the output will have a 50% or higher likelihood of being incorrect.
- Average: whenever a validation objective is satisfied between 50% and 70% of all the tests performed. This indicates an improvement from the Bad value, but still implies that at least a quarter of the output will provide an incorrect XML object.
- Good: whenever a validation objective is satisfied between 70% and 90% of all the tests performed. It is a further improvement compared to the Average value, and implies lower erroneous generation from the model.
- Excellent: whenever a validation objective is satisfied more than 90% of all the tests performed. This is considered the best possible outcome for a validation objective, but given the stochastic nature of LLMs, it is the rarest to see.

Having chosen the validation objectives it is imperative exploring how the syntax recognition and the code generation perform.

4.2 Syntax Recognition

Since the architecture was fully developed and implemented, the only parameters that can be manipulated are the LLM model and the RAG parameters.
Two independent test sets were then defined, each one targeting a specific parameter of the system while keeping all other variables fixed.
The LLM model is the functional brain of the syntax recognition segment of the Tool, so testing different models is important in finding which one fits best in the operational scenario of the Tool, and also works well with the available hardware of only CPUs that was given for this project.
The RAG parameter is simply the amount of examples provided to the LLM. Increasing this variable will help the LLM understand the correct use of more XML tags at the cost of bloating the prompt and risking losing the attention of the LLM towards the user input and possibly increasing the computational time.

Two sets of tests were performed to check the viability of the Tool, the first test simply compared different LLM models with a set amount of examples injected in the user prompt, while the second test compared the output generated with the winning model from the first test while varying the RAG parameter and finding its optimal value, which hits the sweet-spot between XML correctness and computational time.
To better understand the results of the tests an example is provided. This example was used for the testing in both test cases and mirrors a realistic spacecraft operation.
Test input:

The spacecraft's main operation consists of using the PowerDevice named 'PAYLOAD' pointed towards the Earth. When the battery level reaches 20% the spacecraft must put the payload in idle and point its solar panels called 'SOLARPANEL' towards the Sun until the battery level reaches at least 40%, after which the spacecraft shall return to its nominal operation.

Expected XML output:

```
1  <OpMachineWith>
2      <Operation name="NOMINAL">
3          <Mode>
4              <subsystem_commands>
5                  <command functional_id="PAYLOAD">
6                      <PowerCommand cmd="on"/>
7                  </command>
8              </subsystem_commands>
9              <pointing>
10                 <main>
```

```
11                      <pointer>
12                          <fid>PAYLOAD</fid>
13                      </pointer>
14                      <target>
15                          <Nadir/>
16                      </target>
17                  </main>
18                  <secondary>
19                      <MaxSunlight/>
20                  </secondary>
21              </pointing>
22              <thruster_commands>
23                  <command idx="1">
24                      <ThrusterOff/>
25                  </command>
26              </thruster_commands>
27          </Mode>
28          <Event>
29              <effect>
30                  <ToOp target_op="2"/>
31              </effect>
32              <trigger>
33                  <OnSingleBatterySOC crossing="FallBelow">
34                      <threshold unit="-">0.2</threshold>
35                  </OnSingleBatterySOC>
36              </trigger>
37          </Event>
38      </Operation>
39      <Operation name="POWERSAVING">
40          <Mode>
41              <subsystem_commands>
42                  <command functional_id="PAYLOAD">
43                      <PowerCommand cmd="off"/>
44                  </command>
45              </subsystem_commands>
46              <pointing>
47                  <main>
48                      <pointer>
49                          <fid>SOLARPANEL</fid>
50                      </pointer>
51                      <target>
52                          <SunFacing/>
53                      </target>
54                  </main>
55                  <secondary>
56                      <MaxSunlight/>
57                  </secondary>
58              </pointing>
59              <thruster_commands>
```

```
60                  <command idx="1">
61                      <ThrusterOff/>
62                  </command>
63              </thruster_commands>
64          </Mode>
65          <Event>
66              <effect>
67                  <ToOp target_op="1"/>
68              </effect>
69              <trigger>
70                  <OnSingleBatterySOC crossing="RiseAbove">
71                      <threshold unit="-">0.4</threshold>
72                  </OnSingleBatterySOC>
73              </trigger>
74          </Event>
75      </Operation>
76  </OpMachine>
```

This example only shows the most relevant sections of the real XML used for clarity.
This example portrays the complexity of the XML structure that the models must follow, including conditional transitions, nested tags, and the use of the major OpMachine functions. It was used to evaluate both structural correctness and semantic interpretation throughout all of the models.
Unfortunately, given the novelty of this design, the unavailability of already existing datasets and the long wait time between every call of the LLM, the dataset used for the testing was limited, and the results could not be evaluated statistically.
Therefore the following is a qualitative evaluation of the observed test results.

4.2.1 LLM Model Trade-Off

The first test was a comparison between the used LLM models. In this test their ability to understand the user input, generate a structurally and semantically correct XML and the time required to receive a response was compared.
The models tested are the following:

- deepseek-r1:1.5b: this was the first model used, when the available hardware was limited to a single low-spec pc.

- deepseek-r1:7b: after upgrading the available hardware to a server running on CPUs it was decided to test out bigger models. Given the first attempts with this model's smaller counterpart, this medium sized model was chosen.

- gemma3:4b: being the built-in model of Ollama, it was decided to test this model.
- qwen3:4b: chosen for its flexibility in multiple environments.
- qwen2.5-coder:7b: chosen for its focus on code generation, used to check the viability of using such models for XML generation.
- ministral-3:8b: chosen for its adaptability to low-end hardware.
- llama3.1:8b: chosen for its capability of running on CPU only hardware.
- phi4:14b: chosen for its greater size, to test the performance of bigger models.

All of the models were tested with several prompts, repeating tests with the same prompt multiple times.
The performance of each model is displayed in Table 4.1.

model	XML Structure	XML Semantics	Consistency	Computational Time
deepseek-r1:1.5b	Bad	Bad	Bad	Very Low (∼3mins)
deepseek-r1:7b	Good	Bad	Good	High (∼15mins)
gemma3:4b	Bad	Good	Bad	Medium (∼10mins)
qwen3:4b	Bad	Bad	Bad	Medium (∼7mins)
qwen2.5-coder:7b	Good	Average	Average	High (∼20mins)
ministral-3:8b	Good	Good	Good	Very High (∼30mins)
llama3.1:8b	Good	Excellent	Good	Low(∼5mins)
phi4:14b	Good	Good	Good	Very High(∼45+mins)

Table 4.1: Model performance comparison

The results show a few predictable outcomes together with some unexpected ones.
For smaller models, ranging from 1.5b to 4b, the structure of the XML showed major problems, from missing tags, to misspelled tags, to wrongly structured XML sub-trees, these models vastly underperformed. This rendered them unusable, and therefore they were quickly discarded.
What is notable is that, even though the structure of the XML was often wrong, the model was observed to understand the user intent quite accurately, being able to generate a vaguely coherent XML for the specific user input.
Another problem with these models was the lack of consistency, given the wrongly formatted structure of the XML, different runs on the same input often gave wildly different XML, with little to no uniformity.

Going to the medium sized models, 7b to 8b, a notable increase is observed in the correct generation of the structure of the XML. What is interesting is the fluctuation of the quality of the results depending on the model. Some models showed more comprehension of the user input at the cost of computational time, while others seemed quite unreliable, generating wildly different outputs from the same input. An increase in computational time compared to the smaller models is also observed, even though there are still some fluctuations, where some models tend to be relatively quick while others reach up to half an hour.
Lastly one large model, the phi4:14b was tested, and was not further considered, given its unacceptably high wait time for a response. It was also noted how this model often generated extra operations where they were not needed, probably a limitation for the restrictive hardware.
The following are some examples of the most common errors generated by the different models:

- Missing XML tags: it was observed that the pointing functionality of the OpMachine proved to be quite hard for many models to handle, since many either forgot to include the <secondary> pointing tag or the <main> tag, but managed to correctly generate the <pointer> and <target> tags.

- Extra operations: some models hallucinated extra intermediary operations which were never mentioned in the user input. Using the previously displayed example, certain models created an extra operation in which the spacecraft was fully idle.

- Incorrect XML tag arguments: certain models struggled with correctly placing the right parameters inside of the XML from the user input. This was also seen especially with the pointing functionality of the OpMachine, where the models often incorrectly chose the pointer and the target. Also there were certain problems with the correct generation of certain tags, such as the <SunFacing/> tag, which was often misspelled, for example simply as <Sun/>.

At the end of the testing phase, the most crucial qualities that were chosen for the winning model were the ability to correctly generate the XML structure and the computational time.
These two main attributes were chosen because of the dependency of the correct XML semantics and the consistency can be partially improved through the tuning of the RAG parameters.

After performing several tests, the winning model chosen to be used for the OpMachine Tool is the llama3.1:8b, which managed to output consistent correctly structured XML in a timely manner.

Given again the restrictions of the hardware, the ability of llama3.1:8b to run efficiently even on only CPU made it the best choice.

4.2.2 RAG Parameter Trade-Off

Having chosen a model, another set of tests was performed to assign the best value for the RAG parameter.
Several tests were performed with inputs that vary in complexity.
The performance of different ranges of the RAG parameter is displayed in Table 4.2.

Range	XML Structure	XML Semantics	Consistency	Computational Time
1 to 2	Average	Bad	Bad	Low (∼4mins)
3 to 5	Good	Good	Good	Low (∼5mins)
6 to 10	Good	Good	Average	Medium (∼6mins)
10+	Bad	Average	Average	Medium (∼8+mins)

Table 4.2: RAG parameter performance comparison

The results show how the amount of examples injected in the LLM can drastically change the ability of the LLM to correctly understand the user input and generate a correctly structured XML.
For very few examples (1 or 2), the LLM is unable to fully reference the user input with the available tags, being unable to correctly choose certain sub-tags and often compensating with hallucinated tags. This rendered the LLM highly unreliable, given its inability to correctly assess user intent, especially for more complex inputs.
Using few examples (3 to 5), the LLM is capable of correctly understanding user intent and generating structurally sound XML. It still faced struggles with more complex inputs that used several different functions. This is because of the composition of the examples: every example is purposely created to have one or two available functions used in the OpMachine, creating a complete set that includes all of the available functions of the OpMachine. If a user input is complex enough, it could require more functions than five examples can provide, lowering the chances of correct XML generation.
Using more examples (6 to 10), the first signs of LLM fatigue show themselves: if the user input is complex the generation is actually quite good, mirroring the performance of the previous set with low to average complexity inputs, unfortunately, for simple inputs the RAG still injects multiple barely relevant examples, increasing the prompt complexity and consuming more tokens than needed. This cluttering of the prompt lowers the available attention that the LLM can give to the user input, often generating XML that are a collage of the multiple example XML, with barely any connection with the actual user input.
While the average computational time for all of these setups was quite similar, only slightly increasing with the more examples injected in the user prompt, adding even

more examples seemed to completely mess the user input understanding for the LLM, leading to an increase in the computational time, where the model was trying to piece together a valid XML from all of the examples. This configuration often ignored the user input fully, regardless of complexity, but also the structural rules that are injected just before the examples, often leading to structurally incorrect XML that somewhat resembled the injected examples.

From this analysis, it was decided that the most appropriate value of the RAG parameter is four. This middle point value manages to compensate between the need for enough example use OpMachine functions and the low cluttering of the user prompt.

It is important to remember that this set value is only effective for the selected model, the llama3.1:8b. Whenever a new model is selected, a new RAG parameter trade-off analysis must be performed, to correctly boost the capabilities of the model.

4.3 Code Generator

The code generation function is a hard-coded deterministic Python function that creates 360™-compatible Python code from any valid XML of the OpMachine.
Being hard-coded, this function is fully deterministic, so its validation is straightforwards. This function receives the XML object from the LLM, it purifies it from any possible flavor text and comments that the LLM might generate and then it scans it, looking for specific tags, both structural tags, like <Operation>, <Mode> and <Event>, and specific functional tags.
To better understand this process it is important to take a look at the correctly generated Python code from the example in the previous section.
Correctly generated Python code:

```
1 NOMINAL_mode = Mode(Pointing(Main("PAYLOAD", AlongVelocity()), [sc.sc_sys("
      PAYLOAD").On()])
2 NOMINAL_event = Event(OnBatteryLevel("<", 0.2), ToOp("POWERSAVING"))
3 NOMINAL_op = Operation(NOMINAL_mode,[NOMINAL_event])
4
5 POWERSAVING_mode = Mode(Pointing(Main("SOLARPANEL", Nadir()), [sc.sc_sys("
      PAYLOAD").Off()])
6 POWERSAVING_event = Event(OnBatteryLevel(">", 0.4), ToOp("NOMINAL"))
7 POWERSAVING_op = Operation(POWERSAVING_mode,[POWERSAVING_event])
8
9 om = OpMachine(NOMINAL_op, POWERSAVING_op)
```

Having stated this, it is clear to understand that this function has been validated by construction.
While developing the tool, every new addition of a 360™ OpMachine function was then promptly tested with the correct XML tag, to ensure the correct formatting of the final code.
This meant that, when developing for example the Mode() function, the function would specifically look for the Mode tag, then it would through its sub-tags, in search for the relevant pointing, subsystem mode and thruster mode tags. After finding the needed tag, it would generate the correct sub-function, such as Pointing() in the example. After that it would look for the main tag, under the pointing tag, and call the Main() function. Lastly it would look for the specific pointer and target tags to correctly populate the Main() function.
This process was then tested with single XML tags and subsequently with full XML trees. This was easy to do since the function is divided in three main parts, a mode generator, an event generator and the final operation generator: the mode generator looks strictly through the Mode XML tree and correctly generate the arguments for the Mode() function, the event generator would do the same for the Event XML tree, while the operation generator would call upon the previous functions and correctly create the variable names and the final printed structure of the code.
After completing the function it was tested with several XML, and the generated code was then compared with the handwritten one used in the generation of the XML objects in 360™.
After confirming the correct behavior of the function, it became evident that this deterministic function was fully predictable, and therefore validated.

Chapter 5

Discussion of Results

Having performed various tests for both the syntax recognition and the code generation, and having decided on the best LLM model to use and the best value for the RAG parameter, it is important to take a moment to consider what the outcome of the testing means.

5.1 Project Outcome

Looking at the syntax recognition, it is clearly observable that the model can generate a structured XML with controlled accuracy.
For the context in which the Tool was developed and with the used test structure, the Tool works correctly, but its performance is heavily tied on the model used, the architecture of the documentation delivery, the specific design of the documentation and the final structure of the prompt and finally on the RAG configuration.
This renders the Tool conditionally reliable, not universally reliable.
It is imperative to consider that this very specific architecture is well applicable to the OpMachine of 360™, but if used in any other section of 360™, it might not give favorable outputs, and thus should be modified to compensate the change.
Observing the full Tool, it is clear that this does not apply to the code generation function, which, even though tailored only for the OpMachine, can be easily updated following the same design principles to be compatible with any other part of 360™.
it is also interesting to observe how the syntax recognition is mainly responsible for the correctness of the final output, since it is the acting "brain" of the Tool. This is because the syntax recognition, performing the translation of the natural language input into a structured model of the operational logic of the user input, and effectively controls whether the final output is coherent with the user input or not.

In this section a full exploration of the results obtained in the previous section with a more in-depth look at the meaning of certain design choices.
After developing the Tool, it is clear to see that the bottleneck for its correct functioning is the XML structural correctness. If the model generates an incorrect XML, the Tool raises an error and has to be reinitialized. This means that the LLM must understand the XML structure properly, and this behavior is observed in most of the medium sized models. This is only possible thanks to the high level of restriction that the XML schema documentation imposes on the LLM, forcing it to travel in a set of rails predefined by the developer. Together with the XML structural schema, the examples also prove to be vital to show the LLM how to correctly generate the full structure on the XML and not just its nested trees.
Unfortunately, a higher restriction on the LLM means that it is less creative, and this can lead to a lower flexibility in understanding user input, especially for implied context and edge cases.
This is another reason for why smaller LLM models failed: being overly restricted lead them to not be able to correctly piece together the different XML trees defined in the documentation into the proper XML structure. Also the focus on smaller sub-tag certainly requires more computational power, that these models simply cannot provide.

Another important aspect of the correct functionality of the Tool is the correct understanding of the user input by the LLM, and therefore the correct XML semantics.
This characteristic does not strictly block the Tool from functioning, since a correctly structured XML with variables that do not correctly reflect the user's needs will still be parsed and generate functional but incorrect Python code.
If the LLM is highly creative it will be able to understand even the more difficult aspects of natural language, but this characteristic clashes with the requirement for correct XML structure generation, since it asks the model to be less restricted.
Looking at this, it is clear that the best scenario focuses more on model restriction to be able to correctly function, but does not impede the LLM from thinking creatively and therefore understanding user input.
The choice of a "best model" is therefore deeply rooted in this context. It is a compromise, and the best model is chosen not on pure performance for structural correctness, user understanding or computational time, but on the ability to operate in this middle-point. Another big constraint was the computational time: a wait time up to one hour for the Python code to be generated could greatly lower the comfort of the user in using the Tool.
Taking all of these in consideration, the choice of llama3.1:8b is clearly the best,

since this model manages to compensate structural correctness, user intent understanding and computational time the best compared to other models tested.
This choice was also heavily dictated by the currently available hardware, which rendered possible better models unusable.
Lastly it is very important to remember that, being stochastic, the reliability of the LLM will never be perfect. There is always the risk of incorrect generation, even from an input that previously received a correct output.
This is where the consistency of the model is most important. Thankfully, most models seemed to be able to perform discreetly in this aspect, generating consistent outputs from the same input.
The element that supports the LLM in correctly generating XML structure and understanding the user output is the RAG enhanced example delivery, but this system is clearly not universally applicable.
Like shown in the previous chapter, too few or too many examples lead to incorrect output, so the right choice of the value of examples injected greatly influences the model's contextual awareness and user input attention.
It is clearly observable how the attention of a model is limited and it can vary greatly depending on the model itself. Thus, it must be allocated carefully to ensure that the user input is not lost in the prompt.
Also, the chosen value for the RAG parameter does not apply to every model. If a different model is considered, this study must be performed again, to choose correctly the amount of examples needed for the LLM to function properly.
In other tests performed to observe this behavior, it was noted how bigger models could handle, and actually benefited, from a larger number of examples, while smaller models usually required only two examples to function properly.

Another defining attribute that was not discussed previously is the structure of the user input. While the Tool has been extensively tested, if the user input does not explicitly state all of the wanted operations with the related values, the Tool might not be able to correctly generate the user's envisioned ConOps.
it is also imperative that the user input adheres to the rules of the OpMachine of 360™. If the user requires a function that is currently unavailable in 360™, even if the Tool generates a XML with the correct structure, the final output would no longer be compatible with 360™, and therefore unusable.
Given its current state, it has also been noted that the Tool tends to struggle with highly complex inputs. This fault can be explained by the relatively small size of the LLM model used.

In conclusion, it is possible to define the Tool as conditionally reliable. This means that, while not being fully reliable, as long as the syntax recognition segment manages to interpret the user input and generate a correct XML, the Tool will

work as envisioned. It is therefore important remembering that this Tool will not always produce correct Python code, and a final user check must be performed to guarantee that the LLM correctly understood user intent.
Possible strategies to improve this are explored in the next section.

Given all of this, it is possible to state that, despite the limitations, the Tool successfully demonstrates the feasibility of using a LLM-powered application to translate natural language ConOps descriptions in ready to use 360™-compatible Python code.

5.2 Future Improvements

In this section possible improvements that could increase the output correctness of the Tool and its overall performance will be explored.

The main bottleneck encountered during the development of the Tool was definitely the available hardware. In the initial stages of development the only available hardware was a low spec computer, which could barely run 4b models. Halfway through development a remote server was rendered available for the development of the Tool, rendering it possible to run bigger models. Although a major improvement, the server only used CPU, while most commercial LLMs need a GPU to properly run, therefore the use of even bigger models was not permissible, since the computational time needed to run medium models often peaked at one hour.
A further upgrade to a server powered by GPU would further increase the capabilities of the Tool. Running a LLM on GPU often cuts the wait time for a response by a factor that ranges from 10 to 100. This would effectively nullify latency and also allow for bigger models to be used.
Having a bigger model that responds instantaneously would also allow for easier testing and therefore more accurate validation.
The main setback for testing was the wait time, stretching simple tests that would usually require hours to days. Having a faster response would give more time to test bigger test suites, and therefore allow for a less qualitative and more empirical analysis of the results. This would then lead to a more detailed report on the behavior of the LLM, and the ability to use bigger models would allow for an increase in the reliability of the Tool, given the enhanced understanding of its inner-workings.
Another beneficial addition would be an automatic validation of the output, to ensure that any output given to the user would be 360™-compatible. This simple addition would basically reinitialize the LLM whenever it detected incorrect Python code, or when the code generation function would raise an error for an invalid XML,

possibly even feeding the wrong output to the LLM to ensure that the mistake is not repeated.
With the current setup, this functionality would be too time consuming and was therefore not fully explored.

Other than using commercially available pretrained LLMs, it would be interesting to train a domain specific LLM on the specifics of this project, fine tuning it to increase reliability and effectiveness. This would definitely allow the Tool to generate structurally correct XML objects with almost constant reliability, since the LLM model would have been tailor-made to generate said structures. Having also an automatic validation layer after the XML is generated would further allow the Tool to competently generate the structure whilst making little to no mistakes. Having an extra validation layer, and with computational time barely affecting user experience, the Tool would be able to autonomously check the structure of the XML and also its semantic correctness, and whenever needed it would be able to correct itself, possibly also using another pipeline that would explain to the LLM how to correctly catch and handle mistakenly generated XML structures or semantically incorrect XML, further increasing reliability to almost 100%.

Another big improvement would be a more refined user interface: adding an environment that guides the user's input to be coherent with the available functionalities of 360™ would reduce the training required by the user to understand the inner-workings of the OpMachine, and further improve user experience.

Chapter 6

Conclusion

6.1 Overview

This thesis addressed the viability of automating the generation of Concepts of Operations proposing a Large Language Model-based approach within the 360™ environment. The work focused on bridging the gap between high-level, natural language operational requirements and structured, machine readable ConOps configurations by developing a modular Python-based proof-of-concept Tool.
The main achievements of this project were the design and development of the base architecture of the Tool, that separates syntax recognition of the user input from the Python code generation. The syntax recognition segment uses an LLM, supported by structural XML schema and Retrieval-Augmented Generation example pipeline, that translates the user input into a correctly structured and semantically correct XML ConOps. The code generator is instead a deterministic functions that correctly outputs 360™-compatible Python code from the XML ConOps. Several tests were performed to choose both the best performing models and the best suited RAG configuration. The choice for the best configurations depended on the trade-off of several parameters, such as the structural correctness of the XML ConOps, the semantic accuracy of the ConOps and therefore the ability of the LLM to understand user intent, the consistency of the generated XML ConOps and finally the computational time needed to receive an answer after initializing the LLM.
The results demonstrate that the proposed architecture is capable of generating structurally correct ConOps in acceptable computational times on limited CPU-only hardware, establishing a functional baseline for LLM-assisted configuration generation. The modular design improves maintainability, reliability and scalability, while the deterministic code generator ensures predictable and validated Python code.
Overall, the thesis confirms the viability of integrating LLM-based assistance in

the ConOps generation workflow of the 360™ environment and provides a solid foundation for future development and integration into operational aerospace environments.

6.2 Future Work

The finalized project proved to be rather satisfactory for the environment in which it was developed, but it is still important to remember that this Tool would not be as effective in a different context. The current setup of the OpMachine in 360™ offers a variety of functionalities that the Tool correctly generates, but currently, a lot of work is being done to further develop 360™ as a whole and give more functionalities to the OpMachine.
This clearly indicates that the current structure of the Tool could become obsolete in the next update, and it is important to understand the work that must be performed to bring the Tool to full speed whenever necessary.
Whenever new functionalities are added to the OpMachine, it is important to update the documentation for the Tool. All of the new XML tags must be included in the XML structure schema, in which simple tags can be simply added to the main structural documentation, while more complex XML trees must be added in the optional structural documentation. It is also necessary to update the example documentation, by adding new examples that follow the structure of those already built, and that functionally show the correct interpretation of the user intent for the new functionality and the correct complete structure of the XML.
Whenever the amount of optional complex XML trees becomes high enough, it would then be reasonable enhancing the optional XML structure pipeline with RAG. This will allow for a less cluttered prompt while still allowing the model to correctly understand the user requirements.
It is also advisable to perform testing to ensure the correctness of the XML ConOps generation after every new update. This will allow the Tool to maintain its reliability and effectiveness even after multiple updates.
It is also very important to update the code generator function with the new functionalities of the OpMachine. To do so, simply following the already built structure of the XML handling and code generation should be followed. It is also important testing the correct code generation after updating the function.
Similar updates must also be performed in the migration of the Tool from the OpMachine environment of 360™ to any other environment. Before migrating it is advisable to perform a feasibility analysis to better understand if the current architecture of the Tool could perform adequately with the new context.

Acknowledgements

I would like to give my deepest gratitude to professor Elisa Capello for giving me the amazing opportunity of working on this project. I am grateful for being able to explore the world of artificial intelligence while applying the knowledge and skills learned during my university career.

I would like to give a warm and sincere thanks to my supervisors, Gianni Pecora, Daniel Talavera Jiménez and Samuel Reskala Eguiarte, for their constant support throughout my work. Thank you for all the help you have given me and the time you spent with me.

To mom, dad and Gabriele, thank you for all the support you've given me throughout the years and for following me on this journey. With my university career coming to an end I sincerely want to thank you for everything you have done for me. Thank you for believing in me and for allowing me to be here today.

Lastly, I would like to thank you, Dalma, for being there for me when i needed it the most, both in happy and sad times, and for supporting me throughout all of my career and work. I would not be the same person I am today if not for your presence. Thank you.

Ringraziamenti

Vorrei offrire la mia più sincera gratitudine alla professoressa Elisa Capello per avermi dato questa fantastica opportunità di partecipare a questo progetto. Sono grato di aver potuto esplorare il mondo della intelligenza artificiale sfruttando le competenze apprese durante la mia carriera universitaria.

Vorrei ringraziare di cuore i miei relatori, Gianni Pecora, Daniel Talavera Jiménez e Samuel Reskala Eguiarte, per il loro supporto durante il mio lavoro. Vi ringrazio per tutto l'aiuto che mi avete dato e per tutto il tempo che avete passato con me.

A mamma, papà e Gabriele, grazie per tutto il supporto che mi avete dato in questi anni e per avermi seguito durante il mio cammino. Con la mia carriera universitaria che volge alla fine vi ringrazio di cuore per tutto quello che avete fatto per me. Grazie per aver creduto in me e per avermi permesso di essere qui al giorno d'oggi.

Infine, vorrei ringraziarti, Dalma, per esserci stata quando ne avevo più bisogno, sia nei momenti felici che tristi, e per il tuo supporto durante tutta la mia carriera ed il mio lavoro. Non sarei chi sono oggi se non per la tua presenza. Grazie.

Bibliography

[1] IENAI Space. *IENAI Space website*. https://ienai.space/. Accessed: 2026-03-20 (cit. on p. 1).

[2] IENAI Space. *Internal Documentation of 360™*. 2025 (cit. on pp. 1, 2).

[3] Eda Kavlakoglu Cole Stryker. *What is Artificial Intelligence?* (Cit. on p. 4).

[4] Darrell M. West. *What is Artificial Intelligence?* 2018 (cit. on p. 4).

[5] Michael Chen. *What is Machine Learning?* 2018 (cit. on p. 5).

[6] Dave Bergmann. *What is Machine Learning?* (Cit. on p. 5).

[7] Jim Holdsworth Cole Stryker. *What is NLP?* (Cit. on p. 5).

[8] Jeffrey Erickson. *What is Natural Language Processing?* 2025 (cit. on p. 5).

[9] Cole Stryker. *What are Large Language Models?* (Cit. on p. 6).

[10] Alan Zeichick. *What are Large Language Models?* (Cit. on p. 6).

[11] Dave Bergmann Cole Stryker. *What is a Transformer model?* (Cit. on p. 6).

[12] Alan Zeichick. *What Is Retrieval-Augmented Generation?* 2023 (cit. on p. 7).

[13] Ollama. *Ollama website*. https://docs.ollama.com/. Accessed: 2026-03-20 (cit. on p. 21).

[14] Ollama. *Ollama Structured Outputs*. https://docs.ollama.com/capabilities/structured-outputs. Accessed: 2026-03-20 (cit. on p. 24).