UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Laurea Magistrale in Ingegneria Robotica e dell'Automazione

MASTER'S THESIS

Managing AUV missions through small language models and behavior trees

ADVISOR
Prof. Andrea Munafò

CANDIDATE
Emilio Gigante

ACADEMIC YEAR 2024-2025

Abstract

Autonomous Underwater Vehicle (AUV) missions are constrained by limited onboard computation, harsh communication conditions, and strict energy and time budgets, which makes cloud-dependent Large Language Models (LLMs) unsuitable for mission management despite their strong reasoning capabilities. This thesis proposes a mission management approach based on a compact, fine-tuned Small Language Model (SLM) that enables operators to issue natural language commands, which are transformed into sequences of high-level tasks executed through behavior trees interfaced with the vehicle's guidance system.

The SLM, instantiated as FLAN-T5-base and adapted via Low-Rank Adaptation (LoRA), is prompted with a structured combination of available operations, mission context derived from a dedicated mission memory, and the operator's request, and is trained to output purely symbolic task tags that map directly to behavior-tree subtrees.

The method incorporates an explicit confidence estimation mechanism, adapted from the BS-Detector framework, that combines Observed Consistency and Self-reflection Certainty into a scalar confidence score used to automatically trigger clarification dialogues when the model is uncertain, or to replan after execution failures by exploiting mission logs and failure context. The approach is first validated on a proof-of-concept virtual system inspired by the RAMI Marine Robots competition, where the SLM selects operations such as waypoint acquisition, gate traversal, buoy mapping, and pipeline inspection based on evolving mission knowledge. It is then integrated into the mission manager of a simulated AUV whose control stack is natively based on behavior trees, and evaluated using token- and sequence-level accuracy metrics and confidence statistics on dedicated training and testing datasets. Results show that the SLM-based manager achieves high sequence accuracy, correctly triggers the desired tasks, and assigns higher confidence to correct than to incorrect plans, while offering performance comparable to a rule-based mission manager but with substantially improved accessibility and usability for human operators under realistic resource constraints.

Contents

1 Introduction **3**

2 Recent research **5**

3 Problem and goal description **9**

4 Formal approach **11**
4.1 Method 13
4.2 Model 15
4.3 Training 17
4.3.1 Datasets construction 17
4.3.2 Fine-tuning 18
4.3.3 Metrics and parameters 19
4.4 Model confidence 19

5 Proof-of-Concept virtual system **23**
5.1 Development 24
5.1.1 Model output 25
5.1.2 Mission context 26
5.1.3 Model prompt 29

6 Application of the approach on the AUV control system **30**
6.1 Vehicle control system 30
6.2 Behavior tree manager 31
6.3 Simulation 33
6.4 Rule-based usage of the system 35
6.5 SLM mission manager 36
6.6 Experiments definition 39
6.6.1 Performance experiments 39
6.6.2 User accessibility experiments 40

7 Approach validation **42**
7.1 Model training and validation for virtual system 42
7.2 Model training and validation for real system 60
7.3 Experiments in simulation 78
7.3.1 Performance experiments 79
7.3.2 User accessibility experiments 81

8 Conclusions **83**
8.1 Possible future improvements 83

1 Introduction

Recent advances in artificial intelligence have opened significant opportunities across robotics by automating cognitively simple yet time-consuming operator tasks, thereby accelerating workflows and reducing supervisory burden during complex missions [1, 2, 3]. In particular, natural language interfaces offer a direct bridge between human intent and the robot's mission management stack, enabling operators to specify goals and constraints in everyday language while the system translates them into executable plans and actionable updates [1, 2]. This paradigm can streamline mission oversight, allowing operators to focus their attention on higher-level reasoning, exception handling, and safety-critical decisions [1, 3].

To support such a workflow, two complementary capabilities are essential: first, translating natural language requests into robot-executable commands; second, processing system telemetry and mission state to generate intelligible status updates for the operator [1, 2, 4]. When deployed effectively, the language-enabled system acts as a virtual partner that scales human supervision, which is increasingly valuable in scenarios with multiple vehicles or concurrent tasks typical of maritime operations [3].

There has been notable progress in extending these tools to Autonomous Underwater Vehicles (AUVs), which are critical for tasks such as environmental monitoring of marine and lagoon ecosystems and inspection of submerged human-made structures [1, 3]. Representative efforts include natural-language piloting for AUVs and language-driven mission programming for subsea deployments, illustrating how intent expressed in natural language can be grounded to marine-robot behaviors and mission scripts [1, 2]. These contributions help close the loop from operator intent to situated autonomy under the unique constraints of underwater environments [1, 2].

However, many high-performing Large Language Models (LLMs) are computationally heavy and typically rely on cloud execution, which clashes with underwater communication limitations and the energy and time costs of surfacing for connectivity [1, 2]. Given that AUV missions are often long-duration and battery-constrained, minimizing surfacing events for planning or diagnosis is crucial to preserve mission time-on-task and data collection efficiency [1, 3]. These constraints motivate pushing as much decision support as possible onboard, within the compute envelope of vehicle-class hardware [4, 5].

Emerging evidence suggests that Small Language Models (SLMs) can shoulder targeted planning and control-support roles when carefully scoped and fine-tuned, trading generality for compactness and deployability at the edge [4, 5]. For instance, fine-tuned SLMs have been explored for multi-point navigation and task planning, indicating that useful competence can be achieved for domain-specific pipelines without resorting to cloud-scale models [5, 4]. In maritime contexts, such SLMs can provide onboard, low-latency language grounding, with the option to escalate to larger models only when absolutely necessary [4, 3].

Trustworthiness and calibrated decision support are equally important in operator-in-the-loop systems, where miscalibrated confidence can mislead supervision and degrade safety [6]. Techniques for quantifying and exposing uncertainty in model answers can enhance reliability and guide human oversight, an important consideration when language models mediate mission intent and summarize state under partial observability [6]. Such mechanisms align with the

needs of maritime autonomy, where robust human–robot collaboration depends on transparent, calibrated model behavior [3, 6].

At the execution layer, Behavior Trees (BTs) provide a modular, reactive, and verifiable substrate for composing autonomy, making them a natural intermediary between language-driven mission management and low-level controllers [7]. Their clear execution semantics, composability, and fault-tolerant structure are well suited for integrating language-generated intents as high-level goals while maintaining reliable action sequencing onboard [7]. This architectural separation allows language models to operate at the decision and summarization layers while BTs enforce robust control logic and recovery behaviors in the vehicle stack [7].

Building on these insights, this work investigates applying compact, domain-adapted language models to AUV mission management, with two primary objectives: translating natural language requests into BT-compatible commands grounded in mission memory and sensed context, and generating concise, operator-centered progress reports from telemetry [4, 5, 7]. The approach prioritizes onboard autonomy with SLMs to reduce cloud dependency and surfacing frequency, while retaining the option to defer to larger offboard models only as a last resort [4, 5, 3]. In doing so, it seeks to combine the accessibility of natural language interfaces with the robustness of BT-based execution to improve mission throughput and operator effectiveness in underwater operations [1, 2, 7, 3].

Summarizing, this work answers the following questions:

- How can a small language model be used to build an effective mission management system that operators can interface with?
- Can such a system be effective and at the same time have a resource usage compatible with deployment on the on-board computer of an underwater vehicle?
- Can the model have such an understanding of the context that the missions performed are suitable for it as well as for the request itself?
- How can the model on board the vehicle express a degree of confidence in its response? Does this coincide with the probability that the response is correct?

This work aims to investigate the best way to build, train and prompt a small language model so that it can be used to issue commands in natural language that are transformed into task sequences with a relative degree of confidence. A method is presented that leads to the construction of a system that uses a small language model with a certain degree of effectiveness both in expressing the mission plan and the confidence expressed in it, as well as improved user accessibility.

2 Recent research

As mentioned, the large-scale use of artificial intelligence models, particularly language models, is very recent, so even more recent is the existing research that attempts to use these models for the management, in a broad sense, of underwater vehicles.
Some of the most relevant ones, and those that have most influenced this work, will now be presented, highlighting their achievements and limitations.

Composable and modular autonomy for maritime robotics: Bridging human-robot collaboration [3]
Khorrambakht et al. (2025) propose a novel computational framework for multi-AUV maritime operations, emphasizing improved human-robot collaboration through composable and modular autonomy. Their system deploys a layered, hierarchical multi-agent structure: a supervisory agent leverages large language models (LLMs) to interpret natural language operator commands, while individual LLM-powered robotic agents negotiate tasks, manage robot status, and coordinate actions. Deep Reinforcement Learning (DRL) agents onboard each AUV autonomously control navigation, collision avoidance, and task execution.
A key innovation is the negotiation node, which enables agents to collaboratively allocate tasks—optimizing team efficiency for multi-target scenarios—using both LLM planning and external optimization tools. Cooperative target-tracking algorithms further enhance multi-robot localization accuracy by merging bearing-only measurements, robust against communication latency and packet loss.
Simulation studies with two AUVs inspecting ship hulls validate the framework. Results demonstrate: (i) effective operator-to-robot command translation and task execution via LLM agents; (ii) robust handling of critical events, such as low battery return; (iii) fast, optimal multi-agent negotiation; and (iv) reliable shared autonomy (leader-follower formations) under real-world-inspired constraints. The framework shows promise for scalable, mission-adaptive management in complex underwater environments, laying the groundwork for future research on dynamic exceptions, tighter LLM/DRL integration, and hardware validation.

While Khorrambakht et al. (2025) present a versatile and layered autonomy framework leveraging large language models (LLMs) and reinforcement learning for multi-AUV cooperation, several limitations emerge when considering the specific aims of this thesis—namely, mission management of AUVs through *small language models* integrated with behavior trees:

- **Model Scale and Resource Constraints:** The proposed architecture relies heavily on large-scale LLMs for both supervisory planning and agent negotiation layers. Such models often demand computational resources (both at inference and storage) that exceed the capabilities found on-board typical AUV platforms or low-power mission managers. This diverges from the thesis goal of exploring small, efficient language models suitable for embedded operation.
- **Emphasis on Multi-Agent Negotiation:** The framework excels at collaborative task allocation and negotiation in multi-robot systems. However, it places less focus on single-vehicle mission adaptability, robust failure recovery, and the practicalities of embedding decision logic in resource-constrained mission managers—core aspects of this thesis.

In summary this system proves crucial with multiple agents handling, but the size of the models it uses makes it rely on the cloud connection, which can be, in some cases, a hard limitation.

Oceanchat: Piloting autonomous underwater vehicles in natural language [1]
Yang et al. (2023) introduce *OceanChat*, an integrated AI-planning framework designed to pilot Autonomous Underwater Vehicles (AUVs) using natural language commands. Their system is built upon a closed-loop, three-level architecture that connects a large language model (LLM) with classical task and motion planners. Specifically, the LLM module interprets user-issued natural language commands and formulates abstract goals; a hierarchical task (HTN) planner grounds these goals into viable task sequences with logical constraints; and a motion planner translates the tasks into executable actions that consider real-time perception and AUV dynamics.
To address the uncertainties and dynamic conditions of the underwater environment, the framework incorporates event-triggered replanning utilizing real-time Lagrangian sensor data. The authors present *HoloEco*, a simulation platform based on HoloOcean, for photorealistic and high-fidelity validation. Simulation-based experiments, such as autonomous canyon navigation with the EcoMapper AUV, demonstrate that OceanChat efficiently and reliably grounds human-language objectives into robust, executable AUV missions. Quantitative results show their hierarchical (LLM-task-motion) planning yields a comparable task success rate to LLM-guided motion planning but at just 30% of the computation time. The OceanChat approach advances the prospects for intuitive, robust, and efficient mission management for underwater robotics through LLM-driven natural language interfaces.

By placing the LLM as a high-level interface that translates natural language into abstract goals—and then grounding these goals with a hierarchical task planner and a dedicated motion planner—the system leverages the strengths of language models exactly where they are most effective: semantic understanding and instruction parsing, rather than direct execution. This layered approach ensures robust translation between human intent and executable AUV missions.
Furthermore, the authors' development of the HoloEco simulation environment stands out, providing an advanced and photorealistic platform for validating mission scenarios and facilitating reproducible research. OceanChat is built on high-capacity models, such as GPT-3-based LLMs, which furnish the system with strong generalization and reasoning. However, as with similar recent works, these models currently require a cloud connection to operate, limiting their immediate applicability to contexts demanding full autonomy or strict communication constraints.

Fastnav: Fine-tuned adaptive small language-models trained for multi-point robot navigation [5]
Chen et al. (2024) present *FASTNav*, a method that enables lightweight, adaptive small language models (SLMs) to efficiently perform multi-point robot navigation through natural language commands. Addressing the challenges of deploying resource-heavy large language models (LLMs) on edge devices, the authors propose a three-module approach: (1) parameter-efficient fine-tuning of SLMs with domain-specific navigation data using LoRA, (2) an iterative teacher-student learning scheme where large teacher models guide SLMs to boost their task performance, and (3) a navigation controller that leverages the SLM's output sequence of target points for robot motion execution.
To create robust fine-tuning datasets, the authors employ a human-in-the-loop strategy, ensuring that the models learn both the reasoning process (in natural language explanations) and precise

spatial waypoints in structured (JSON) form. The teacher-student iteration further refines the SLMs, with the teacher generating prompts and feedback, yielding SLMs capable of nearly matching the navigation accuracy of LLMs such as GPT-4.
FASTNav is evaluated both in simulation (using TurtleBot3 in a virtual hospital) and on real robots (Direct Drive DIABLO in laboratory and building environments), demonstrating significant accuracy improvements, reduced inference time (as low as 2.87 seconds per task), and high success rates. Compared to traditional LLM-compression techniques and LLM-based planning methods, FASTNav shows at least a 30% improvement in navigation success rate for models of similar memory footprint, and achieves edge deployment viability with strong privacy and responsiveness. The study highlights FASTNav's potential as a scalable, privacy-respecting, and cost-effective solution for embedded, language-guided robotic navigation.

The approach taken by Chen et al. (2024) in implementing small language models for robotic navigation is particularly fit for on-board implementation, especially in how they leverage parameter-efficient fine-tuning and the teacher-student method to adapt SLMs effectively for real-world tasks. Their model training pipeline—combining natural language explanations with formatted spatial data—demonstrates an innovative strategy to boost reasoning and navigation accuracy on resource-constrained hardware. However, it's important to note that FASTNav is developed and evaluated primarily for structured indoor environments such as hospitals and office spaces, which differ significantly from the dynamic and perceptually challenging nature of underwater missions relevant to this research. Furthermore, the system is tailored to point-to-point navigation tasks, lacking support for a diverse repertoire of high-level actions or tasks beyond simple waypoint following. It also does not incorporate environmental awareness or adaptation in its decision-making process, a capability crucial for robust mission management in the unpredictable underwater domain.

Word2wave: Language driven mission programming for efficient subsea deployments of marine robot [2]
Chen et al. (2025) introduce *Word2Wave* (W2W), a novel framework aimed at enabling real-time and intuitive mission programming for Autonomous Underwater Vehicles (AUVs) through natural language commands. The W2W system features a set of carefully designed language rules and atomic command structures that efficiently map human natural language or speech into computer-interpretable mission instructions. These are implemented by training a small language model (T5-Small) using a sequence-to-sequence approach, allowing the system to generate mission plans robustly and with high efficiency, even on resource-constrained hardware.
A comprehensive pipeline is established, combining automatic training data generation via prompt engineering with GPT-based models, SLM fine-tuning on 1,110 diverse mission samples, and minimal, user-friendly human-machine interface elements for real-time mission visualization and confirmation. Benchmark comparisons show that the T5-Small model achieves competitive accuracy (BLEU 0.879, METEOR 0.813) and nearly doubles inference speed compared to larger models such as BART-Large, while requiring 85.1% fewer parameters.
The authors evaluate W2W both in controlled experiments and actual AUV deployments, showcasing support for widely used subsea mission patterns (e.g., lawnmower, polygonal, spiral, and ripple surveys), and validate the interface via a user study. Participants completed mission programming in less than 10% of the time required by standard commercial AUV software and assigned W2W a high usability score of 76.25. These results highlight W2W's strengths in efficiency, intuitiveness, and ease of use for subsea mission programming. The paper concludes

by outlining future work toward supporting dialogue-driven, multi-shot mission reasoning and broader integration into subsea robotics applications.

Among the works surveyed, the Word2Wave framework aligns most closely with the goals of this thesis and have been especially influential in this own research—in fact, in this work was adopted a similar model to the one they employed. The system is simple and straightforward, which contrasts with the often cumbersome interfaces of commercial AUV planners, yet still enables the execution of a diverse range of standard subsea missions through natural language. The focus on time-sensitive deployments is also highly relevant, as it underscores the practical need for rapid, intuitive mission planning in real-world underwater scenarios.
However, despite these strengths, Word2Wave's approach has some limitations. The generated missions are entirely unaware of the actual underwater environment during execution; that is, all decisions and mission sequences are derived purely from the human-issued commands, rather than from mission progress or environmental feedback. As a result, while the framework excels in making mission programming efficient and accessible, it does not address the challenges of dynamic, environment-aware adaptation that are critical for robust autonomous operation—highlighting a core area for advancement in this own research.

3 Problem and goal description

This work aims to be a solution to this request: *given an AUV mission plan described in natural language, how close can one get to feeding the plan into the system and expect the plan to be completed?*

The starting point is to identify the challenges involved in underwater vehicle missions. These missions must be carefully planned, as deployment usually requires hiring a boat for half a day or a whole day, which is quite costly. Then we must consider that these are electric vehicles with small batteries, as they have to be placed on board together with the computer, and everything has to be watertight, meaning that high temperatures are involved, which do not lead to efficient operation for either the battery or the on-board computer. This, together with safety measures for the vehicle itself and the flora and fauna it may come into contact with, means that these vehicles have to move at a moderate speed, making the mission even more time-sensitive. It should also be noted that underwater communications can only take place acoustically, so for any problem or mission plan update, the robot must return to the surface to come back "online" or, in some cases, it must even be retrieved for troubleshooting or data download.
For these reasons, it is advisable to have a unit on board the vehicle itself that intervenes when the initial plan fails at the executive level. This unit must therefore replace a human operator and their ability to initiate operations according to the needs of the mission, both those predetermined and those that may arise during the mission, to make the vehicle as autonomous as possible.

Moreover, those who operate the vehicle are not necessarily the same people who design it, and therefore need to be trained on how it works. This is where language models can be introduced into the management process. If the operator could be provided with only the mission description and its initial plan in natural language, this would significantly reduce and simplify the work for both parties.
As seen in the previous section, Large Language Models have considerable reasoning abilities, which can very effectively replace certain human operations, including making decisions when a problem arises in the initial mission plan or when, of course, it has been anticipated that such a problem could arise.
However, as mentioned above, these models operate on servers that are completely out of scale compared to the typical on-board computers of these vehicles, but also compared to any type of portable device that can be taken on board a vessel or even just to a pier or coastal site.
In fact, their use is linked to access to the cloud. Access to this cloud for this type of mission may be an option in some cases, for example for lagoon or coastal missions, but much more complicated in the open sea, whether the team of operators is on a boat, whether the vehicle has the ability to connect when it is on the surface, or whether there are solutions that involve surface robots that follow the submerged robot with which they maintain communication by acoustic means. In any case, it seems obvious that the vehicle will have no way of consulting an LLM when operating underwater.

For this reason, it is useful to consider using smaller models that can accompany the model throughout all phases of its mission, and perhaps interface with a larger model only when necessary. These models certainly have reasoning capabilities [4], albeit limited, so the goal is to explore these limits in order to get as much as possible out of these tools.
SLMs are generally used for very specific tasks, so it is important to choose the task that a

model of this type can perform, maximising its usefulness within the management process. As mentioned above, this is primarily to allow the operator to provide the system with commands in natural language, especially if starting from a written plan.
Another interesting task that the model could perform is that of mission supervisor, insofar as it has a certain grasp of the mission's progress and, regardless of the input given to the execution system, is able to recognise whether the initial task objectives have been achieved or, in the event of mission failure, to identify the cause of such failure. Thinking even further one could use the information given by the supervisor to enhance the planning phase.
Let us say, for example, that a mission fails and the problem is identified as not having found the desired objects in the area just explored. the supervisor model could ensure that this information is incorporated into the instructions of the planning model, which, properly trained for such an eventuality, could decide whether to explore a different area, repeat the survey of the same area, or return to the surface to request information, depending on the other information available.

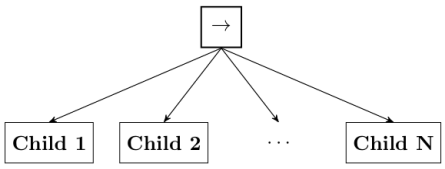
Focusing on the planning model, it would be useful if it could access data derived from the mission's progress, such as which areas have already been explored, which objects have been found, what are the characteristics of the objects examined, or which missions have already been completed.
The aim is to make choices based on knowledge of the mission's history, avoiding unnecessary missions, especially exploration missions, which are certainly time-consuming.

Another issue, with regard to the specific nature of the task performed by the model, is to imagine what the output of this model might be. Since it is intended to replace the operator, it is assumed that the output will in turn be a high-level command, which will have to be collected by a mission management system that we could define as intermediate, since it has to interface with the guidance and navigation systems that directly control the vehicle.
Behavior trees can be a very effective solution because if a name or tag is associated with a certain tree, the role of the model could simply be to trigger that tree, which has a structure that can effectively manage execution, returning a notification of success or failure to the system, and therefore possibly to the model. Given the modularity of the trees, the latter could also be associated with information about which of these modules failed, thus making it possible to trace the reason for the failure.

4 Formal approach

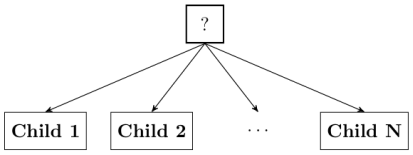
This section will provide a formal structure for the approach that will then be used in the following sections, first on a system designed specifically as proof of concept and then on a simulation system of a real robot.

The method must first be identified as a link between the operator and an execution system based on behavior trees, so it is best to start with a description of the latter to understand what is required of the system.
The article [7] describes briefly how these structures work.
A **behavior tree (BT)** is formally defined as a directed rooted tree structure comprising internal control flow nodes and leaf execution nodes. Node relationships follow standard parent-child terminology, where the root node has no parent and all other nodes have exactly one parent. Control flow nodes must have at least one child. The tree structure is typically visualized with children positioned below their parents, as illustrated in fig.1.
Execution begins at the root node, which generates tick signals at a specified frequency and propagates them to its children. A node executes only upon receiving a tick. During execution, the child returns one of three states to its parent: Running (execution in progress), Success (goal achieved), or Failure (goal not achieved). The classical BT framework defines four control flow node types—Sequence, Fallback, Parallel, and Decorator—and two execution node types—Action and Condition. These node types are detailed in fig.2.



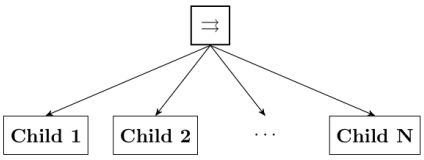
Graphical representation of a Sequence node with *N* children.

(a)



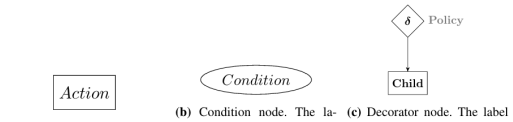
Graphical representation of a Fallback node with *N* children.

(b)



Graphical representation of a Parallel node with *N* children.

(c)



(a) Action node. The label describes the action performed.

(b) Condition node. The label describes the condition verified.

(c) Decorator node. The label describes the user defined policy.

Graphical representation of Action (a), Condition (b), and Decorator (c) node.

(d)

Figure 1: Images from [7].

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Figure 2: Table from [7].

The important thing is to define that these structures manage the execution of tasks, but it is assumed that the system can perform multiple tasks, so there must be **multiple trees**. What the presented method addresses is how the tree to be triggered is chosen so that, starting from the operator's intent modulated as a request, the tree to be triggered to satisfy that request is chosen.
A **language model** is useful in this sense in the first instance because it can receive the operator's intent in the most direct way possible, i.e. as a **request in natural language**.
These models return text, which must indicate which trees should be triggered and in what order. For this reason, in this method, **the model is trained to return a string that corresponds only to a list of items**, each of which is uniquely associated with a tree, so that each of these items can be defined as the name tag of the tree with which it is associated.
In this case, the model acts as a **planner**, because it generates a plan that corresponds to the tasks list.
It should also be borne in mind that, as the robot continues to perform actions or acquires new knowledge in the course of its operations, the same request from the operator may result in a different list of actions depending on how the environment or the robot's knowledge of it has changed.
The information acquired from the environment as a result of the operations performed is then saved and processed in such a way that it can be added to the request when it is made, allowing the model to make the most appropriate choice based on the **context**.
On top of that, the model is also used to express a **degree of confidence in the response** generated. If it is not considered sufficient, the operator will be asked to provide additional information to the model so that it can give a more confident response.
A formal description of the method is provided in the next section.

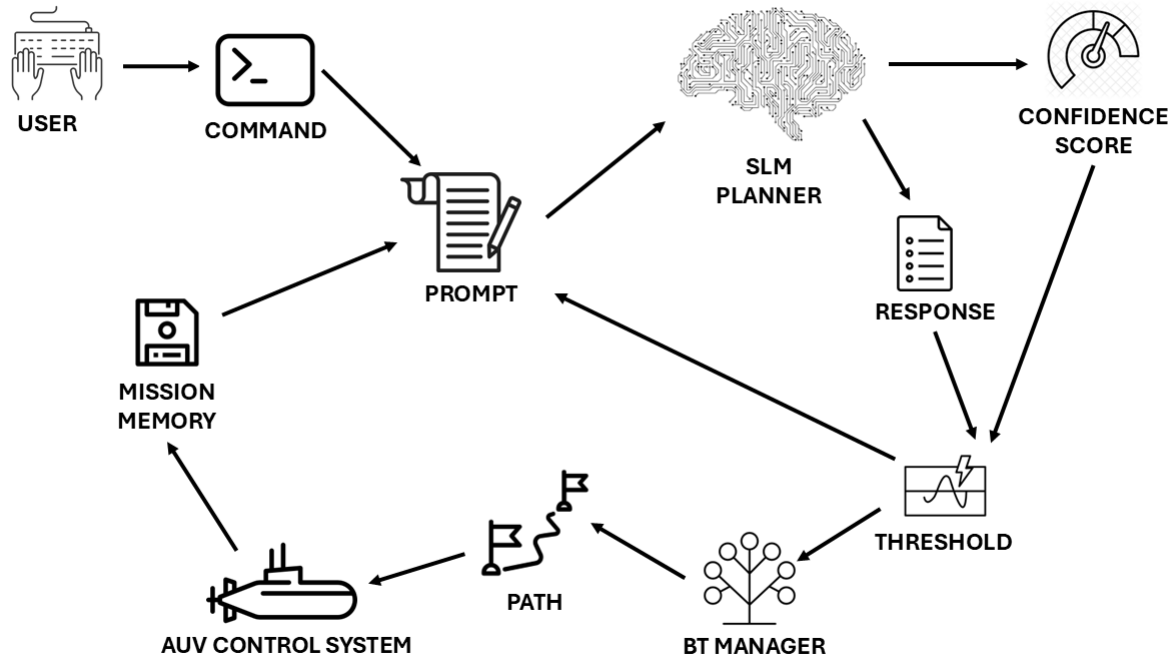


Figure 3: When the user sends a command, it is combined with information from the mission memory to construct the prompt that is fed to the SLM, which returns the response along with a confidence coefficient. If this coefficient exceeds the threshold, the response is processed to trigger a BT; otherwise, it returns to the prompt and the user is asked to add further clarifying information.

4.1 Method

First, the **small language model** (*SLM*) is defined as a function that transforms a text **prompt** (*P*) into a **response** (*R*), also in text form.

$$SLM(P) = R \tag{1}$$

The response coincides with a string of text that corresponds to a list T_R of **tasks name tags** t_i that the system is set to handle.
From now on the symbol $\oplus$ will be used with the meaning of strings concatenation.

$$R = T_R = t_1 \oplus t_2 \oplus t_3 \oplus ... \tag{2}$$

An example is provided to clarify the notation:

- P = ”Close the valve”
- t_1 = ”reach valve”, t_2 = ”grab valve”, t_3 = ”close valve”
- T_R = ”reach valve, grab valve, close valve”

$$SLM(P) = T_R = t_1 \oplus t_2 \oplus t_3 \tag{3}$$

The confidence score, which is described in detail in section 4.4, serves to trigger, if lower than a certain threshold, a request for clarification to the model, in which the user must provide

additional information beyond that which generated the uncertain response.

The structure of the prompt must be constructed following an analysis of the tasks that the vehicle can perform and the context in which it operates.
It is important that the prompt contains the minimum amount of information necessary to make one choice rather than another.

The planner is called upon in three situations that are structurally similar but with important differences.

As mentioned, a prompt is a text string, so every part of the prompt that will be mentioned later must also be a text string, so that the union of the parts corresponds to the concatenation of multiple text strings into one.

In general, the **prompt** (P) should contain a list of **all available tasks** (T_a), a description of the mission **context** (ctx), a **command** (cmd) which is the request that the user makes to the system in natural language, and a more technical **instruction** (ins) which instructs the model to select one or more of the available tasks and provides guidance on how to generate the response. It should be noted that in the following equations is considered implicit, hence omitted, that each field has a name that defines it (as shown in example).

$$P = T_a \oplus ctx \oplus cmd \oplus ins \tag{4}$$

for example:

- T_a = ”find pipeline, check pipeline, reach valve ...”
- ctx = ”No pipeline found”
- cmd = ”Detect the pipeline ID”
- ins = ”Generate a comma separated tasks sequence from the available to satisfy the command”

P = ” Available tasks: find pipeline, check pipeline, reach valve ...
Context: No pipeline found
Command: Detect the pipeline ID
Instruction: Generate a comma separated tasks sequence from the available to satisfy the command

(Each field is appended in a new line, this is equivalent to the comma separation)

This is the main case, but if an uncertain response is generated from a prompt, a clarification mechanism is triggered in the system, which generates a **clarification prompt** (P_c) in which two elements are added to the previous prompt (P_p): one indicating the **previous response** (R_p) considered uncertain and the **clarification** (cla) field where the user is asked again to provide information.

$$P_c = P_p \oplus R_p \oplus cla \tag{5}$$

for example, let us use the previous example as P_p:

- R_p = "check pipeline"
- cla = "You have to find the pipeline first"
- R = "find pipeline, check pipeline"

The other case occurs when one of the tasks initiated by a previous response fails. In this case, if the model is trained for this eventuality, it triggers automatic re-prompting (P_f) without requesting further information from the user, but using information about the task that has just failed, in particular through an element called **failure context** ($fctx$).

$$P_f = P_p \oplus fctx \tag{6}$$

This failure context involves defining which **task failed** (t_f); listing the **mission events** (E) appropriately retrieved from the mission logs and transformed into natural language so that they can be placed in chronological order as sentences; if the failed task was part of a sequence, **the tasks that remained in the queue** (t_q) will also be included, giving the model perspective on the phase in which the task failed.

$$fctx = t_f \oplus E \oplus t_q \tag{7}$$

for example:

- R_p = "find pipeline, check pipeline"
- $fctx$ = "task 'find pipeline' failed. Mission events: survey started. pipeline not found. task failed. Remaining task: 'check pipeline' "
- R = "find pipeline plan B, check pipeline"

It is important that there is a system that manages when the planner is triggered and that takes care of completing the command provided by the user with all the context elements useful for the model to make its choice, this will be called **manager**.

4.2 Model

The model chosen for the planner of the system described above is **FLAN-T5-base** [8]. In the following section its choice is justified and it is describe how it was fine-tuned in order to obtain the desired characteristics.
Building on the insights from prior work [2], it was observed how T5 [9] models are highly effective for translating unstructured mission descriptions into structured control sequences.
T5 is a text-to-text encoder-decoder architecture based on Transformers [10].

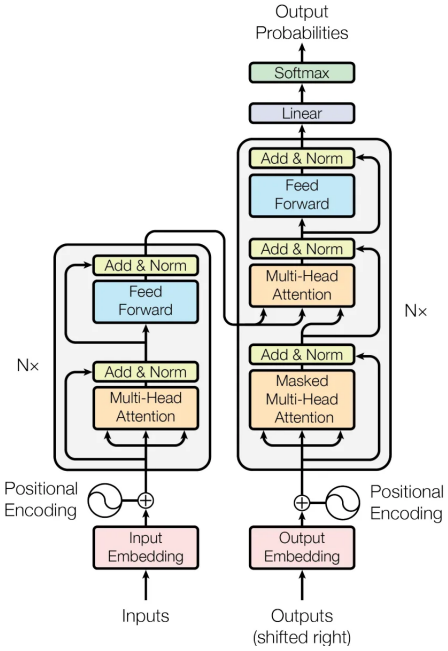


Figure 4: Title: Transformer's Architecture
Source: "Attention Is All You Need" by Vaswani et al. (2017) [10].

However, as this research and experimentation progressed, it was chosen to adopt FLAN-T5 as the core language model. This system is designed to integrate mission memory, context-sensitive command interpretation, and interactive clarification mechanisms—requirements that benefit significantly from enhanced instruction handling and adaptability. FLAN-T5 builds upon T5's foundation by incorporating instruction tuning[11, 8], which exposes the model to a vast diversity of tasks and corresponding instructions during pre-training. This results in advanced capability to process and generalize from new, varied prompts with minimal additional fine-tuning.

In practice, FLAN-T5's improved sensitivity to prompt structure makes it especially apt for this framework, where the model must distinguish between operational commands, leverage mission context, and trigger clarifications when user intentions are ambiguous or the model is unsure. Its superior zero-shot and few-shot generalization[11, 8] reduce the need for extensive retraining while supporting versatile dialog and instruction-following relevant to complex mission management.
Thus, FLAN-T5 not only provides the strong text classification and translation foundation established with T5, but also a better fit for the instruction-driven, context-rich interactions fundamental to the system's architecture.

To consolidate this choice, another model was also tested. Specifically, it was attempted to train it in exactly the same way as it was done with FLAN-T5, but on an LLaMa model of comparable size (more specifically, the comparison was between FLAN-T5-small and teeny-tiny-llama [12]). Although the T5 model has half the parameters of the LLaMa model, the former proved to have considerably better results with the same training, while the latter, despite the training and simple and precise instructions, tended to deviate from the context and invent answers. So the conclusion was drawn, that the model was suitable for the purpose and that the choice it was done at this point was more about the various "sizes" of the model.

The first assumption was trying to use the smallest possible model in order to prioritise computational lightness and inference speed, which is always important during autonomous vehicle missions.
Therefore, the first choice was FLAN-T5-small. It is notable that each model used was fine-tuned to respond adequately to the requests. The small model proved effective in proposing the list of operations by choosing from those available, and it also proved effective in grasping the variety of the command's field of meaning, always returning the correct types of operation. However, it seemed to be completely insensitive to the mission memory description, not taking into account, to give an example already mentioned, the number of buoys present before defining the list of operations to cross the gate. This led me to use a larger model: the small version has about 80 million parameters, while the base version has 250. Using FLAN-T5-base allowed me to include mission memory in the model's sensitivity, to have greater sensitivity to command variations, and subsequently to insert a reliable confidence assessment as described in the previous sections. This model was used both on the theoretical system and on the one applied to the robot simulation framework.

4.3 Training

Once the model has been defined, attention shifts to its training, as T5 is trained on generic corpora, and FLAN-T5 is more specific to certain fields such as law or translation. However, what is asked of this model is extremely specific, so targeted fine-tuning is necessary.
Typically, language models are trained on extensive datasets, which are constructed by processing a considerable amount of data that is recorded specifically for these purposes or for statistical studies in the field of interest.
It is difficult to find data that is useful in this context, so it is advisable to construct a dataset tailored to the characteristics of the vehicle and the environment in which it operates. It is important that the examples replicate each prompt model and that all possible combinations of context and command are included. In particular, the linguistic variations of the command must be numerous and varied enough to allow the model to grasp important patterns and be able to better deduce intentions.

4.3.1 Datasets construction

In this work, a dataset with the following characteristics is constructed:

- A batch is defined as a set of examples that have the same output. The examples per batch must have the same number to prevent the model from having a statistical preference rather than one derived from reasoning, especially for similar cases. In this case, 100 examples per batch were chosen.

- Since there may be different types of prompts that have the same output, their ratio to the total number of examples in the batch must be defined. In this case, if there are examples with classic and clarification prompts in a batch, their ratio will be 70/30; if there are failure and clarification prompts, their ratio will also be 70/30. No other combinations are envisaged.

- This same dataset is used to verify the confidence scores that will be shown in the next section.

- The dataset that is constructed is shuffled and divided into two parts, one containing 80% of the initial examples, which is used for training, and one containing the remaining 20%, which is used for model validation during training.
- A testing dataset is built in parallel with the training dataset to validate the model through training metrics.
- The testing dataset is smaller than the training dataset, specifically 10 times smaller, but it is more challenging for the model as it uses more complex command variations, which are sometimes unnecessarily complex and sometimes even confusing as they add elements that are not necessarily consistent, but this puts the model to the test.
- The testing examples are associated with the corresponding training examples, so they follow the same rules for the prompt type ratio.

4.3.2 Fine-tuning

Once the dataset has been constructed, the model is fine-tuned.
The dataset will contain a certain amount of linguistic variability in the commands, but these will probably be fairly specific, and the verbs and objects of the mission have few synonyms that could make the meaning of the requests ambiguous, so there is a limit to this variability.
As mentioned, the fine-tuning of these models is usually done on large datasets built from real data collected from real situations in the relevant field over long periods of time. Therefore, techniques must be used that allow the model to be trained even with a limited pool of examples.
A parameter-efficient fine-tuning (PEFT) technique called Low-Rank Adaptation (LoRA) [13] is used to overcome this difficulty. It allows model adaptation with a significantly lower computational cost and a lower chance of overfitting on small datasets. LoRA works by injecting tiny trainable decomposition matrices into specific model architecture layers and freezing the pre-trained model weights.
Mathematically, for a pre-trained weight matrix W_0, instead of learning the full parameter update ΔW, LoRA decomposes this update into two low-rank matrices A and B, such that the modified forward pass becomes:

$$h = W_0 x + \frac{\alpha}{r} BAx \tag{8}$$

where W_0 remains frozen, $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d \times r}$ are the trainable low-rank matrices, r is the rank (bottleneck dimension), and α is a scaling hyperparameter. By choosing $r \ll d$, the number of trainable parameters is reduced from d^2 to $2dr$, enabling efficient adaptation even with limited training data.
In this work, the LoRA configuration targets the FLAN-T5-base model [8] with the following hyperparameters:

- rank: controls the dimensionality of the low-rank decomposition, essentially we are choosing the rank of the matrices B and A of the previous example, so an higher r value means we are tuning more parameters;
- α: basically the ratio α/r acts as a learning rate multiplier specifically for the LoRA adapter weights, without adding parameters;

A regularization method called dropout probability in LoRA randomly sets a portion of the trainable LoRA parameters to zero during training in order to help avoid overfitting [14].

According to the QLoRA paper [14] the best use of the LoRA method is to train all the layers. The paper also states that if we are training all the layers increasing the rank does not lead to consistently better results, so it is advisable to keep it as low as possibile.
In this work's case these LoRA parameters were used:

- rank = 8
- $\alpha = 16$
- dropout probability = 0.1

This configuration reduces the trainable parameters from approximately 248 million to 5.9 million (2.36% of the original model), making fine-tuning feasible on limited hardware while maintaining model quality.

4.3.3 Metrics and parameters

Once the training method has been defined, it is necessary to define the metrics that are taken into consideration during training.
As in all fine-tuning processes, the metric that guides it and on the basis of which the weights are adjusted is the **training loss**. However, during training it is advisable to monitor two others, which use the model at each epoch to make inferences on the examples in the validation set, verifying:

- Token accuracy [15]: the percentage of tokens present in the generated output that match those in the target output, which is then averaged across all examples in the validation set.
- Sequence accuracy [16]: if the generated output matches the target output perfectly, giving a full score if so and a zero score if not, the metric score is the average amog all examples in the validation set.

The sequence accuracy metric is the most stringent, but in this context it is also the most useful, since the desired behaviour of the system will only occur if the model returns the exact sequence.
In fact, among all the models saved at each training epoch, the one that is ultimately saved is the one with the highest sequence accuracy score.
The learning rate is decreased linearly as the epochs progressed, getting to zero to the last step.
A safety measure is also adopted to avoid overfitting, specifically an early stopping technique [17] that assesses whether the loss remains too constant for a certain number of epochs. This means that the model is no longer learning and there is a risk that it will fail to generalise and will learn the examples by heart.

4.4 Model confidence

The prompt engineering of the model has been highlighted as an area of focus, as it is a matter of great importance and is often the subject of research [18] [19].
However, no matter how much attention is paid to the construction of the prompt, the results may be inaccurate, inconsistent, incorrect and sometimes even nonsensical [20].

For this reason, an important feature that was chosen to implement in this work's system concerns the model's ability to return, together with the desired output, a value that reflects what we could in fact consider to be a degree of **confidence** in the response it is giving us based on its training.

The research [6] introduces the ironically named **BSDetector** method, which was adapted to this case study.
The method involves calculating two scores: Observed Consistency (OC) and Self-reflection Certainty (SRC), which are combined to obtain the confidence score (CS). This value is used to determine the degree of confidence that the system assigns to the response given by the model. This score is compared with a threshold, which determines whether the system trusts the response or not.
Let us look specifically at how the aforementioned scores are calculated.

Observed Consistency (OC)
The command field (*cmd*) is extracted from the prompt (P) that generated the response we want to evaluate. A function (g_v) is then used which, through the model, generates a certain number of variants of the command ($cmd_{v,i}$) with a certain temperature value (T).
It has been empirically assessed that seven variants are a sufficient sample to express a variety that is good proof of consistency.

$$g_v(cmd, T) = [cmd_{v,1}, cmd_{v,2}, cmd_{v,3}, cmd_{v,4}, cmd_{v,5}, cmd_{v,6}, cmd_{v,7}] = \vec{cmd}_v \tag{9}$$

Temperature is essentially a parameter that, when increased, allows the model to generate a sentence that is ideally identical in meaning but with a greater linguistic difference from the original. It should be noted that this parameter is highly sensitive, and that pushing it too far will result in sentences that not only differ in meaning from the original, but are sometimes meaningless [6].
Empirical tests were carried out to assess the ideal value of this parameter, eventually set at 0.9, in order to make sentences as different as possible while maintaining the same meaning.

New prompts ($P_{v,i}$) are constructed replacing the original command in the original prompt with the command variants.

$$P(cmd_{v_i}) = P_{v,i} \quad \text{for} \quad i = 1,2,3,4,5,6,7 \tag{10}$$

Each of the prompt variants is used to generate a response ($R_{v,i}$) through the model.

$$SLM(P_{v,i}) = R_{v,i} \quad \text{for} \quad i = 1,...,7 \tag{11}$$

Each of the responses generated from the variants is compared with the response to the original prompt. If the response generated by the variant matches the original response perfectly, it is assigned a full score, indicated by a variable (x_i), associated with the response, with a value of 1. If there is no exact match, this value is 0.

$$\text{if} \quad R_{v,i} \equiv R \Rightarrow x_i = 1, \quad \text{else} \quad x_i = 0 \quad \text{for} \quad i = 1,...,7 \tag{12}$$

An average of these values is then calculated by summing them and dividing by the total number. Once converted into a percentage, this value represents the score.

$$OC = (\sum_{i=1}^{n_v} x_i / n_v) \times 100 \quad \text{where} \quad n_v = dim(R_v) = 7 \tag{13}$$

This is, as the name of the score suggests, a good measure of the consistency of the model.

Self-reflection Certainty (SRC)
Once the response has been generated, a certain number of variants of the command are created with very low temperature values, 0.3, similarly to how it was done in eq. 9. This is done in order to help the model in its self-evaluation, not varying to much from the original command. It has been empirically assessed that five variants are sufficient to ensure that the model evaluates the command effectively.

After that, a number of prompts are created that are identical to the one that generated the response, but with the command replaced by its variants, like in eq. 10.

This scoring request (P_s) is added to each of the previous ones:

```
You previously generated this plan step: "{original response}".

Question: Was this step correct, not sure, or incorrect
based on the available operations and current knowledge?

Answer only with:
- 1.0 if correct
- 0.5 if not sure
- 0.0 if incorrect
```

$$P_{SRC,i} = P_{v,i} \oplus P_s \quad \text{for} \quad i = 1, ..., 5 \tag{14}$$

The model will return one of these three values for each of the prompt variants.
The summation of the responses represents the percentage that defines the score.

$$SLM(P_{SRC,i}) = R_{SRC,i} \equiv s_i \quad \text{where} \quad s_i = 0 \wedge 0.5 \wedge 1 \quad \text{for} \quad i = 1, ..., 5 \tag{15}$$

$$SRC = (\sum_{i=1}^{n_v} s_i) \times 100 \quad \text{where} \quad n_v = 5 \tag{16}$$

Confidence score (CS)
The previously mentioned scores are combined simply through a single parameter β.

$$CS = \beta(OC) + (1 - \beta)(SRC) \tag{17}$$

Basically, one should rely on self-reflection certainty to the extent that one trusts the model's judgement.
Since the paper [6], which is based on GPT 3 and 4, uses a beta value of 0.7, meaning it does not place much trust in this capability, in this case one should trust this model even less, as it is considerably smaller in size. In fact, after conducting the necessary checks, it was noticed that

the Self-reflection Certainty score is very unreliable, but it was still decided to give it a little importance in the final score to distinguish between cases with the same OC, so a beta value of 0.8 was set, making it less relevant. It is in some cases useful at least to see the score, it will be shown in the validation section.

Threshold
A threshold must be set in the system for the confidence score that determines the maximum amount of uncertainty we accept in order for the generated response to be accepted. In practice, if the CS is greater than the threshold, the response is processed directly for execution; if it is less than the threshold, the clarification mechanism is activated.
The threshold must be chosen as a trade-off between allowing the mission to run smoothly and taking a safety step by verifying responses defined by the threshold as uncertain.
If the threshold is too high, the user will often be asked to provide clarification for the response generated, making the command issuance phase tedious.
If the threshold is too low, potentially incorrect tasks could start executing immediately, forcing the user to abort the task and start over, which is a waste of time.

The correct approach would be to analyse the average score of incorrectly classified examples compared to correctly classified ones and set the threshold higher than the former. However, experiments carried out on the systems analysed in the following sections show that this is too cautious a measure, often forcing the user to intervene to clarify an output that is already correct. It was therefore decided to set the threshold at 50%, as this value aims at representing the correctness of the response, hence a value lower than 50% would mean that the response is more incorrect than correct.

5 Proof-of-Concept virtual system

This section describes the application of the method seen in section 4 to a system created specifically to demonstrate the potential of the method itself. To this end, it was necessary to identify a benchmark on which to base the concepts and actions to be used.

It was decided to base the competition on the 2025 RAMI Marine Robots competition, in which the Ergo Team from the Department of Information Engineering at the University of Pisa recently participated. this competition is described using the same introduction that the organisation itself provides in the rule book for the competition:

> *The RAMI (Robotics for Asset Maintenance and Inspection) Marine Robots competition aims at addressing surveillance and I&M tasks achieved by aerial and underwater robots in risky and/or hostile marine environments. In these scenarios, human intervention is challenging or impossible since a direct robot-to-operator communication link cannot be generally guaranteed, rendering autonomous decisions indispensable to perform safety-critical tasks. Further, autonomous robots can help in reducing operational costs while ensuring the assets' and the operators' safety. Teams of researchers are offered the chance to test their ideas and showcase their talent in a series of close-to-life vignettes focused around the maintenance and inspection of critical underwater infrastructures. The teams' evaluation process of RAMI involves tasks related to autonomous navigation, data acquisition, detection, classification, manipulation and autonomous decision-making for inspection purposes in underwater scenarios.*

There were three different missions in the competition. It was decided to base this work's system on the first two, which were reproduced below directly from the rule book:

> **Task Description TBM-1**
> Robots will start their mission at the position Start 1 as indicated in Figure 2 and have to:
> - Receive and process the location of a waypoint. The information will be shared by the UAV via CATL messages reporting about the UAV survey and the waypoint to reach (see the related section for the details of the protocol) (related achievements in set A1).
> - Reach the waypoint received from the UAV in underwater autonomous navigation (see 6.2.4, achievement set A1).
> - Pass through the gate (see Figure 2 and see 6.2.4, achievement set A1). - Mapping the coloured buoy area providing a geo-localized map with the buoy positions (see 6.2.4, achievement set A2).
> - Detect the colors of each buoy and react in a specific mode (e.g. turning around the buoy – see later for details) (see 6.2.4, achievement set A2).
> - Survey the pipeline structure (see 6.2.4, achievement set A3).
> - Detect, localize and determining the dimensions of the damage (a red marker) positioned on one of the two pipes connected to one pipeline structure (see 6.2.4, achievement set A3).
>
> **Task Description TBM-2**
> Robots can start their mission either at the position Start 1 (in front of the pipe to

inspect) or at position Start 2 (in front of the pipeline structure to investigate) as indicated in Figure 2 and have to:
-Follow and inspect a pipe (with a length of 6-9 meters) detecting, localizing and identifying coloured markers (red and green) (see 6.3.4 achievement set A1).
-Identifying the manipulation console recognizing and determining the shape of a black number over a red marker positioned on the manipulation console (see 6.3.4 achievement set A2).
-Intervene on the pipeline structure: staying in touch with the damaged pipe, closing a valve and finally grabbing the ring-pole and bringing that to the surface (see 6.3.4 achievement set A3).

The references mentioned in the descriptions are of little importance at this time and will be revisited later if necessary.

An image of the competition area was also included, also taken from the rule book.

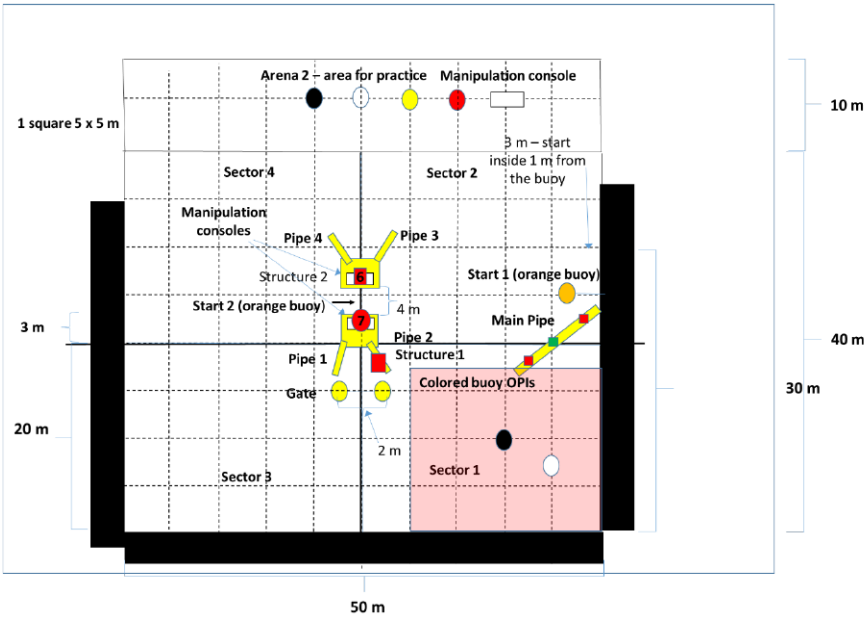


Figure 5: Example map provided by the 2025 RAMI competion, as similar to the one that was actually used in the competition.

Starting from this point, the first step was to build a system that would allow an operator, beginning from instructions of this type, to generate a series of tasks that could bring the mission to completion.

5.1 Development

The question initially asked was how to build a system that would allow the operator to avoid simple logical steps in expressing the command to be given to the robot, since such commands are normally given, to put it simply, by entering specific codes

What the focus was on, therefore, was how this choice is usually set up. There are usually several scripts or a single script based on certain parameters that must be specified before or during the mission.
The missions will certainly always be preconfigured or modified during execution, but it is useful to skip the manual step of the mission selection. This would be done by using a single interface on which to write one's intentions, as if asking another person to do what one is thinking.

Starting from this, the first step was clearly to build this interface, called **chat**, where this command could be given.

The next step was to evaluate what this command meant for the model. What we essentially want is for the model to correctly classify our command, creating through its reasoning a map that links that command to a series of operations that the vehicle will have to perform.
The model itself has no information about what we are going to do, so the request could be interpreted as consistent with underwater robotics as well as agricultural robotics, based on the interpretation of certain terms due to its pre-training.
It is therefore clear that context must be provided to the model, adding important information to the prompt regarding the request that allows the model to make the correct choice.

Talking about context, it is worth specifying that this is not done in general terms. It is of no benefit to the model to know the area it is in or the model of vehicle it is using if this is not important in determining the correct operations that will be included in the request. To get a clear picture, it was therefore useful, after identifying the starting point, to also identify the destination. In other words, leaving aside the prompt settings for the moment, what do one expect the model to provide? What instructions should it be able to give me?

It is worth remembering that this system will be integrated with a high-level mission management system, so the action expressed by the model does not need to be extremely basic, because there will certainly be execution logics that will manage more basic actions and decisions. However, operations must be simple enough to ensure that they can be composed appropriately based on the information known at that time.

5.1.1 Model output

Assuming a generic execution system designed to perform RAMI tasks, the following operations were used:

- **acquire target**: activates the receiver and processes the received waypoint
- **move to waypoint**: Given the waypoint it computes the path and reaches it
- **find yellow buoy**: starts a survey to find a yellow buoy and verify if the whole gate is found at last
- **pass through**: evaluates if the gate position is known, if so computes the crossing path and follows it
- **survey**: generic survey that acquires interesting points, more specifically potential buoys
- **map area**: parses potential buoys to verify them and performs the according move

- **find given pipeline**: starts a survey to find the pipeline structure with the correct ID
- **check yellow pipe**: knowing the pipeline structure it checks both pipes to look for the red marker that symbolizes the damage
- **check main pipe**: finds the main pipe and follows it to check the coloured marker
- **close valve**: grabs the valve, correspondent with the damaged pipe, and turns it off
- **catch ring**: catches the ring and removes it from the housing
- **surface**: emerges to surface (with the ring)

The description of these operations is based on purely logical functions created specifically to simulate the evolution of the mission.

5.1.2 Mission context

This bring us back to the question of how the model should choose one or more of these operations.
Another very important thing, to guide the model's choice, is to provide options, so in addition to the command, a list of all the operations described above will also be provided.
Let's say, for example, that the mission just started and the user asks the system to go through the gate.
Based on the options the user has, it is logical to think that the list of operations expected is:

```
find yellow buoy, find yellow buoy, pass through
```

However, if you already knew the position of one of the two yellow buoys, this list would be excessive, as it would involve carrying out more surveys than necessary, wasting a lot of time.
So one would obviously expect:

```
find yellow buoy, pass through
```

Similarly, if the user already knows the location of the gate, it would simply go through it:

```
pass through
```

It is clear that this logic could easily be implemented in the execution part of the system, and this will indeed be the case in the section illustrating the application to the real system.
However, it is useful to show the sensitivity of the model to changes in the information available to it.

The next step, therefore, is to build a system that creates a sort of mission **memory**.

Returning to the previous example, when one or both yellow buoys are found, they should be saved in this memory so that this information can be retrieved, processed and added to the prompt, as mentioned, together with the available missions and the command.

The actions described above as part of the operations are used to update a state-machine that represent knowledge of the environment in which the theoretical AUV operates. In practice, this is represented by a **World** structure that in turn contains the following structures:

- **Coordinates**: position vectors in NED frame
  - Current coordinates of the vehicle
  - List of waypoints, received externally or from the computed path
- **Sensors**: used in the different operations
  - Camera: used for object recognition
  - Sonar: used for interesting points (potential buoys) search
  - Receiver: used to receive the waypoint from the external source
- **Buoys**: list of found buoys, each of which is described with following features
  - color
  - position
  - image (boolean value: acquired or not)
- **Gate**: defined by the two yellow buoys forming it, hence their position
- **Pipelines**: list of pipeline structures, defined with the following features
  - console marker: which defines the pipeline
  - pipes: two yellow elements on which one may find the damage marker
  - image
- **Interesting points**: list of potential buoys found through the simple survey
- **Main pipe**: defined by the position of one of its ends and the markers on it

The status is updated during the mission, so when the prompt is constructed, it containes updated information.

It is important to remember that we are using a small language model, so we must try to enter the information into the prompt in natural language and as concisely as possible, so that only information useful for making choices is present.

In fact, returning to the example of passing through the gate, it is clear that it is not useful to provide the model with the NED or GPS coordinates of the buoys to enable it to make the correct choice. Instead, it is useful to provide a list of buoys by colour, or, even more relevant to the mission, a count of the yellow buoys separate from the rest of the coloured buoys.

It is equally clear that when a command is given to the model relating to the gate scope, information relating to the pipeline structure is, at best, useless and, at worst, harmful, as it is a source of confusion for the model.

It is important to ensure that the information given to the model is filtered according to what is actually useful.

In large models, or in any case when there is extensive content from which to retrieve information, this is done through Retrieval Augmented Generation (RAG) [21] systems that search the sources available to them, or the web in general, for information consistent with the user's request.

However, this is certainly a very simple system, so a filtering system was implemented, based on the exact recognition of certain keywords in the command, which will lead to the addition of useful context to the prompt.

So, ultimately, here is how information is transformed to make it useful for the model, catalogued according to the field it belongs to:

- Received waypoint information
  - Content: *"Target acquired/ not acquired"*
  - Keywords: *"target", "waypoint", "coordinates", "goal", "destination"*
- Gate information
  - Content: *"0 of 2/ 1 of 2/ 2 of 2 yellow buoys: gate not found/ partial/ found"*
  - Keywords: *"buoy", "buoys", "gate"*
- Pipeline information
  - Content: *"Pipeline found/ not found"*, if the pipeline is found there can be *"Damaged pipe found"*, and if the latter is present then there can be *"Valve closed/ not closed*, and *"Ring untouched/ touched/ grabbed*
  - Keywords: *"pipeline", "pipe", "pipes", "id", "red", "marker", "markers", "damage", "damaged"*
- Main pipe information
  - Content: *"Main pipe found/ not found"*, if the main pipe is found *"Main pipe/ No main pipe markers found*
  - Keywords: *"main pipe", "marker", "markers", "red", "green", "damage", "damaged", "pipe"*
- Interesting points information
  - Content: *"Potential buoys/ No potential buoys detected in the buoy area"*
  - Keywords: *"interest", "interesting", "area", "surroundings", "survey", "object", "thing", "buoy", "buoys"*

The main purpose of the list we have just seen is to show, rather than the actual content of the memory field, the fact that some keywords from different areas coincide. This, like the rest of the system, is designed to test in a very targeted and controlled way how the model reacts to the introduction of information that, as mentioned above, is misleading but which would generally be expected in realistic systems using real data, if only because the aforementioned RAG systems use the same transformers as the models for retrieval.

5.1.3 Model prompt

Finally, we have seen all the elements that make up the prompt, which must be given to the model to generate the list of operations.
As regards the normal course of the mission (4), the prompt therefore consists of:

- **Available operations**: The list of all of the possible operations from which the model can choose to build its list from.
- **Current knowledge**: The knowledge gathered so far about the mission environment and the objects that were found during the mission.
- **Command**: A request provided by the user through the chat.
- **Instruction**: Asks the model to provide a list of operations from the available ones, specifying the format in which they should be given.

Whereas if a normal request generated an output that was assigned a low confidence score, a clarification request would be triggered (5), which would lead to the construction of a prompt like this:

- **Available operations**: The list of all of the possible operations from which the model can choose to build its list from.
- **Current knowledge**: The knowledge gathered so far about the mission environment and the objects that were found during the mission.
- **Command**: The same command that generated the unconfident output.
- **Previous response**: The unconfident output, so the model has a standard of comparison.
- **Clarification**: Here, the user is asked to provide additional information, along with the previous information, which will either lead to the model or confirm a correct choice that they were unsure about, or change their choice.
- **Instruction**: Asks the model to provide a list of operations from the available ones, specifying the format in which they should be given and to pay attention to the clarification field.

At this point, we have described all the parts of this system. As already defined in the previous paragraphs, from the user's point of view, we will now have a system that generates a series of operations based on the command provided. If the model's confidence in its response is high, the operations will be performed, while if the confidence does not exceed the threshold, the user will be asked for clarification so that the model can confirm or modify its response.

6 Application of the approach on the AUV control system

The approach discussed in section 4 was applied to the control system developed by the ERGO team for the Zeno AUV owned by the Department of Information Engineering at the University of Pisa.
Important research work on LLMs is currently being carried out within the department in the field of autonomous underwater vehicles [3].
This research was also validated using Zeno, but on missions where the vehicle remained connected to a computer on the ground.
This is because Zeno certainly cannot connect to the cloud once submerged, but also because its onboard computer has limited capacity, the characteristics of which are listed below:

Main computer: Advantech MIO-3260

- interface: Ethernet
- OS: Ubuntu 18.04
- documentation: `https://www.advantech.com/products/460a67de-a7c8-94dc-0809-336fd757(`
  `mio-3260/mod_9b95d114-45d5-433a-aab5-33a5dd4f4fda`

Payload computer: Udoo x86 II Ultra

- interface: Ethernet
- OS: Ubuntu 18.04
- documentation: `https://shop.udoo.org/udoo-x86-ii-ultra.html`

The system was tested on a computer with Intel(R) Core(TM) i5-10200H CPU @ 2.40GHz, 2400 Mhz, 4 core, 8 logic processors. This hadled also the environment simulation and its graphic visualization, without particular speed problems.
Statistics on resource usage on this machine can be found in section 7.3.

6.1 Vehicle control system

Zeno interfaces with the Robot Operating System (ROS) platform in its Melodic version. The system that determines the vehicle's movement is the **Guidance Manager** node, which receives information from the **Behavior Tree Manager** and transmits data to the **Path Follower** node or the Relative Guidance node. The latter was not used in the simulations described below. Each one of this nodes was designed by the ERGO team members.

The robot's position is determined by its latitude, longitude and depth coordinates. In terms of rotation, only yaw can be controlled, while roll and pitch remain fixed at zero. Descent and ascent are performed without these rotations, which led me to simplify the simulation by effectively creating a two-dimensional control, i.e. controlling only latitude, longitude and yaw.

The guidance system essentially manages two actions:

- **goal**: The vehicle reaches the target coordinates with the shortest possible route, i.e. a straight line, also adjusting the yaw to the desired value once it has been reached.

- **survey**: Coordinates are defined that the vehicle must reach, and the circular area that the vehicle must survey is centred at this point. A series of waypoints are generated that the vehicle must reach in sequence to complete the survey, which can be done starting from outside the area and following a transect path (also called a "lawn mower") or starting from the centre of the area with a spiral-like broken line.

All other actions that will be analysed in the following sections are combinations of these two and of simulated environment perception systems.

The role of the Guidance Manager is to receive instructions to initiate one of the two actions and generate waypoints which, once reached in sequence, determine the completion of the request. The Path Follower aims to create a relative error instance with respect to the next waypoint to be reached, which is checked to zero by the vehicle.

6.2 Behavior tree manager

The Behavior Tree Manager receives instructions on the type of mission to be performed and the parameters, then uses the Guidance Manager to manage the execution of the mission through behavior trees.

Behavior trees are a simple and robust execution logic that is primarily based on checking conditions for the execution of actions, which was described in section 4.
The mission statement is provided in a message called **Set Info Mission Manager** (SIMM) containing the following information:

- mission tag: defines the type of mission to be performed, and therefore which main tree to trigger
- latitude, longitude, depth, yaw
- radius: when surveys are conducted, it defines the area
- yaw flag: defines whether a goal must also be achieved with the correct angular position
- spiral flag: defines whether the survey path should be constructed as a lawn mower or spiral pattern
- color flag: defines the target colour for those types of missions that search for a certain buoy or perform a certain action in relation to it

Below are the types of missions available, defined as mentioned in the tag instance, which trigger a corresponding tree. Their description is based on what actually happens to the vehicle:

1. **go to waypoint**: reaches the desired coordinates and angular position.
   It is successfully completed when it reaches the goal.
2. **pass through the gate**: reaches a waypoint; starts a survey to find one of the yellow buoys; once found, starts a smaller area survey to find the other one; once both buoys have been found, it reaches a waypoint that is equidistant from the two buoys, one metre away from the line connecting them; it reaches another waypoint positioned symmetrically to the line connecting the buoys, effectively passing between them.
   It is successfully completed when the gate is crossed.

3. **survey**: When it reaches a waypoint, it follows a lawn mower or spiral path covering the entire desired area.
   It is successfully completed when it finishes the path.

4. **stop**: This is not a mission and does not trigger any tree, but it stops the guide and declares any ongoing mission 'aborted'.

5. **find the buoy**: reaches a waypoint, starts a survey and only stops when it finds the buoy of the desired colour.
   It is successfully completed when it finds the buoy.

6. **make the move based on the buoy**: If the buoy of the desired colour has not already been found, it behaves as in the previous mission until it is found; once found, it calculates four points around the buoy and reaches them in sequence.
   It is successfully completed when it reaches the last of the four points.

7. **map the area**: Performs a survey in the given area, but it is considered completed successfully only if at the end of the path it found all of the buoys.

This is an example of how the SIMM message looks like, paired with a name tag that defines the unique message:

```
"go to NW goal":
  mission_id: 0
  mission_tag: 1
  latitude: 44.09594791  # NW subarea center
  longitude: 9.86475898  # NW subarea center
  depth: 0.0
  yaw: 0.0
  radius: 2.0
  use_yaw: true
  use_spiral: false
  color: ""
  description: "Goal NW - Northwest subarea center"
```

The original node only included the first four missions, but the last three were added as well (linking also the second one to the perception system) in order to have more variety in the mission manager implementation that uses the language model, which will be described in the next section.

During system execution, data is saved and then used by the mission manager:

- mission objects: defines which buoys have been found up to that point in the mission
- mission logs: all ROS environment logs useful for defining the status of the mission are saved here.

Since the ROS Melodic environment runs on an outdated version of Ubuntu, with respect to the ones which allow AI libraries to run, it was necessary to build a bridge node on which to exchange information on a more recent environment on which a language model can be run.

6.3 Simulation

The vehicle normally uses this relative error to determine the movement of a virtual joystick that replaces the one that should teleoperate Zeno via special software called Mission Manager. For simplicity, this work uses a node that represents Zeno's dynamic model and, receiving the relative error directly, controls it to zero, publishing the navigation status, which in turn surves the perception node and the BT manager node.

In real missions Zeno uses a camera on its snout to detect objects and is equipped with a neural network that recognises them in real-time.
The simulated perception system was created in a very simplified manner, using 2D Stage simulation software, whereby simulated objects, in this case buoys, are defined only by their coordinates and colour.
If the buoy is less than one metre from the vehicle and falls within its field of vision (i.e. within a 120° angle centred on its snout), the buoy is considered recognised, including its colour. These values were considered reasonable and consistent with some recent work in the field [22] [23].
A visualisation node displays the progress of the mission on the Rviz software.

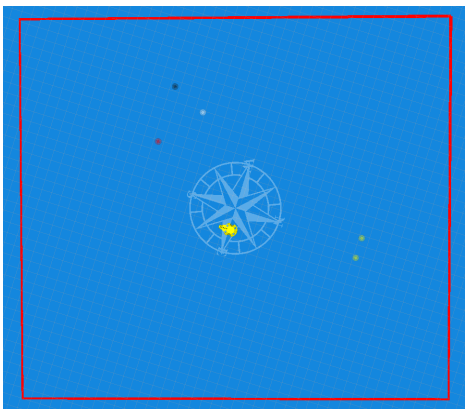
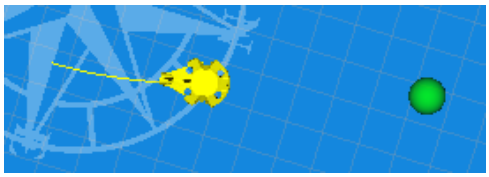
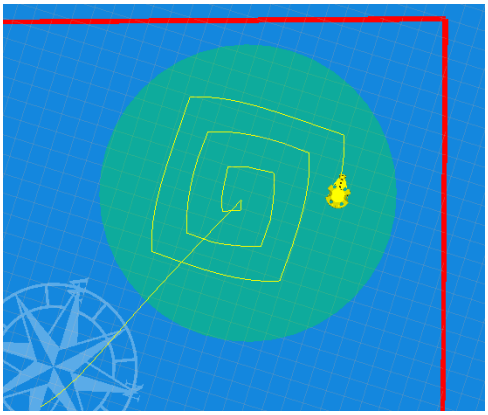


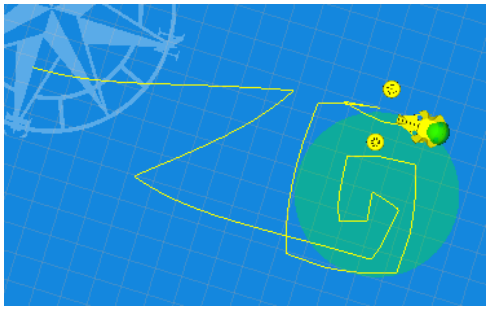
Figure 6: The simulation was monitored through Rviz. The yellow vehicle represents Zeno, the two yellow buoys in the northeastern area are the gate, while the buoy area is in the south-western sector. The red lines delimit the safe area.



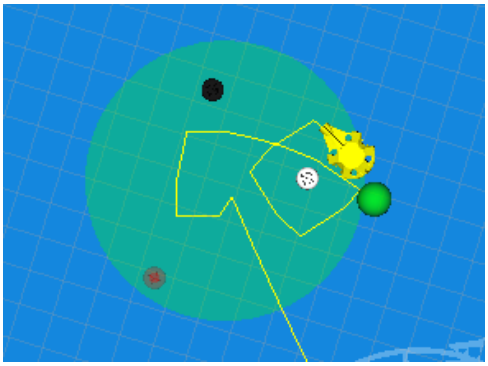
(a) The green sphere represents the goal coordinates in the map, Zeno reaches it in a straight line.



(b) The green circle represents the area that Zeno has to survey, doing it in this case with a spiral pattern.



(c) Zeno attemps a lawn mower pattern to find a yellow buoy, when it finds the first one it starts a spiral survey, after finding the second a waypoint is put in front and behind the gate.



(d) Zeno ignores the first buoy because it was not of the specified color, then when it finds the white buoy it performs the move requested, in this case a diamond pattern around the buoy.

Figure 7: The RViz graphical representations of the mission types configured in the system are shown here.

Limits

The simulation is designed in an oversimplified manner, as its purpose is mainly to demonstrate how the SLM mission manager works, rather than to provide an accurate simulation of the missions that the system will have to face in a real scenario, or at least this work does not go that far.
Here are some simplifications that the simulation has compared to a real scenario:

- The simulation takes place in 2D, so the vehicle's movement occurs entirely at zero depth.
- Buoys are the only objects on the map that are not vehicles themselves and are defined solely by their position and color.
- No obstacle avoidance system has been implemented, so the simulated vehicle may intersect with the buoy at certain points.

- No uncertainty is implemented in the perception system, so the object will always be recognised if its coordinates are within the vehicle's field of vision, which is based only on position distance and pose of the vehicle.
- There is a safe area within which the mission must be carried out. If any of the objectives or the path created for the vehicle have points outside this area, the mission will be cancelled.

6.4 Rule-based usage of the system

This section describes how to use the system described above by interfacing directly with the behavior tree manager, without using the SLM mission manager, which will be covered in the next section.
As mentioned above, the BT manager requires message SIMM to start a mission. A configuration file has been created that associates a name tag with a unique message SIMM, so that once the system has been initialised, a programme can be used with the name tag of the desired mission as an argument, and this will send message SIMM to the BT manager, which will start the execution.
This system is simple and effective, but it should be noted that there is no assistance to the operator other than the information that can be read from the mission terminals, so the operator must decide what to do on a case-by-case basis based on his assessment of the mission's progress.
Its limits are clearer looking at the experiments in section 7.3

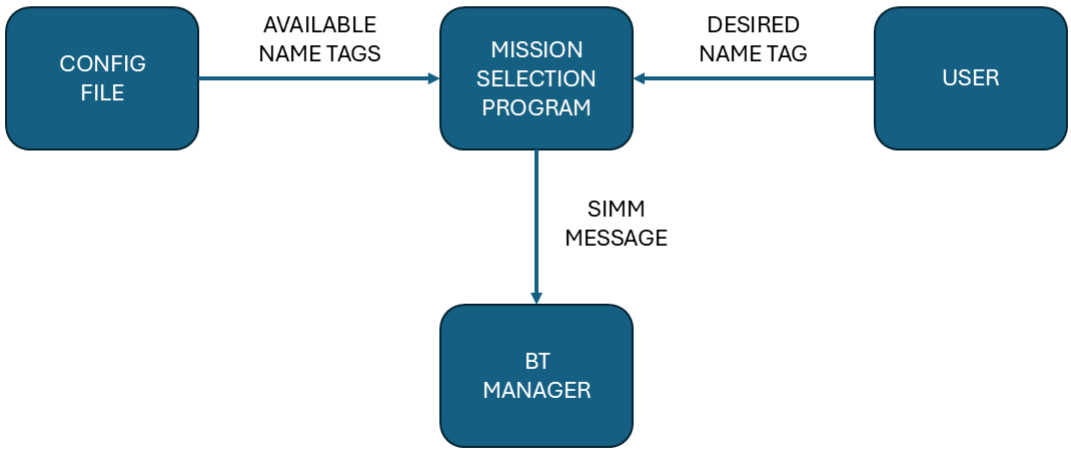


Figure 8: The user must understand the meaning each missions, as it is required to use the programme to send the mission to the BT manager using the name tag of the desired mission.

6.5 SLM mission manager

The framework described in this section is the application of the manager element seen in section 4 which uses the planner (the model) to send instructions to the vehicle control system seen previously.

The system consists of three elements:

- **chat**: the mean by which the user provides text for the system.
- **planner**: constructs the prompt and uses the SLM to generate the sequence of missions and the confidence score associated with each response.
- **manager**: receives information from the ROS environment, which it provides to the planner for the construction of prompts, receives prompt responses that define the mission that is sent to the ROS environment.

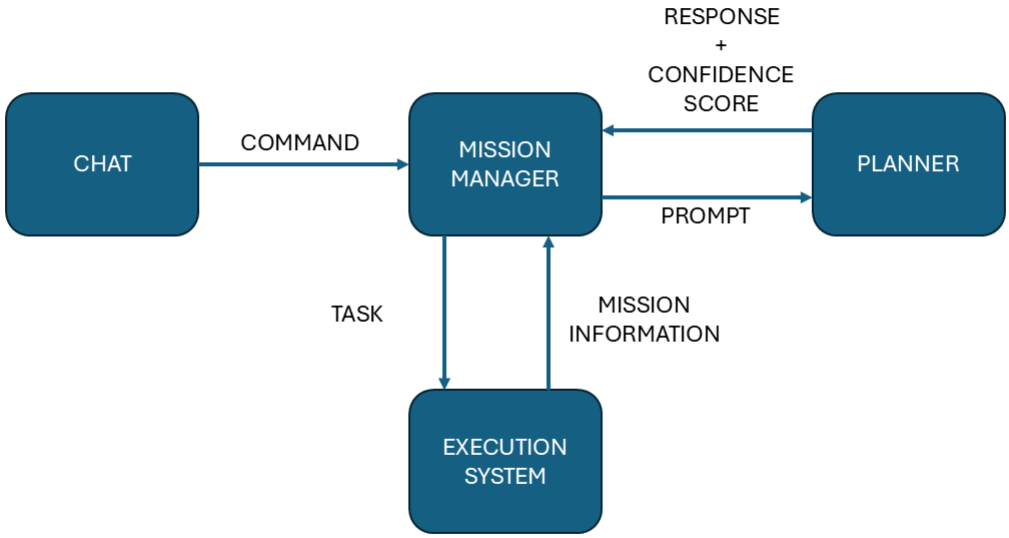


Figure 9: The mission manager is responsible for handling the flow of information described in figure 3. Specifically, it sends the user command and mission information to the planner, and also receives the model's response and processes it for execution.

Generic missions and missions similar, as far as possible, to the RAMI rules have been defined:

- four missions to reach the north-west, north-east, south-west and south-east quadrants of the map
- four missions to survey the entire area of the same quadrants
- a mission that reaches a waypoint, which in the RAMI mission should be obtained externally

- a mission for crossing the gate
- a plan A and B to scan an area to find all the buoys, the reason for the double plan will be explained later
- a plan A and B to perform the corresponding move for each buoy

To improve the versatility of mission selection, the configuration file is reloaded each time the model is called by the manager, so that the user can change the parameters of a mission in real-time. If necessary, also add a new mission with a new name tag.

The model is trained on certain examples that it may receive but for which the vehicle system is not configured, to return the word "skip", which is recognised by the manager as a command to do nothing, or, if present within a sequence, to move on to the next mission.
For example, when asked to parse the pipeline structure, the model, which does not have a mission among its options that can satisfy this request, would basically respond with a random mission, forcing the operator to stop operations, whereas in this way the model decides autonomously to respond to the request with inaction.

At this point, let's proceed by analysing the possible prompts that can be constructed for the model.
Let's start with the classic prompt (4):

- **Available missions**: all the name tags in the missions configuration file, plus 'skip'
- **Memory**: field that contains two subfields:
  - Buoys found so far: the colours of the buoys found up to that point in the mission
  - Previous missions: list of missions successfully completed up to that point
- **Command**: given by the user
- **Instruction**: asks the model to provide one or more missions from the available ones

Here the memory field plays the role of context. In a competition such as RAMI, which takes place within a limited time frame, the memory of the missions carried out since the start of the session and the buoys found may also be superfluous. However, let us imagine a multi-robot control scenario, in which a single operator has to manage several AUVs. In this case, supervising each individual robot could prove difficult, and the robot's ability not to perform a mission that has already been carried out, perhaps even by another robot, could save time and resources. The same applies to objects found.
Therefore, it would be appropriate for the model to be trained to recognise the presence of the mission in the memory and to choose, depending on the mission planning upstream, a different mission, which may also be a plan B for the same mission, or to actually return the same mission to which the user can change the parameters.

Precisely for this latter case, there is a mechanism in the mission manager that verifies that the mission generated is not among those already completed. If the mission has already been completed, it creates a new prompt identical to the previous one, but adds two fields:

- **Previous response**: the previously generated response that matched the already completed mission.
- **Repeat mission**: input provided by the user which, if affirmative, effectively repeats the same mission, perhaps giving the user time to reflect and change the parameters. If negative, it returns the expected response in the case of training, which, as mentioned, may be a subsequent mission, a different mission, or a plan B for the previous one.

The system provides for uncertainty management with a function identical to that described in section 4.4, whereby when the model's confidence score is below the 50% threshold, the clarification mechanism is triggered, which constructs the clarification prompt (5):

- **Previous response**: Depends on the previous output, we want either to confirm or to change.
- **Clarification**: Has to provide either some variations of a confirmative sentence or some additional info that allows the model to change its response.

The reason why there is a plan B for 'map buoy area' and 'move' missions is that a failure management mechanism has also been implemented, which in both cases can occur, for example, when no buoy of the desired colour is found in the area where the survey was carried out. In this case, the system triggers an automatic reprompting mechanism that calls the model in question to decide what to do. The prompt in the event of failure (6) takes the form of the classical prompt plus a **failure context** field (7) that containes this information in the said example of the failed "find buoy" mission:

```
Failure context: Mission 'make move black A' failed.
Mission 1 started. | Mission reset initiated. | Move mission started. |
Checking mission memory for buoy information. |
Loaded 0 buoys from mission memory. | Buoy position loaded from memory. |
Buoy not in memory - executing FindBuoy mission... |
Survey operations are underway. | System is configuring parameters. |
Initiating survey operations. | Guidance is ready. |
System and guidance are ready. | Survey operations are underway. |
Survey was completed successfully. |
Mission FAILED - target buoy NOT found | The mission failed.
Remaining missions were: make move red A, make move white A.
```

This provides us with information about the failed mission, the events that caused it to fail, and the remaining missions, which in turn provide information about the stage of the mission at which the queue was interrupted.

In this case, the model has been trained, for example, to respond with a plan B, which in practice consists of the same mission but in a different area of the map where we think those buoys are actually located. It should be noted that one could use an exact replica of plan A as plan B to be sure that there are no buoys in that area, and explore another area as plan C. One could also decide that if no buoys are found in that area, then move on to the next mission in the queue, for example, checking first that the other buoys are not there before changing area.

The manager shows for each completed mission the events that occured during it to the user, but

it is clear that these events are useful to the model only to the extent that this helps the model making the correct choice. In this case the mission events are not so numerous, so it is simpler to add all of them in the prompt. But the longer is the prompt, the more the model struggles, so one could think of using a language retrieval system [21] to select only those events that are important to allow the model to make the correct choice.

This system works, but it is presented not so much as a tool immediately useful for the vehicle management, but rather as a way of exploring the capabilities of small language models for tasks that were previously performed through other tools and which must therefore be adapted to the use that must be made of them based on the context.

6.6 Experiments definition

Experiments were carried out on the simulation system described in section 6.1, evaluating the differences between the success rates of the same sessions performed using the rule-based system initially envisaged and the mission manager described in section 6.5.
Two types of experiments will be conducted:

- Performance experiments that will define the success statistics of the two systems in comparison with the operator who has full knowledge of how the system works.
- User accessibility experiments in which different operators are asked to use both systems to assess their ease of use.

6.6.1 Performance experiments

The **success rate** is defined as the number of all successfully completed operations out of all the queries sent to the system, in all sessions.
This for both the SLM mission manager system and the rule-based system.

When the benchmark is the mission success rate, we can consider a single session as a test for both the system that uses the SLM mission manager and the rule-based system.
This is because the SLM manager works on top of the BT manager, essentially replacing the operator. Therefore, by analysing a single session, we can assess which errors are due solely to the SLM manager and which are due to the system more generally, in which case they would be considered errors on both sides.
For example, if the error were due to a model response that does not correspond to the correct mission to perform the task, then this would be considered an error only for the SLM manager, whereas if the chosen mission were correct but there were an error in execution, it would be considered a failure for both systems, because it would have occurred regardless of the presence of the SLM manager.
With regard to sessions, these were the considerations adopted in carrying out the experiments:

- Three sets of sessions are performed.
- The first two sets of sessions will follow the RAMI guideline, whereby missions within a session will always be performed by first going to the waypoint, then passing through the gate, then mapping the buoy area (with planned failure of plan A and success of plan B) and performing the moves for each buoy.

- For the first set of sessions, the commands used are taken from those used during training, using a different command for each session (for each expected response).
- For the second set of sessions, the commands testing commands are used, always using a different command for each session.
- The third set uses other commands taken from the testing commands, but uses a different order of missions for each session. It should be noted that if the moves performing mission is not preceded by the map buoy area mission, the latter becomes superfluous, as the buoy performing mission will first search for the buoys, failing its plan A and succeeding in plan B.
- The computational resources used by the manager are saved for each session.
  - Average CPU usage during all the sessions (%)
  - Peak CPU usage for each session (%)
  - Average RAM usage during all sessions (MB)
  - Peak RAM for each session (MB)

  The mean, standard deviation are computed for all the sessions, minimum and maximum registered.
- For each mission sent by the mission manager to the BT manager via the bridge, the transmission latency is calculated. The mean and standard deviation are then also calculated.
- Confidence scores will be shown for the first set, then for the first two sets, and finally for all three sets. Resources and latency results will be shown for all three sets.

6.6.2 User accessibility experiments

The idea is to give operators the RAMI instructions (they were mentioned in section 5) and a brief description of how the systems works, and ask them to complete as many tasks as possible until each one decides that they have completed the mission.
In this case the rule-based system and the SLM manager are tested separately, because the goal is to test the full experience of the operators using both different approaches.
Unfortunately there was not enough time to organize for five people to test the system, the only human operator involved was me personally but I didn't include myself in this test since it needs to be performed by a person that doesn't know the whole system to prove its accessibility. Therefore, Chat GPT was asked to impersonate each of these five operators, giving it only the information that a human operator would have had.
Here are the details on how the experiments were conducted:

- Each operator is provided with the RAMI TBM1 task list.
- To use the rule-based system, the operator is simply told that the missions they can send are those in the configuration file and how to use the programme to send them.

- To use the SLM manager, operators are asked to make requests to the system in natural language. They are informed that if the clarification mechanism is triggered, they must provide information to confirm or change the model selection. They are also informed that when the model responds with "skip", it means that the system is unable to handle the request.

There are some considerations to be made regarding the use of Chat GPT as a substitute for a human operator:

- LLMs lack the embodied experience of operating physical robotic systems, missing the real-time spatial reasoning and sensorimotor feedback that human AUV operators develop through direct system interaction [24].
- ChatGPT demonstrates systematic biases in task interpretation that differ from human operators, particularly in how it prioritizes competing mission objectives and handles ambiguous task specifications provided in natural language [25][26].
- LLMs fail to respond consistently when mission descriptions are reworded or when the RAMI task list is presented with minor variations, whereas human operators develop robust mental models that generalize across equivalent formulations [27].
- ChatGPT cannot replicate the full diversity of operator strategies for interfacing with the rule-based system, tending instead to produce archetypal interaction patterns that may not represent the true range of accessibility experiences [28][29].
- LLMs struggle to maintain consistent reasoning about sequential mission tasks and system state changes, particularly when the clarification mechanism is triggered and operators must verify or revise model selections across multiple interaction cycles [30][31].
- Using ChatGPT-simulated operators in a blind evaluation of system accessibility violates experimental assumptions about unconfoundedness, since the model may have different priors about mission planning than naive human operators unfamiliar with the system [32].
- ChatGPT systematically biases its responses toward completing all available tasks, whereas human operators naturally exhibit fatigue, frustration thresholds, and context-dependent motivation that affect accessibility perception [33].
- LLMs fail to exhibit the problem-solving variance and creative workarounds that human operators spontaneously generate when encountering system limitations, showing instead deterministic responses that underestimate or overestimate system usability [29].

7 Approach validation

Fine-tuning, as well as testing, was performed using a Python 3 based Jupyter notebook running on Google Colab with T4 GPU acceleration.
The method used to obtain the results is described in section 4.3.
The results for section 5 will be presented first, followed by those for section 6.

7.1 Model training and validation for virtual system

The characteristics of the training dataset are listed below:

```
TRAINING DATASET
  - Total examples: 2015
  - Unique output patterns: 21
  - Average examples per pattern: 96.0
  - Max examples per pattern: 100

TRAINING EXAMPLES PER PATTERN
  - Plan: 1. acquire target 2. move to waypoint: 100
  - Plan: 1. catch ring: 100
  - Plan: 1. catch ring 2. surface: 100
  - Plan: 1. check main pipe: 100
  - Plan: 1. check yellow pipe: 100
  - Plan: 1. clear memory: 15
  - Plan: 1. close valve: 100
  - Plan: 1. find given pipeline: 100
  - Plan: 1. find given pipeline 2. check yellow pipe: 100
  - Plan: 1. find given pipeline 2. check yellow pipe 3. catch ring: 100
  - Plan: 1. find given pipeline 2. check yellow pipe 3. close valve: 100
  - Plan: 1. find yellow buoy: 100
  - Plan: 1. find yellow buoy 2. find yellow buoy: 100
  - Plan: 1. find yellow buoy 2. find yellow buoy 3. pass through: 100
  - Plan: 1. find yellow buoy 2. pass through: 100
  - Plan: 1. map area: 100
  - Plan: 1. move to waypoint: 100
  - Plan: 1. pass through: 100
  - Plan: 1. surface: 100
  - Plan: 1. survey: 100
  - Plan: 1. survey 2. map area: 100
```

The training parameters were empirically tuned through a trial and error process adapted to the dataset used and are reported below:

- Number of epochs = 10
- Batch size = 8
- Gradient accumulation steps = 2
- Initial learning rate = $3.5 \cdot 10^{-4}$

- Warm-up steps = 50
- Early stopping patience = 3
- Early stopping threshold = 0.05

Which led to this development:

Epoch (Step)	Training Loss	Validation Loss	Token Accuracy	Sequence Accuracy
1 (101)	11.1701	0.9470	0.9378	0.3226
2 (202)	0.5075	0.0985	0.9631	0.4888
3 (303)	0.1564	0.0568	0.9787	0.6998
4 (404)	0.1106	0.0409	0.9853	0.7916
5 (505)	0.0815	0.0265	0.9894	0.8486
6 (606)	0.0678	0.0208	0.9937	0.9057
7 (707)	0.0575	0.0199	0.9940	0.9156
8 (808)	0.0500	0.0169	0.9939	0.9181
9 (909)	0.0442	0.0154	0.9954	0.9305
10 (1010)	0.0423	0.0153	0.9950	0.9256

Table 1: Training and validation metrics per epoch (1612 training examples, 403 validation examples).

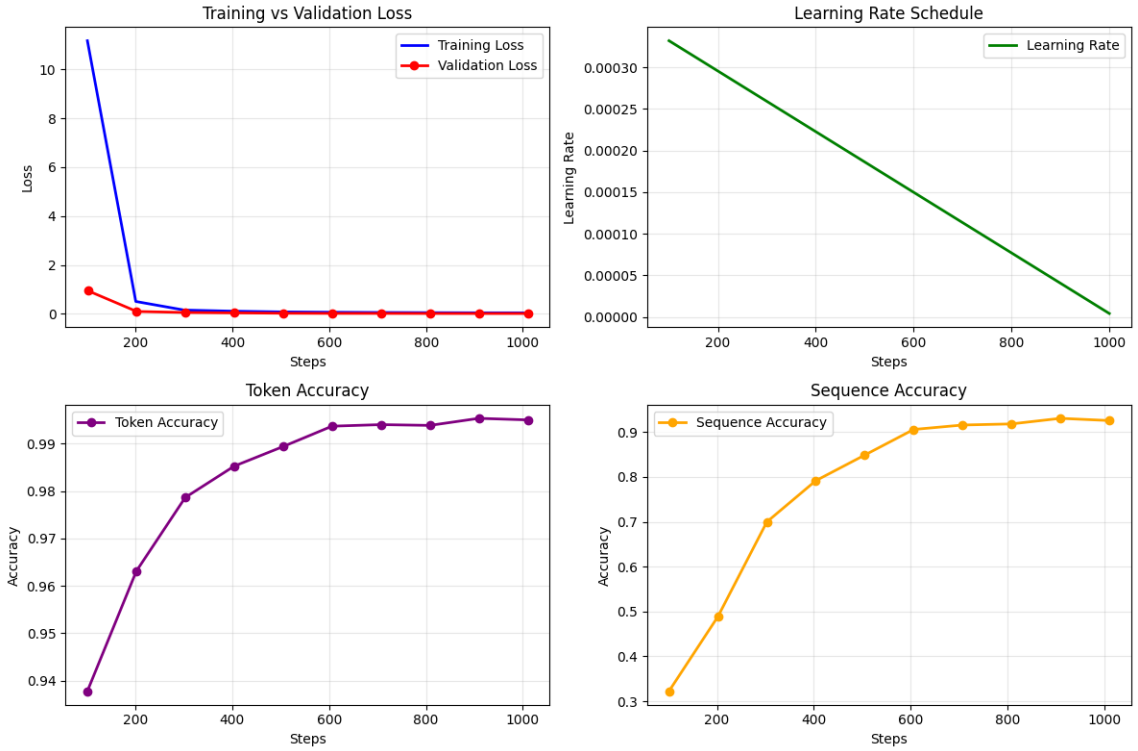


Figure 10: Metrics trend during training of the planner for the PoC system.

From the table 1 and the plot 10 we can see that good training loss levels and even better validation loss values are achieved. The fact that the validation loss is lower than the training loss is actually unusual, but it is due to the similarity between the examples in the training and validation datsets, since they are split from the same one and the validation dataset is much smaller.

Furthermore, the decline in training loss does not stop, which would otherwise have triggered the early stopping mechanism, so we can assume that the training has been effective, which we will verify shortly.
We can also see that the token accuracy is very high, while the sequence accuracy, which we recall is the crucial metric in this case, tells us that more than 9 out of 10 examples are classified correctly.

Just to give an example, the training variations of the command that leads to crossing the gate are:

```
"variations": [
    "pass through the gate", #1
    "move through the gate", #2
    "pass through", #3
    "cross the gate", #4
    "pass across the gate", #5
    "cross through the gate", #6
    "move across the gate", #7
    "navigate through the gate", #8
    "proceed through the gate", #9
    "advance through the gate", #10
    ...
```

For test variations for the same command are:

```
    "test_variations": [
        "execute gate transit with both reference positions
         maintained in visual acquisition", #1
        "proceed through the gate aperture maintaining equidistant
         spacing from the flanking posts", #2
        "gate passage requires navigation through the defined
         reference channel", #3
        "cross the gate boundary defined by the positioned anchors", #4
        "complete gate traversal by passing through the yellow
         signal channel", #5
        "transit authorization requires passage through both gate
         reference positions", #6
        "the gate defines a specific transit corridor that must
         be followed exactly", #7
        "navigate the gate passage while maintaining proper
         clearance from the positioned guides", #8
        "gate crossing protocol requires directional passage
         between both defined positions", #9
        "establish gate clearance by transiting through the complete
         signal perimeter" #10
        ]
```

As can be seen, some of these commands are absurd in the context in which the model is used, but the task of the model, at the limit of its capabilities, is still to map a request to the most

suitable available operation.

The overall results of the testing notebook will now be shown first on the training dataset, which is not a test in itself as the results should logically be good, but which helps us to have a yardstick for comparison, especially with regard to confidence scores.

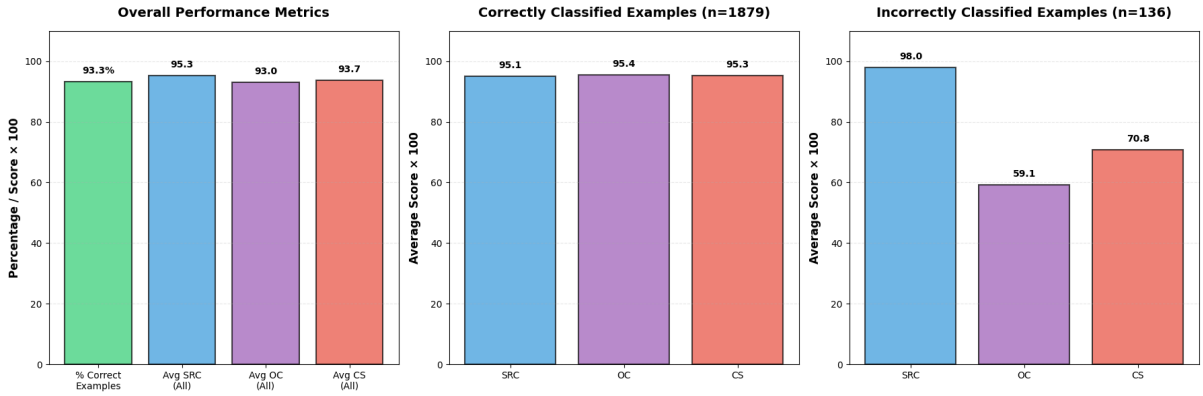


Figure 11: PoC system test with initial dataset: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

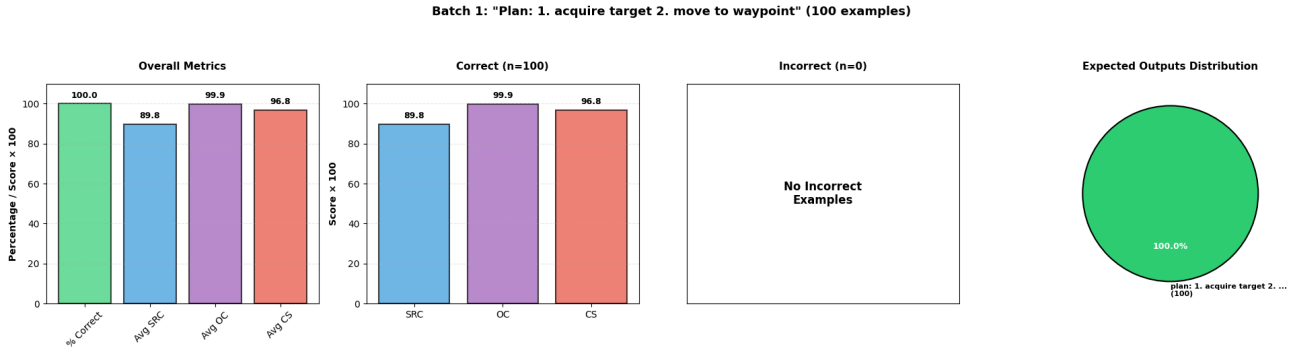


Figure 12: PoC batch 1: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

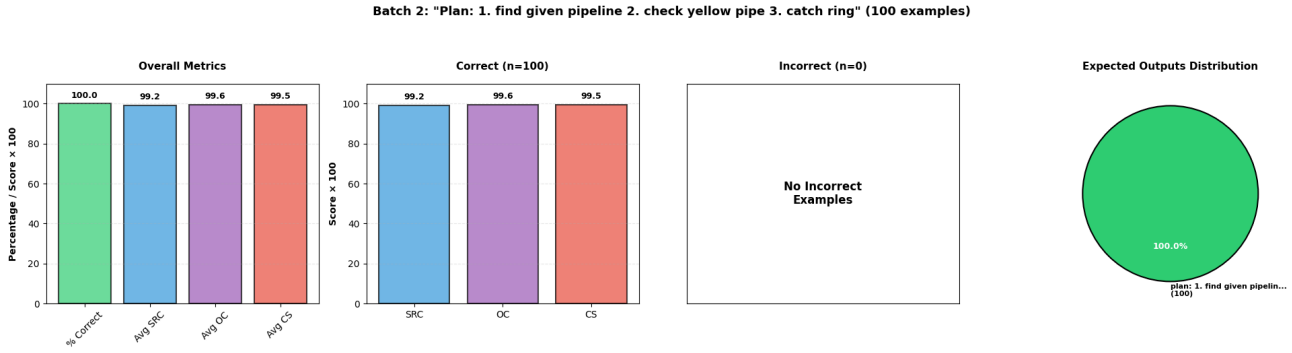


Figure 13: PoC batch 2: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

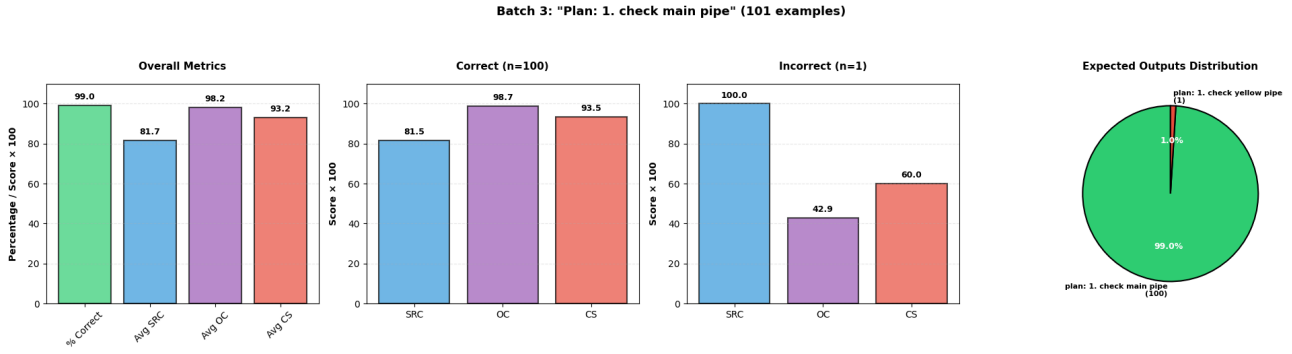


Figure 14: PoC batch 3: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

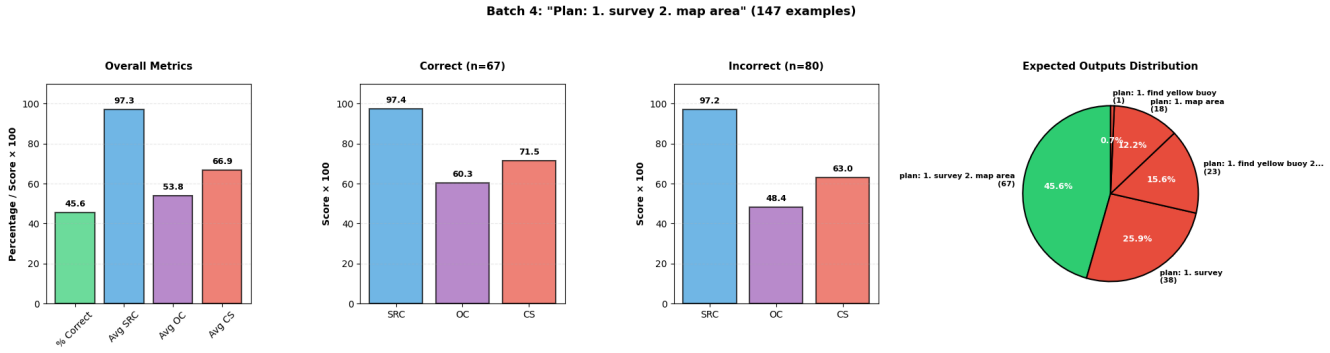


Figure 15: PoC batch 4: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

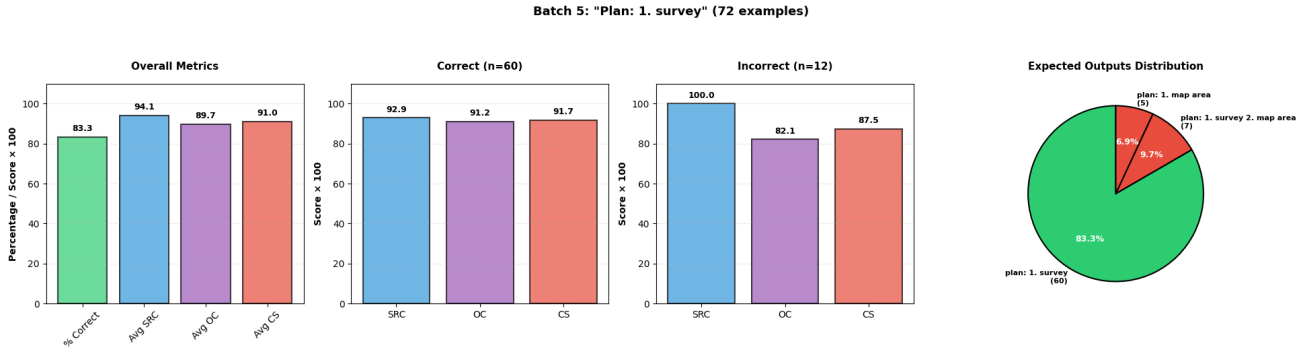


Figure 16: PoC batch 5: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

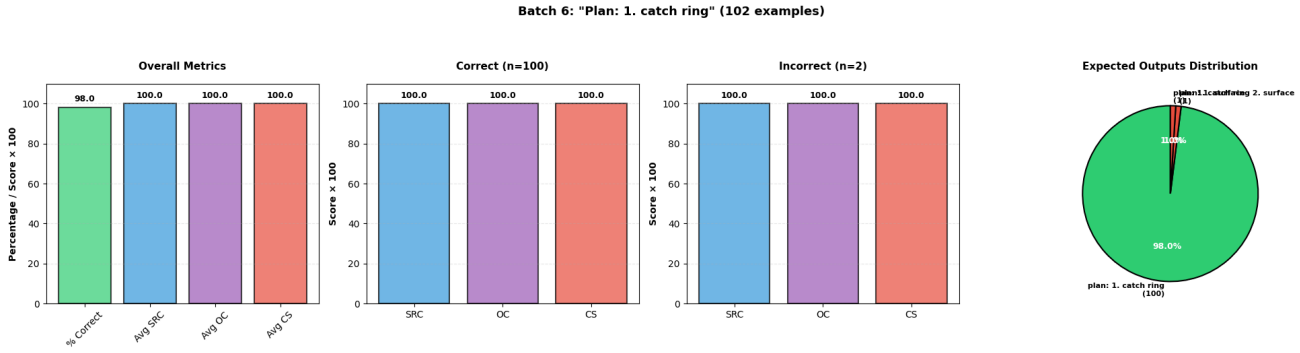


Figure 17: PoC batch 6: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

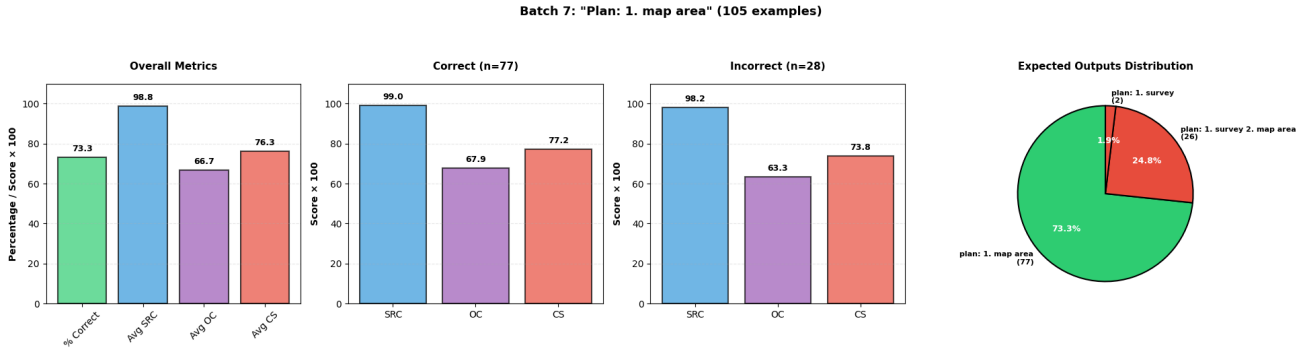


Figure 18: PoC batch 7: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

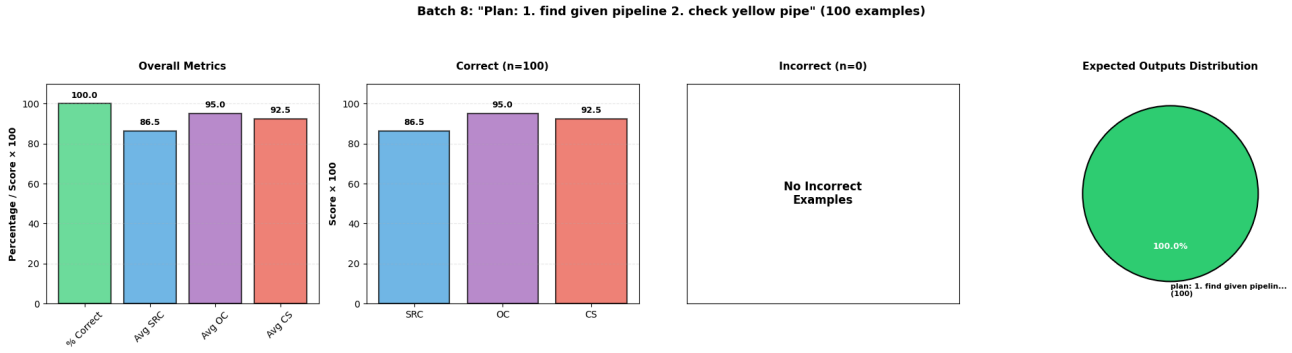


Figure 19: PoC batch 8: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

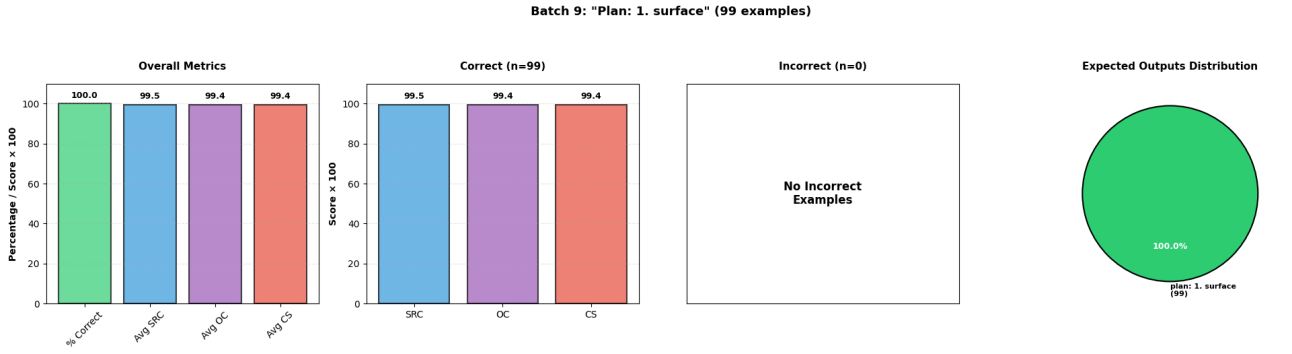


Figure 20: PoC batch 9: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

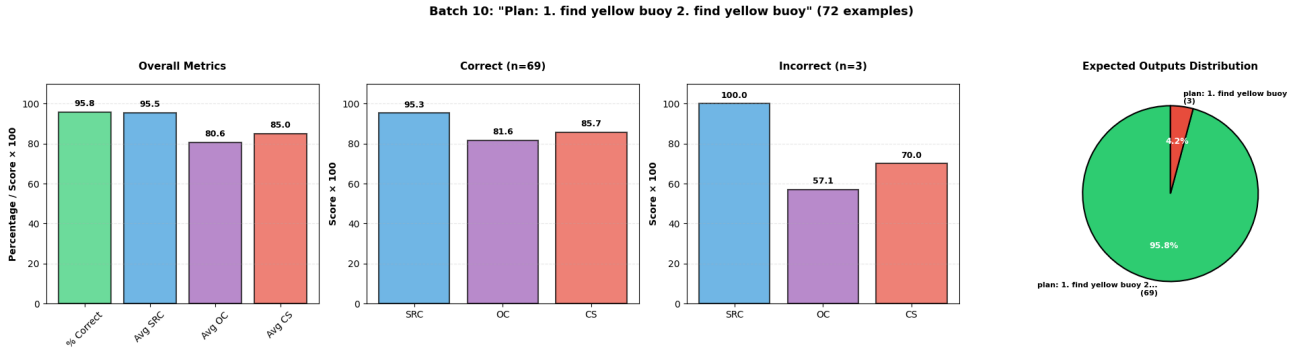


Figure 21: PoC batch 10: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

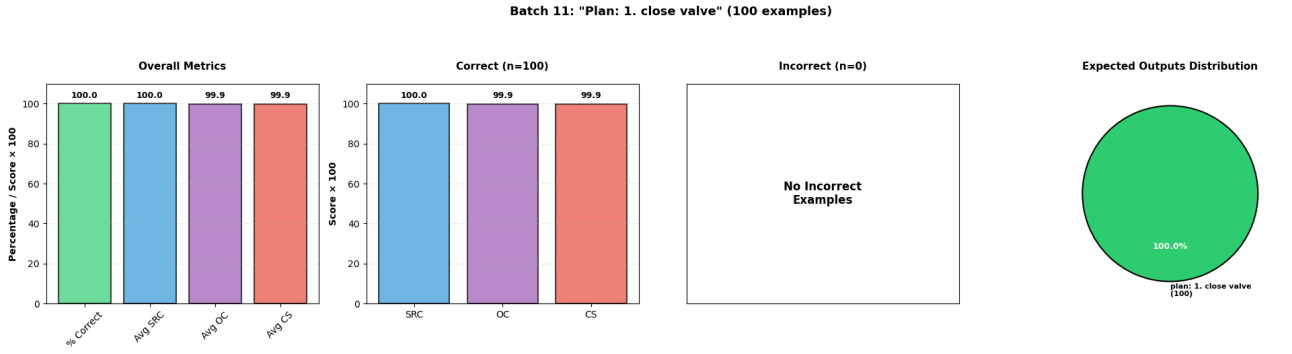


Figure 22: PoC batch 11: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

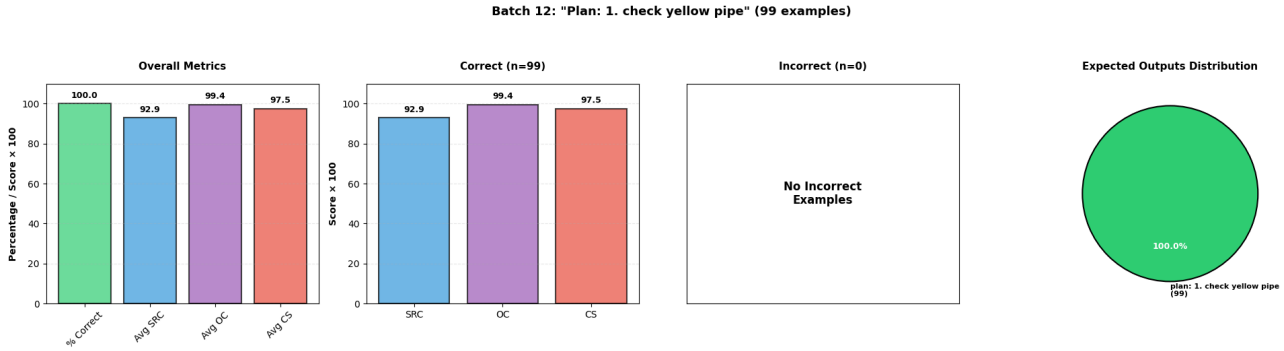


Figure 23: PoC batch 12: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

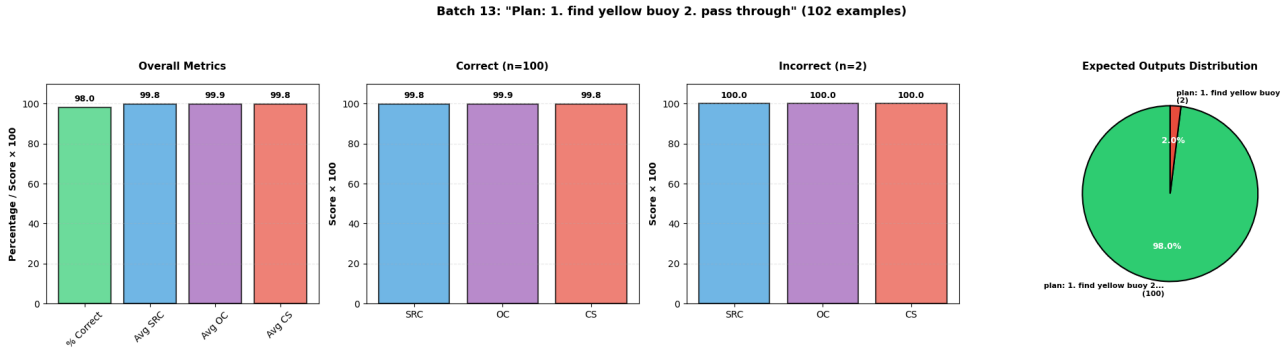


Figure 24: PoC batch 13: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

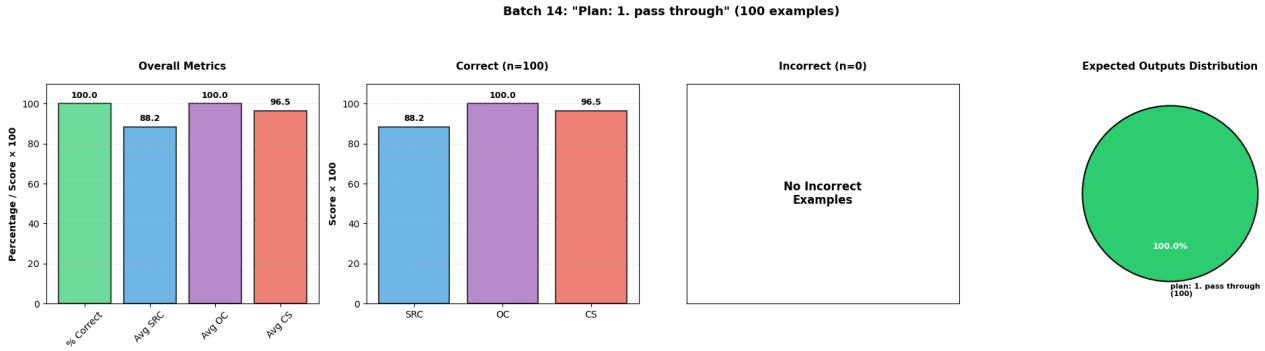


Figure 25: PoC batch 14: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

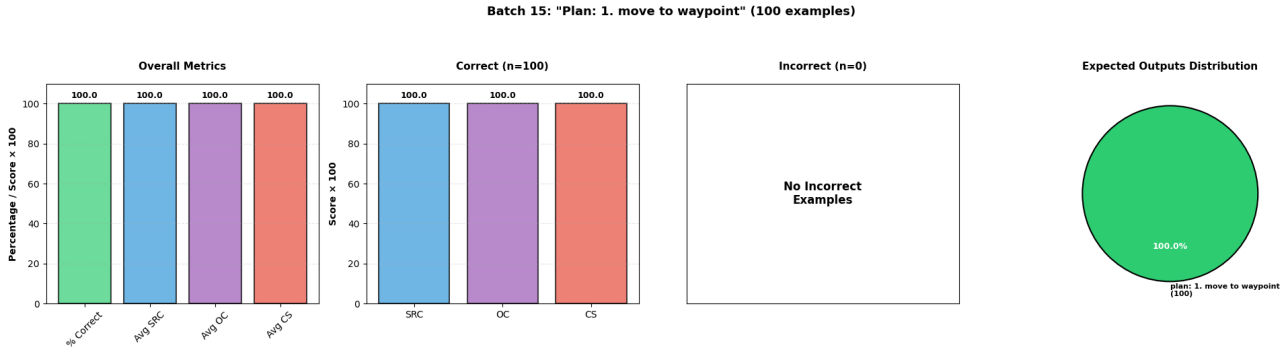


Figure 26: PoC batch 15: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

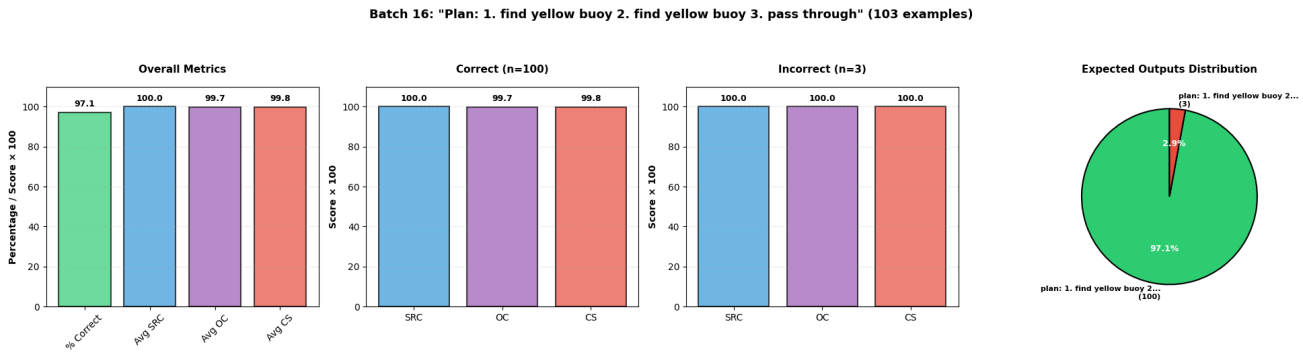


Figure 27: PoC batch 15: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

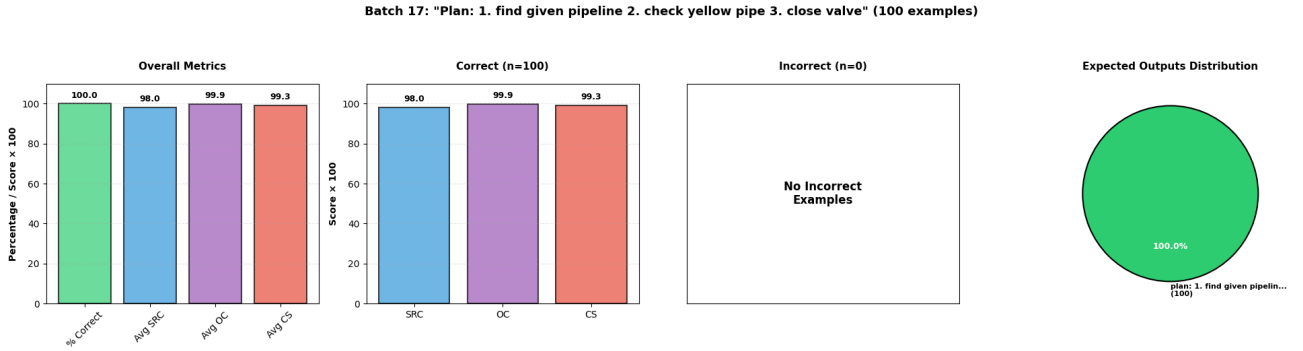


Figure 28: PoC batch 17: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

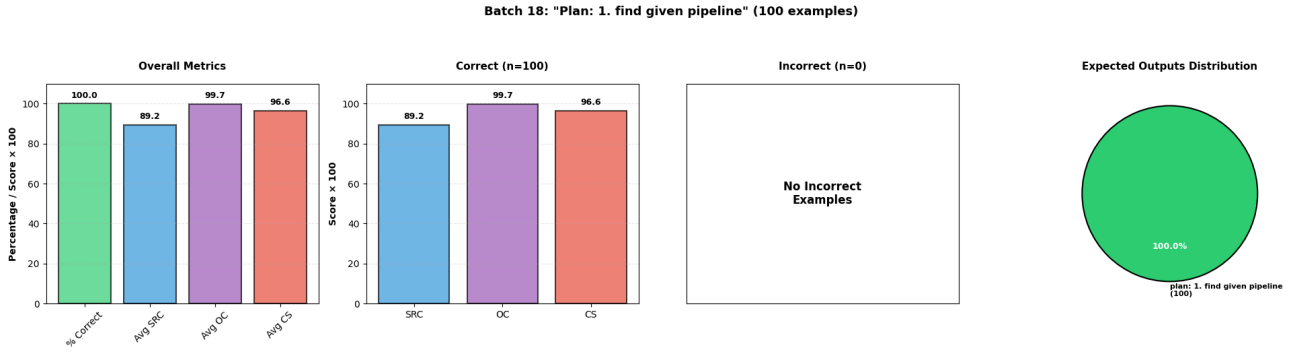


Figure 29: PoC batch 18: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

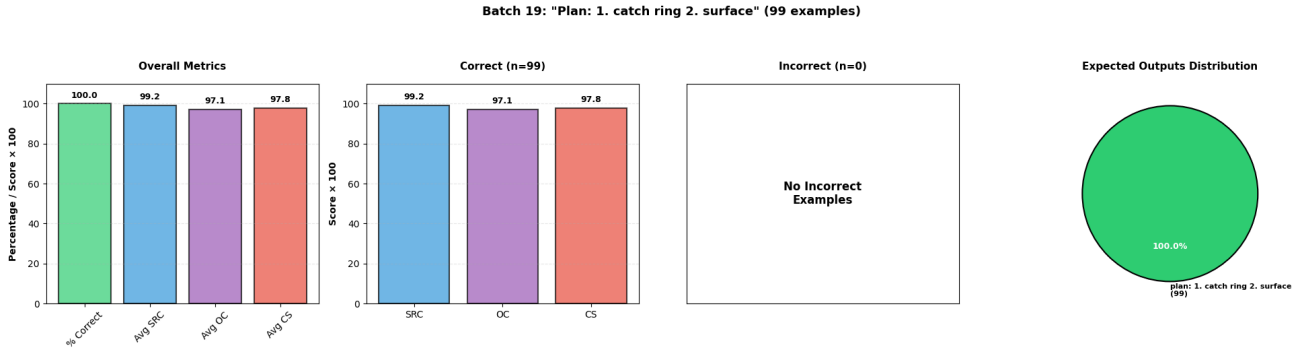


Figure 30: PoC batch 19: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

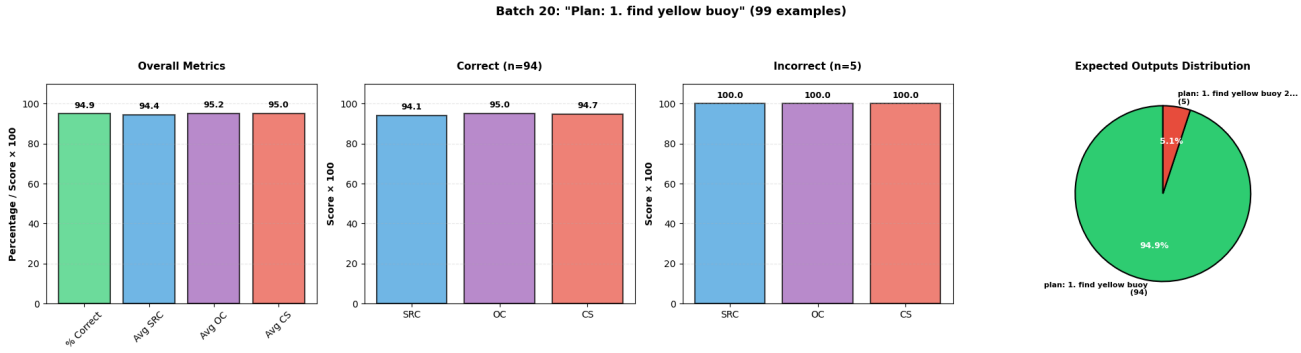


Figure 31: PoC batch 20: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

Figure 32: PoC batch 21: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

Instead, these are the results on the test dataset, that verify that the model can generalize:

==========

DETAILED METRICS REPORT

==========

LOSS METRICS

Total examples evaluated: 210
Mean loss: 0.028626
Median loss: 0.002296
Std dev: 0.064768
Min loss: 0.000112
Max loss: 0.452142

TOKEN ACCURACY METRICS

Total examples evaluated: 210
Mean token accuracy: 0.9683
Median token accuracy: 1.0000
Std dev: 0.1219
Min token accuracy: 0.2308
Max token accuracy: 1.0000

SEQUENCE ACCURACY METRICS

Total examples evaluated: 210
Mean sequence accuracy: 0.9286
Median sequence accuracy: 1.0000
Perfect matches (seq_acc = 1.0): 195/210
Partial matches (seq_acc $>$ 0.5): 195/210

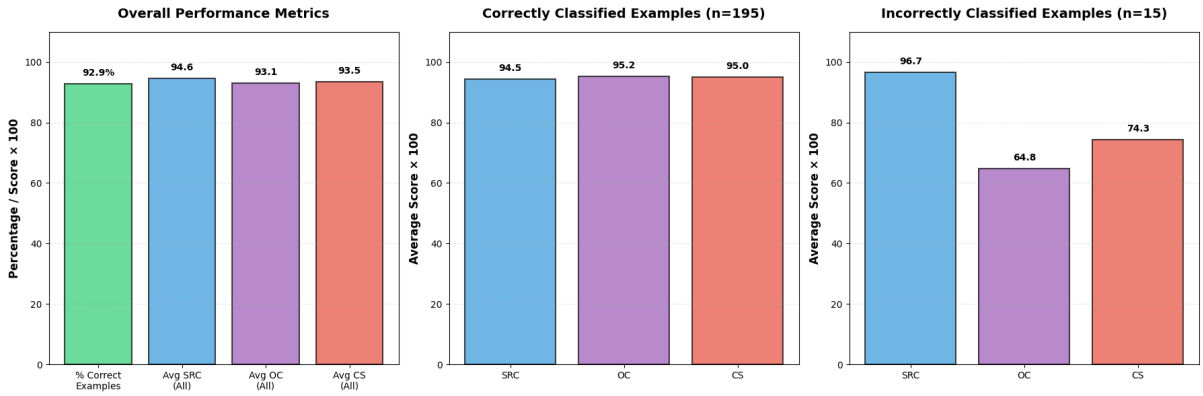


Figure 33: PoC system test with testing dataset: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

Based on these results, we can make the following observations:

- Sequence accuracy is almost the same as in the test on the training dataset, which means that the model is not particularly disturbed by the artificial noise introduced in the test dataset compared to the training dataset. Furthermore, the test dataset, which is 10 times smaller, is equally indicative.
- Despite careful training, most misclassified prompts have a high confidence level, but still lower in average with respect to correctly classified examples.
- The batches that the model finds most difficult are those related to the 'survey' and 'map area' operations, as they are the ones that present the most ambiguity, both semantically between themselves and with the concept of potential buoys introduced as artificial ambiguity (fig. 15, 16, 18).

- Usually errors lead to results that remain within the "conceptual field" of the expected output, so they are not hallucinations. For example in batch 7 (fig. 18) most errors are due to choosing to survey and map instead of just map, which is still wrong but is in this case catious since the objects need to be found before they can be parsed, similar case is batch 5 (fig. 16).
- It can be seen that for incorrectly classified examples, the SRC score is on average even higher than for correctly classified examples, which proves the unreliability assumed previously.
- The OC score is usually more adherent with the percentage of correctly classified example in general and also per batch, which makes it more reliable.
- Given the results, it would be good to match the confidence score with the OC score, but this has not been done mainly to show the difference in performance between the two scores.
- The solution to make the system more effective could be to use only the OC score as a confidence coefficient and increase the uncertainty threshold.

Small version
FLAN-T5-small was fine-tuned using the same methods and the same dataset in order to compare performance and justify the choice of the larger model. The results are shown below.

- Number of epochs = 10
- Batch size = 8
- Gradient accumulation steps = 2
- Initial learning rate = $6 \cdot 10^{-4}$
- Warm-up steps = 50
- Early stopping patience = 3
- Early stopping threshold = 0.05

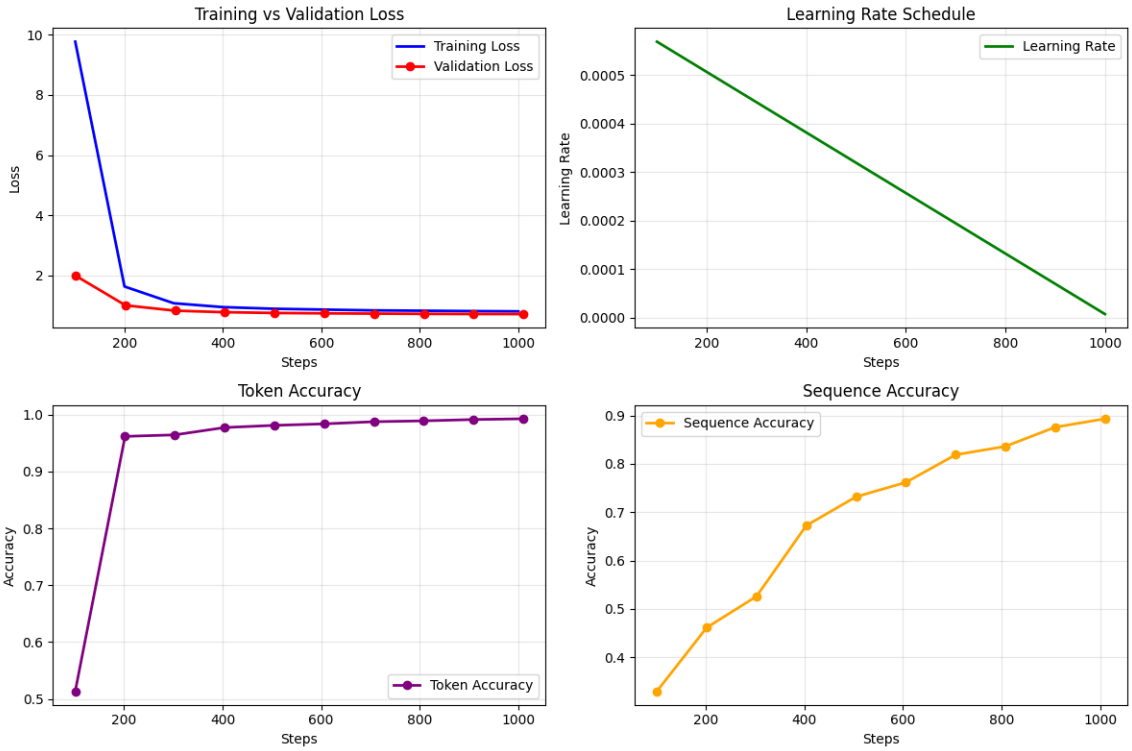


Figure 34: Metrics trend during training of the small version of the model.

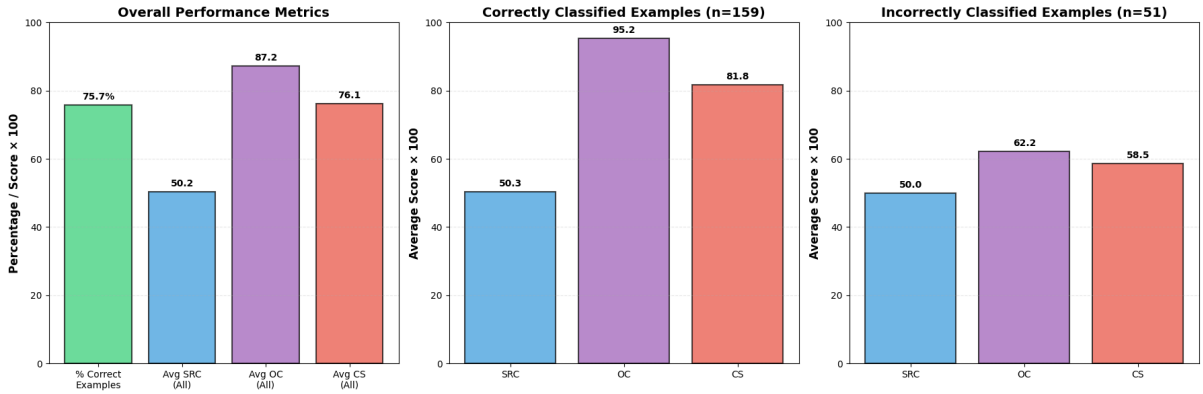


Figure 35: PoC system test with testing dataset on small version of the model: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

We can see how the percentage of correctly classified examples drops substantially, along with the SRC score, while the OC score remains high. This causes a disconnect between the confidence score and the percentage of correctly classified examples for this model. However, we can still see that the incorrectly classified examples have a lower average score.

Keyword filter off

A test was also carried out, again using the base version of the model, but without using the command-based keyword filter, so that the model has all the information available on the

progress of the mission in the prompt each time, even information that is not relevant to the request.
The results are shown below:

- Number of epochs = 10
- Batch size = 8
- Gradient accumulation steps = 2
- Initial learning rate = $4.5 \cdot 10^{-4}$
- Warm-up steps = 50
- Early stopping patience = 3
- Early stopping threshold = 0.05

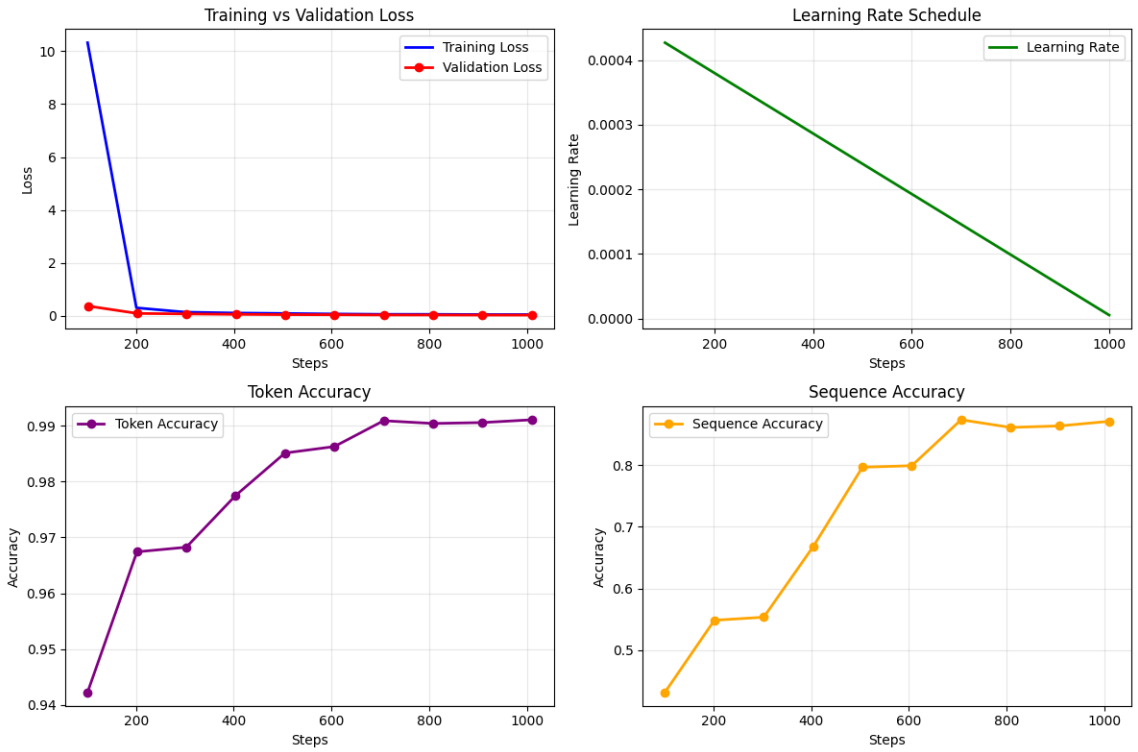


Figure 36: Metrics trend during training of the model with the dataset built considering the keywords filter off.

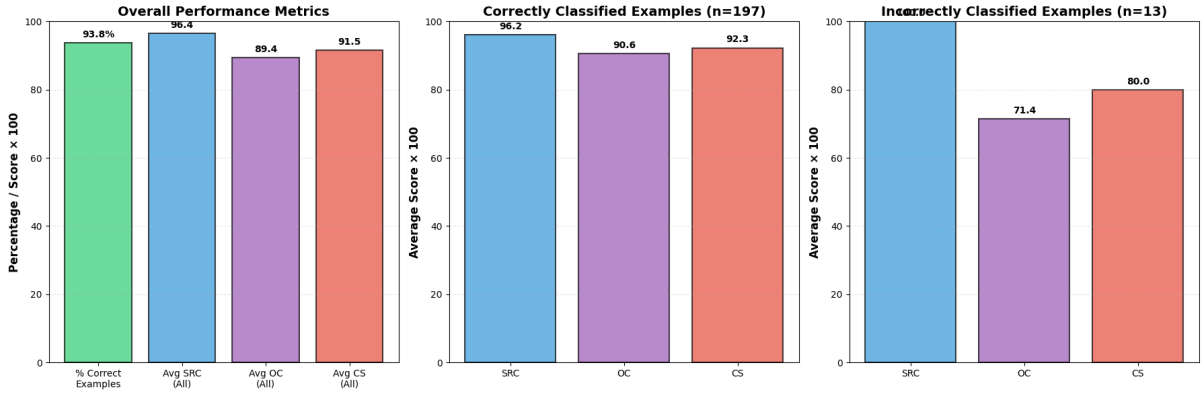


Figure 37: PoC system test with testing dataset built considering the keywords filter off.: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

We managed to achieve a similar sequence accuracy compared to the original version, although the metrics suggest a certain degree of overfitting. In general, the model seems to work well anyway, indicating that the keyword filter may not even be necessary. However, it should be noted that the confidence levels for incorrectly classified cases are higher than in the original version, which could make it more difficult to find an adequate confidence threshold.

7.2 Model training and validation for real system

The validation of the model was carried out in a completely analogous way to section 7.1, that is, by testing the model both with regards to the training metrics and the confidence scores. The following are the characteristics of the dataset used for training and testing:

```
================================================================================
DATASETS GENERATED
================================================================================

TRAINING DATASET:
   - Total examples: 1700
   - Unique output patterns: 17
   - Average examples per pattern: 100.0
   - Max examples per pattern: 100

TEST DATASET:
   - Total examples: 170
   - Unique output patterns: 17
   - Average examples per pattern: 10.0
   - Max examples per pattern: 10


TRAINING: Examples per output pattern:
   - NE quadrant survey: 100
   - NW quadrant survey: 100
```

```
   - SE quadrant survey: 100
   - SW quadrant survey: 100
   - central survey: 100
   - cross gate: 100
   - go to NE goal: 100
   - go to NW goal: 100
   - go to SE goal: 100
   - go to SW goal: 100
   - go to received goal: 100
   - make move A: 100
   - make move B: 100
   - map buoy area A: 100
   - map buoy area B: 100
   - skip: 100
   - stop_mission: 100

TEST: Examples per output pattern:
   - NE quadrant survey: 10
   - NW quadrant survey: 10
   - SE quadrant survey: 10
   - SW quadrant survey: 10
   - central survey: 10
   - cross gate: 10
   - go to NE goal: 10
   - go to NW goal: 10
   - go to SE goal: 10
   - go to SW goal: 10
   - go to received goal: 10
   - make move A: 10
   - make move B: 10
   - map buoy area A: 10
   - map buoy area B: 10
   - skip: 10
   - stop_mission: 10

================================================================================
```

Here follow the training parameters used, found empirically through trial and error:

- Number of epochs = 7
- Batch size = 8
- Gradient accumulation steps = 2
- Initial learning rate = $3 \cdot 10^{-4}$
- Warm-up steps = 30
- Early stopping patience = 3

- Early stopping threshold = 0.05

Which led to this results:

Epoch (Step)	Training Loss	Validation Loss	Token Accuracy	Sequence Accuracy
1 (85)	14.6265	2.1338	0.9240	0.1294
2 (170)	0.8542	0.0832	0.9807	0.6735
3 (255)	0.1440	0.0309	0.9967	0.9412
4 (340)	0.0871	0.0235	0.9972	0.9500
5 (425)	0.0620	0.0183	0.9984	0.9706
6 (510)	0.0511	0.0161	0.9984	0.9706
7 (595)	0.0473	0.0161	0.9985	0.9735

Table 2: Training and validation metrics per epoch
(1360 training examples, 340 validation examples).

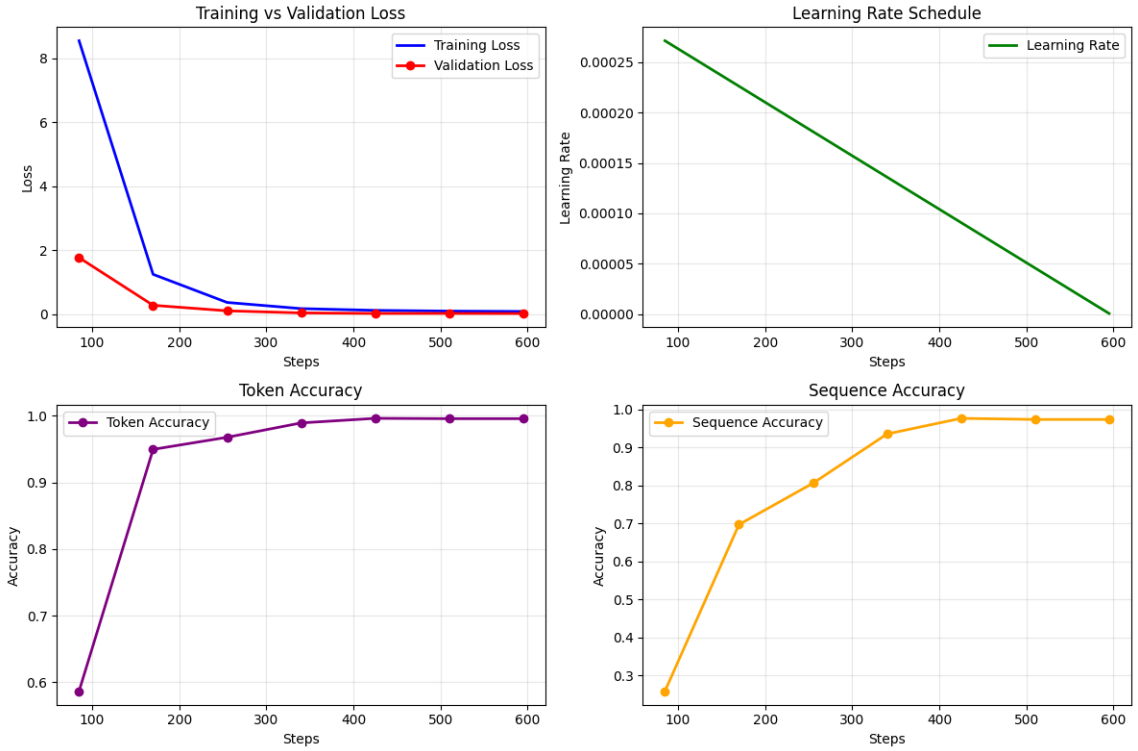


Figure 38: Metrics trend during training of the model for the real system application.

It can be noted that we managed to achieve a fairly high sequence accuracy, which remained constant over the last epochs, but at the same time, the training loss always decreased with a variation between the epochs higher than the threshold which would have triggered the early stopping mechanism, thus avoiding overfitting. As mentioned for section 7.1, it is unusual for the validation loss to be lower than the training loss. This is mainly due to the similarity of the examples in the original batch that was split to form the training and validation dataset, and consequently to the smaller number of validation examples.

Here is a summary of the test performed on the model with the dataset from which the training and validation datasets were split, to provide a comparison point for the results on the testing dataset:

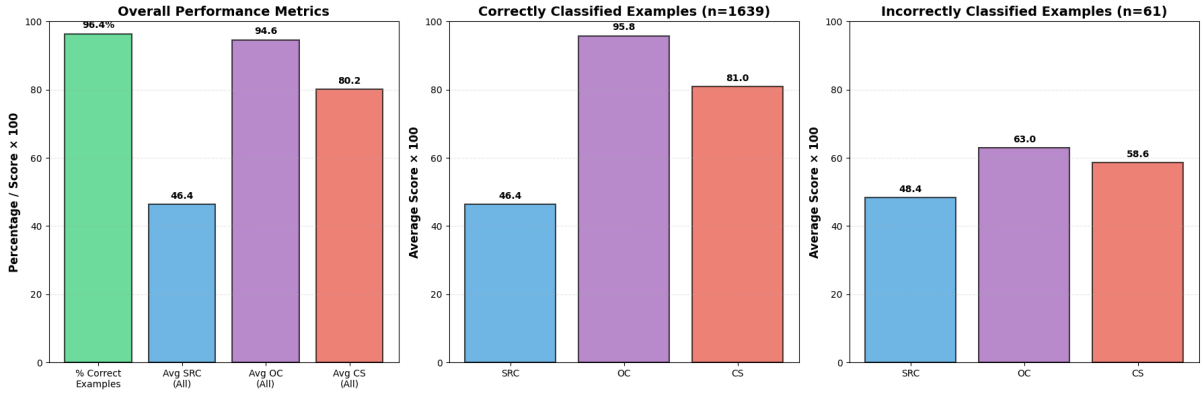


Figure 39: Real system test with initial dataset: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

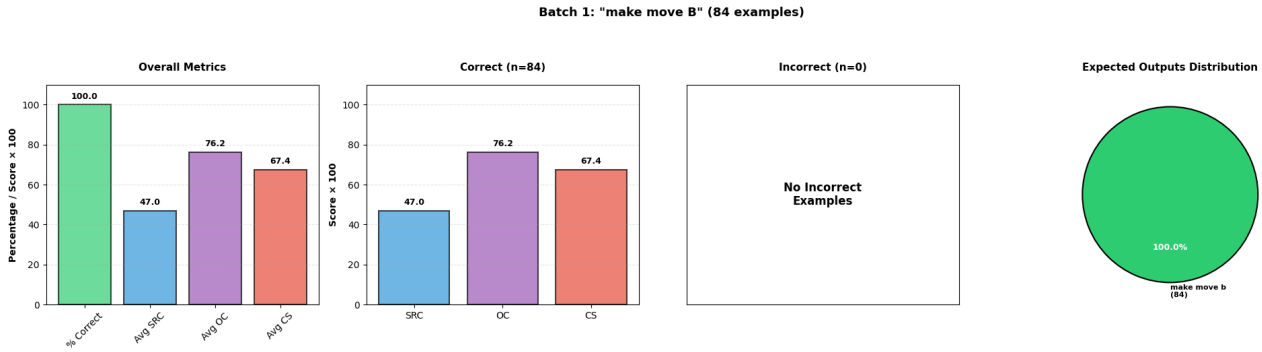


Figure 40: R batch 1: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

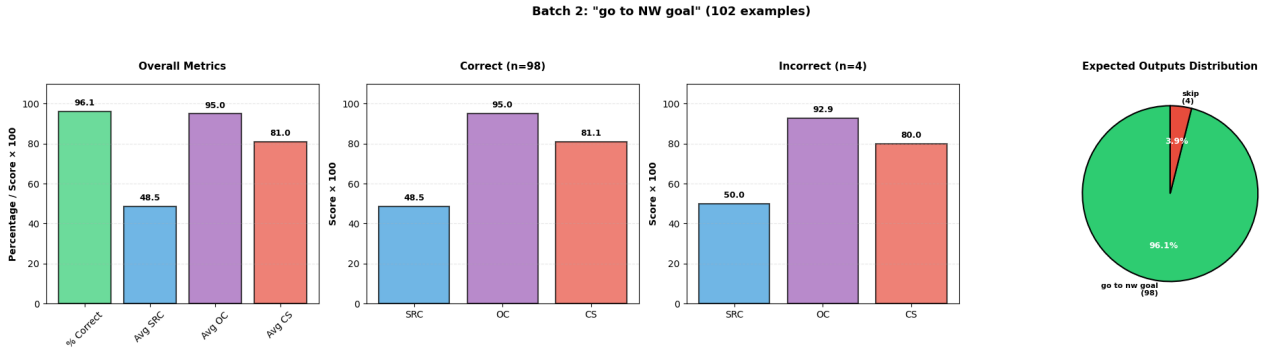


Figure 41: R batch 2: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

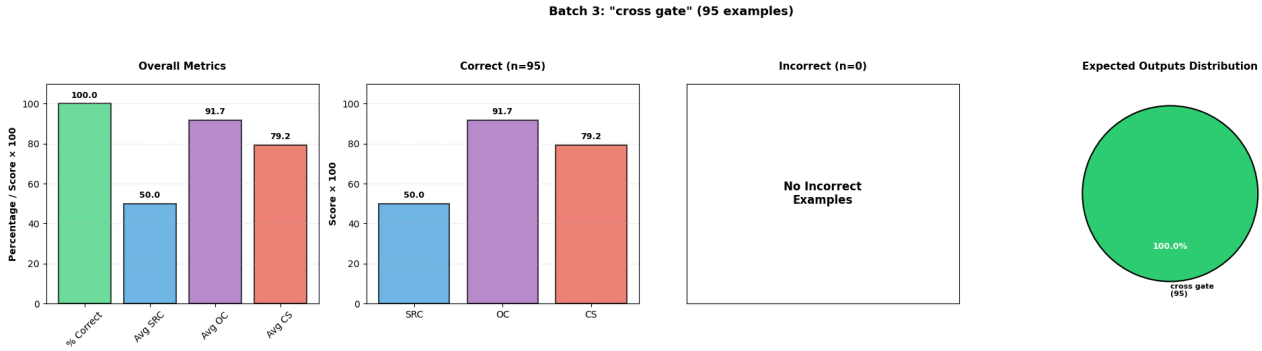


Figure 42: R batch 3: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

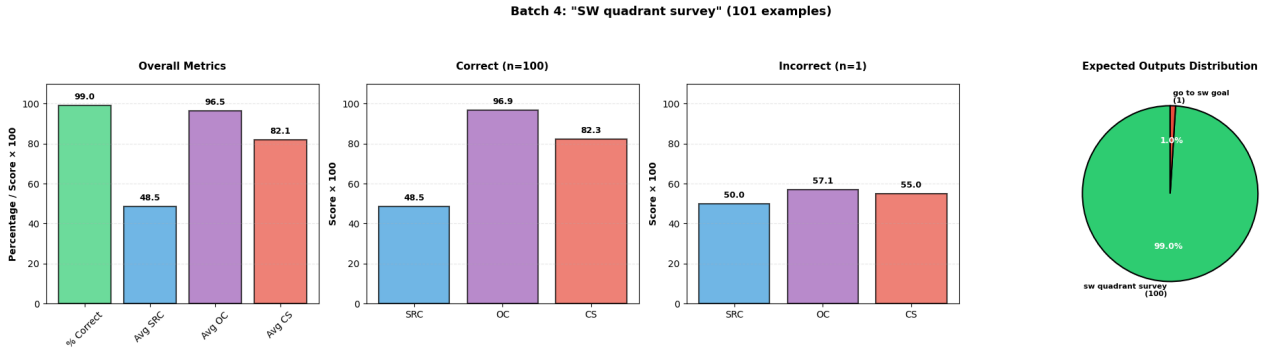


Figure 43: R batch 4: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

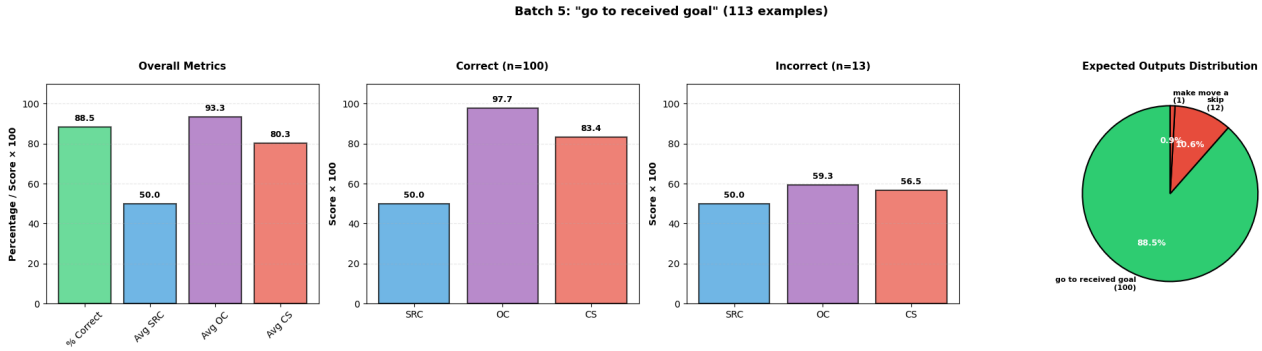


Figure 44: R batch 5: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

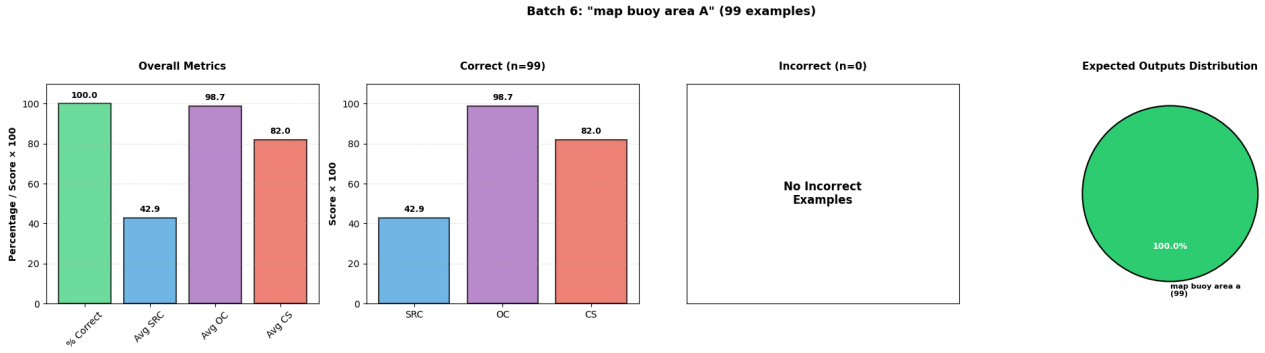


Figure 45: R batch 6: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

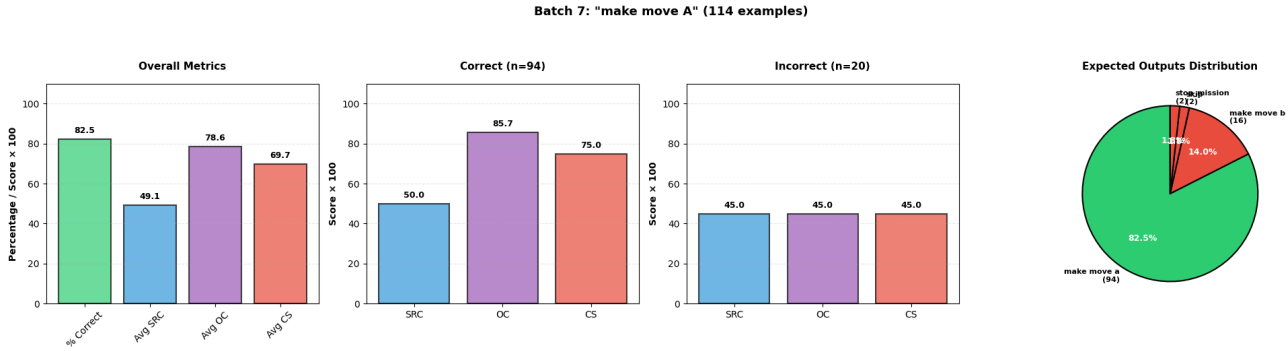


Figure 46: R batch 7: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

Figure 47: R batch 8: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

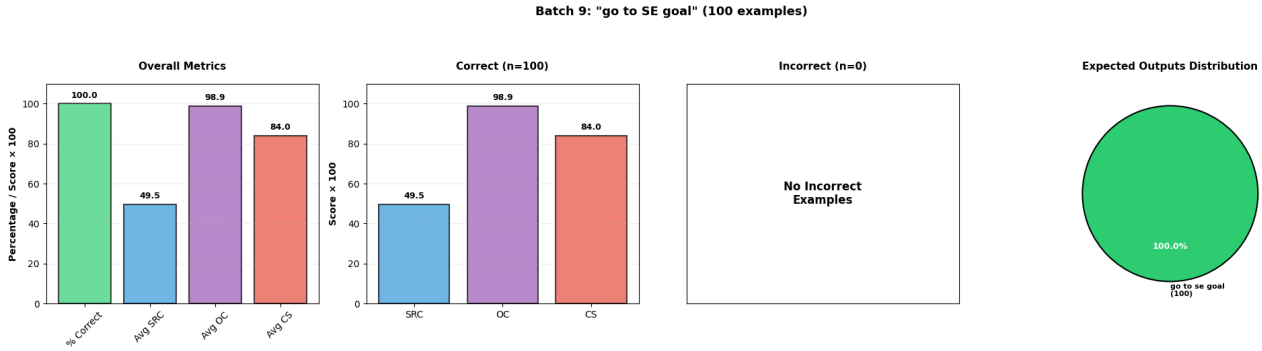


Figure 48: R batch 9: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

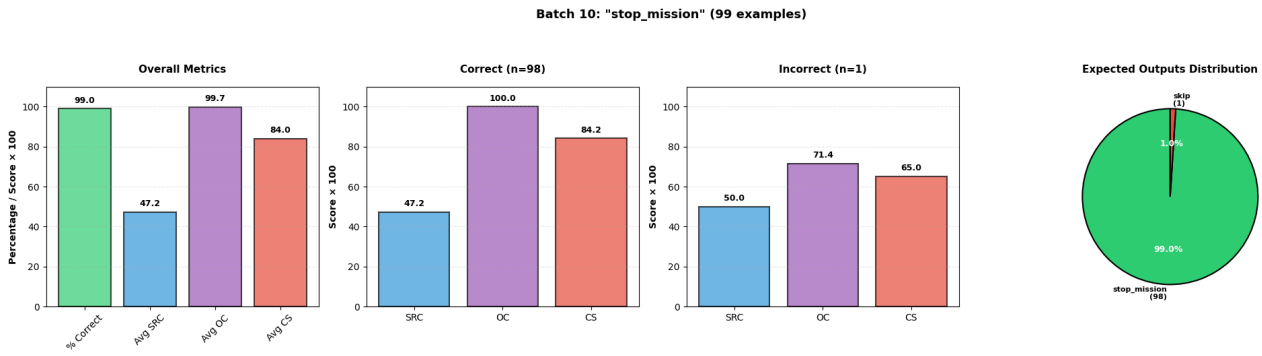


Figure 49: R batch 10: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

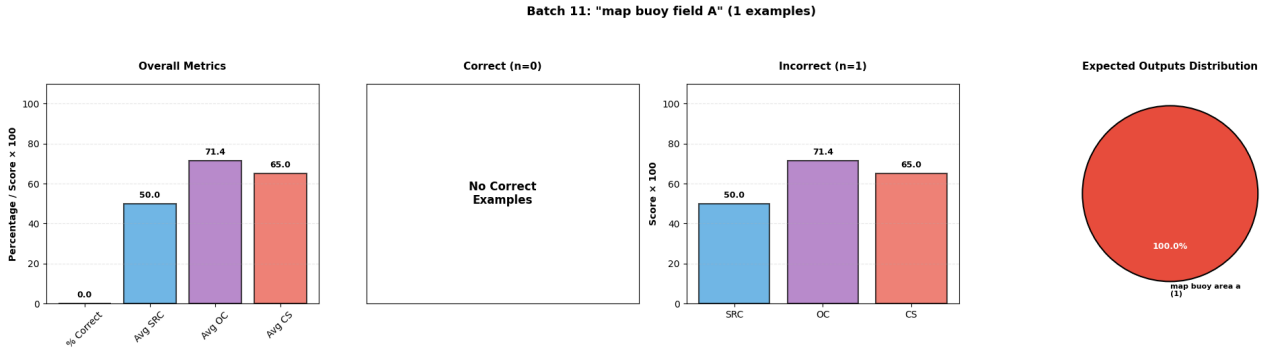


Figure 50: R batch 11: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

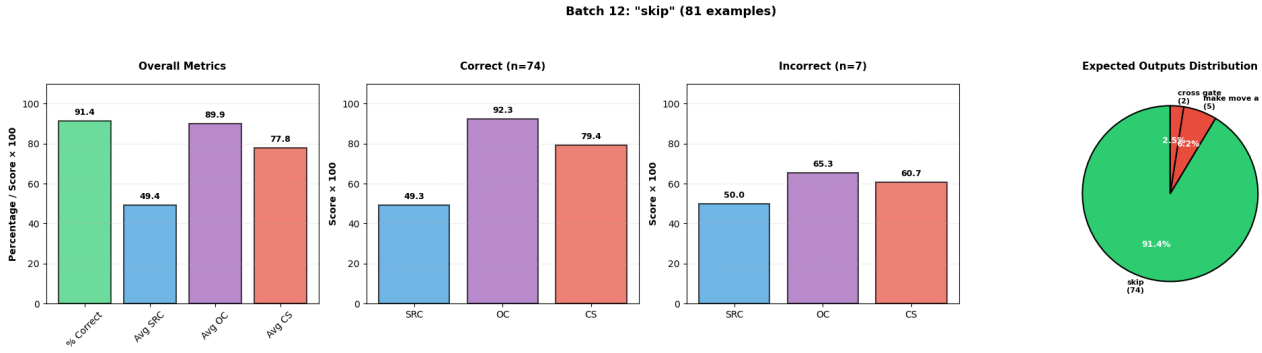


Figure 51: R batch 12: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

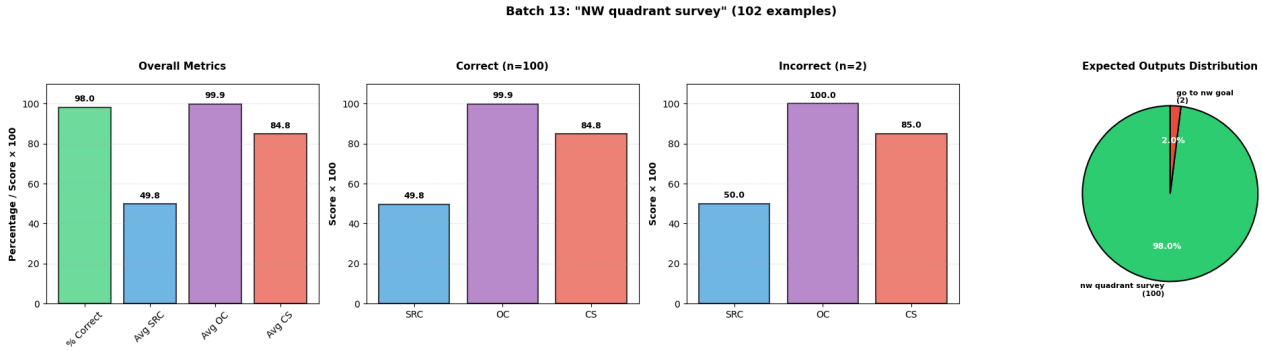


Figure 52: R batch 13: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

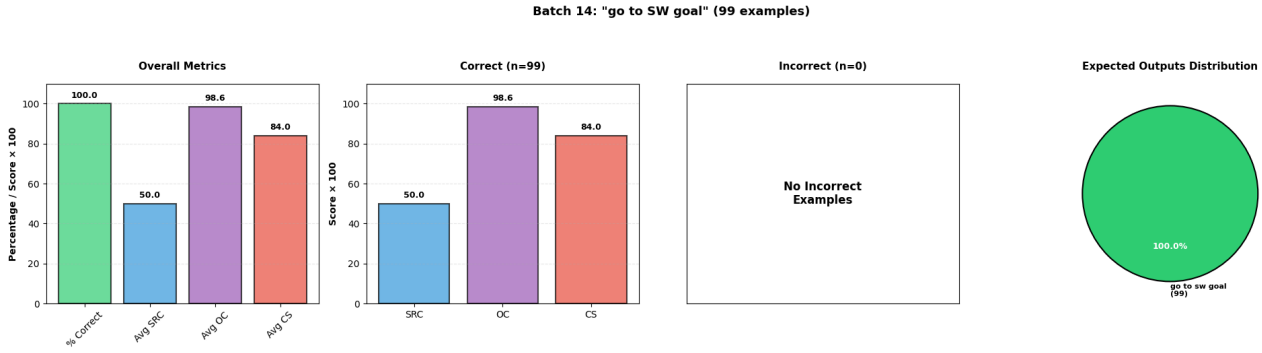


Figure 53: R batch 14: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

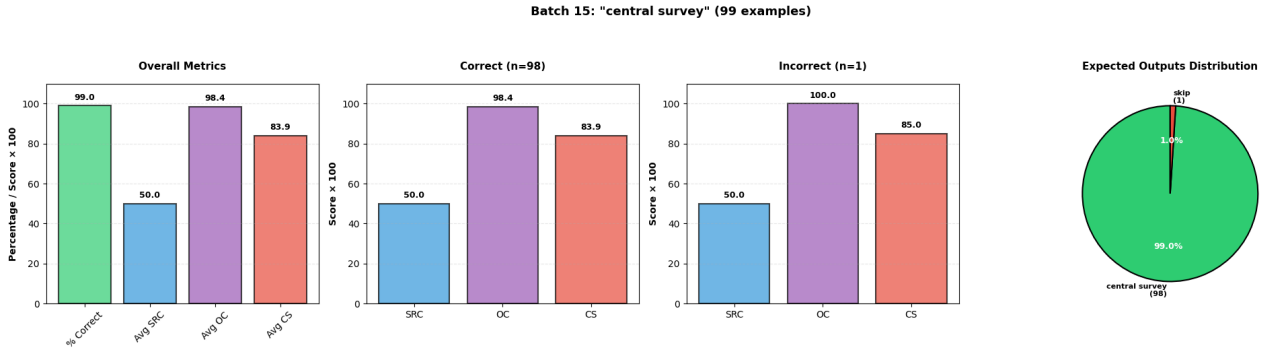


Figure 54: R batch 15: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

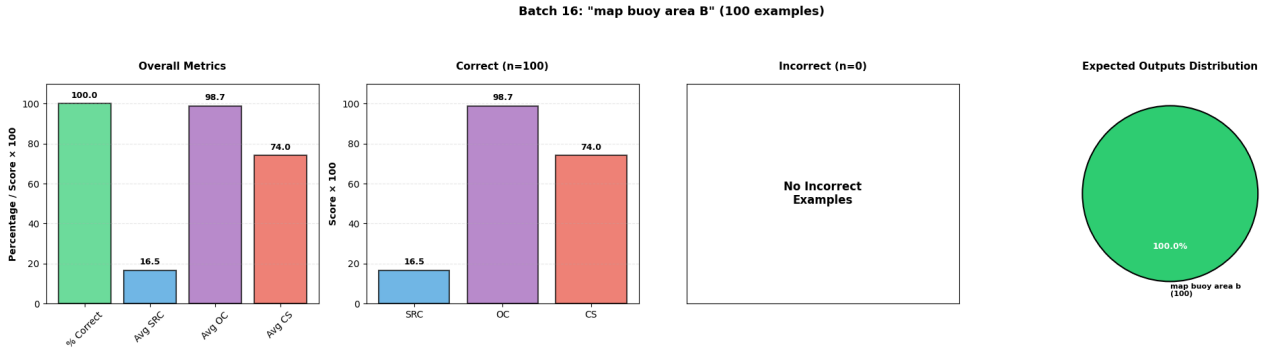


Figure 55: R batch 16: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

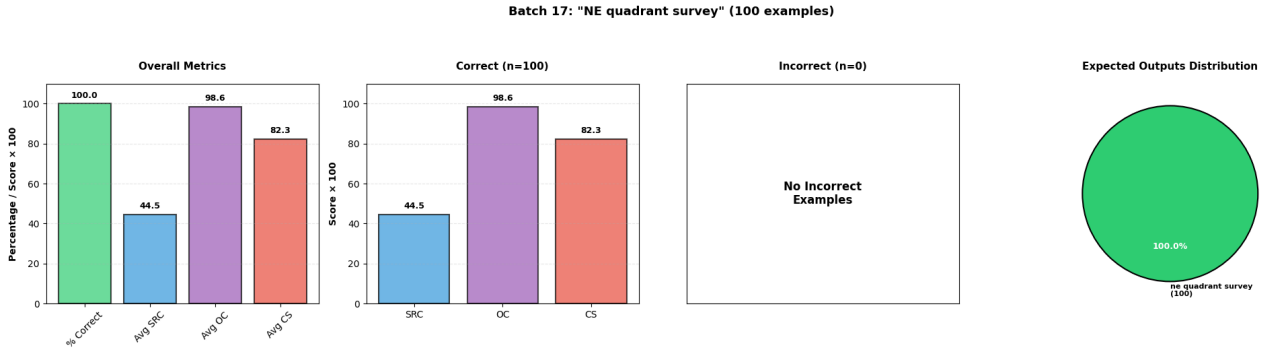


Figure 56: R batch 17: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

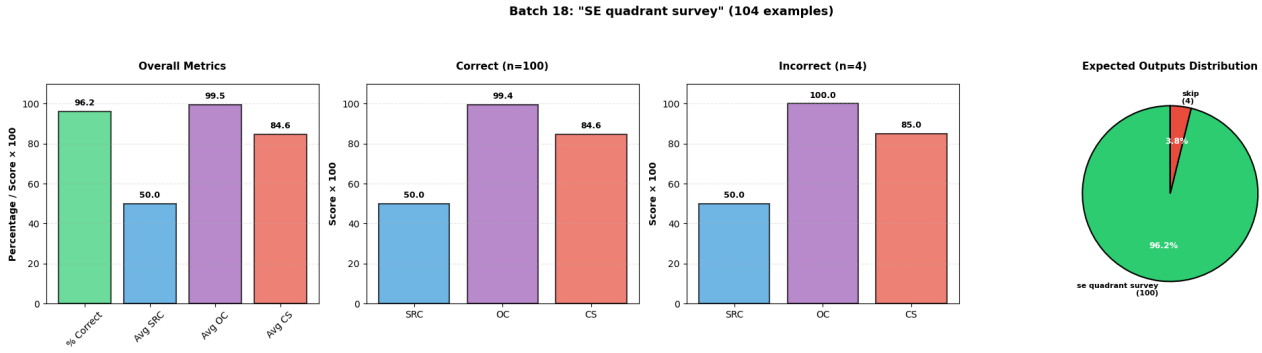


Figure 57: R batch 18: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

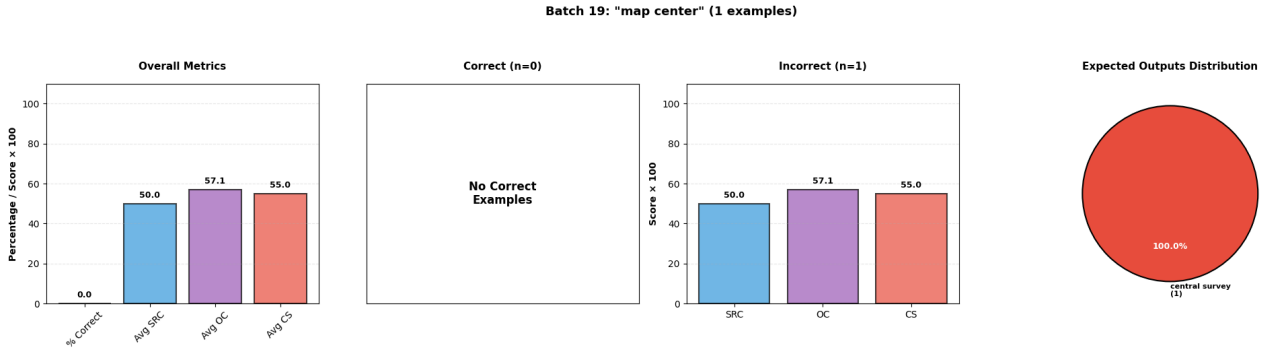


Figure 58: R batch 19: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

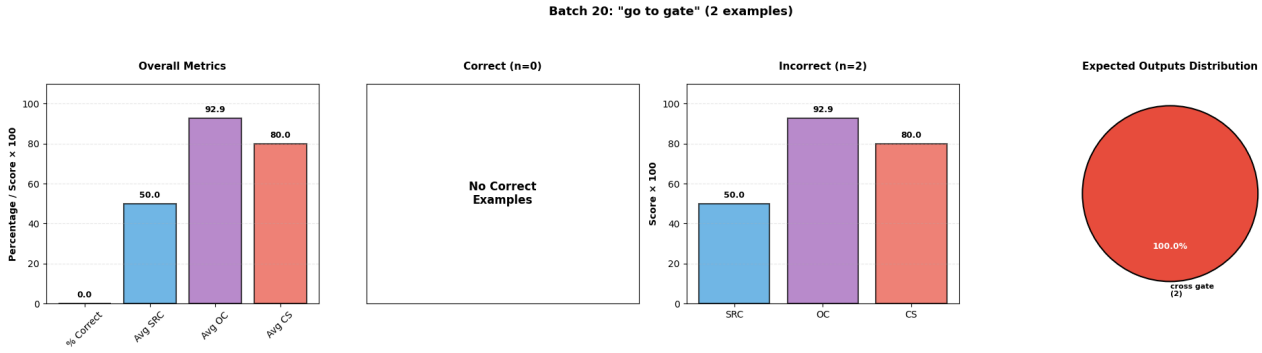


Figure 59: R batch 20: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

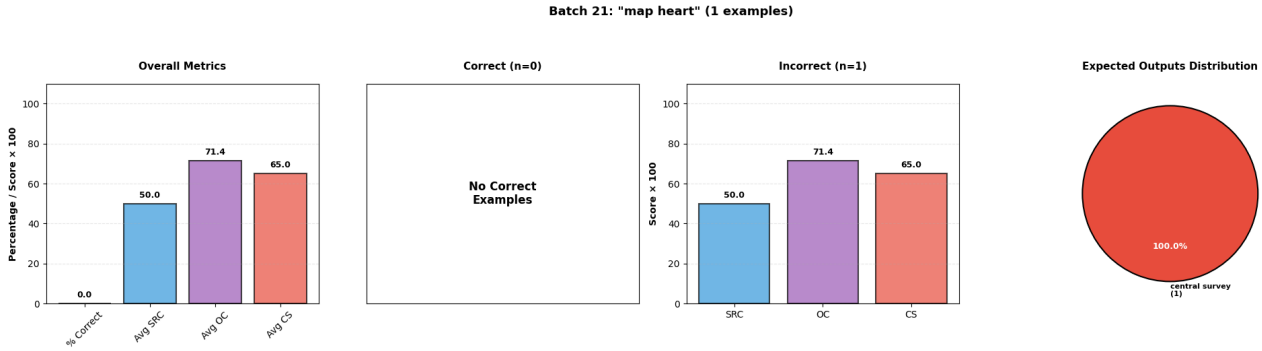


Figure 60: R batch 21: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

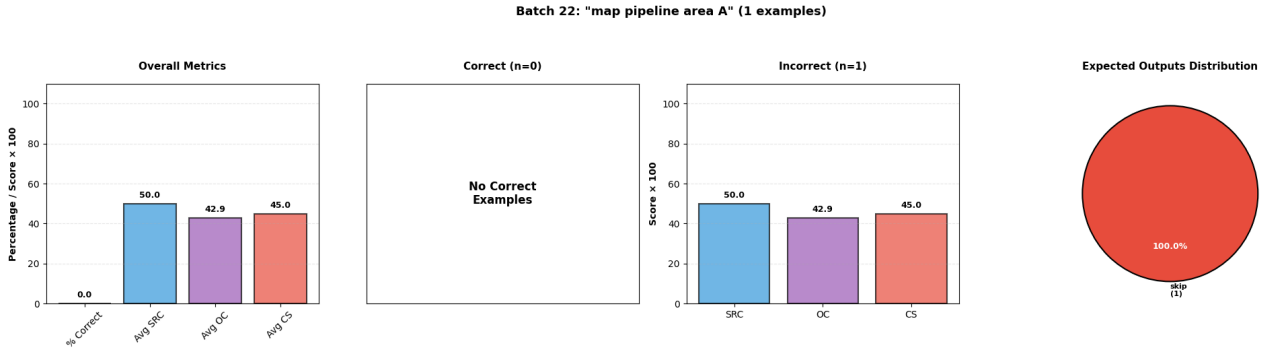


Figure 61: R batch 22: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

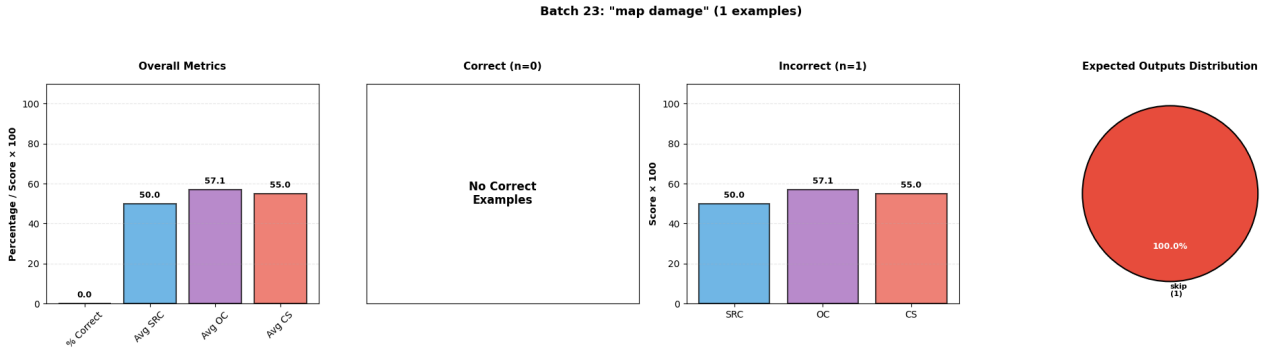


Figure 62: R batch 23: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

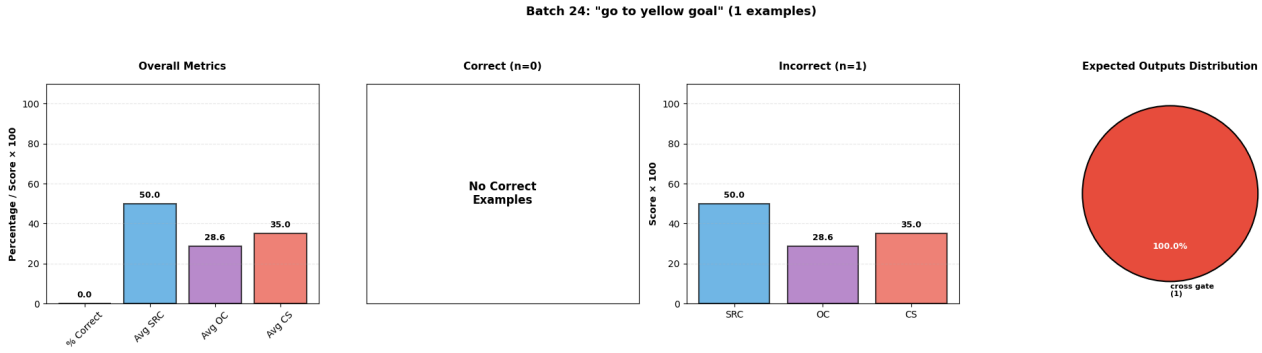


Figure 63: R batch 24: A batch represents the set of examples classified by the model with the output shown above. In the first graph, we see the percentage of examples correctly classified in this batch (matching the target label of the example) compared to the average confidence scores. We see the average confidence scores for only the correctly classified examples in the second graph and the incorrectly classified examples in the third. In the fourth graph, we see the target labels of the incorrectly classified examples in this batch.

Instead, these are the results on the test dataset, which verify that the model can generalize:

==========

DETAILED METRICS REPORT

==========

LOSS METRICS

Total examples evaluated: 170
Mean loss: 0.031603
Median loss: 0.001558
Std dev: 0.185356
Min loss: 0.000002
Max loss: 2.309174
Assessment: EXCELLENT - Very low loss indicates good model fit

---

TOKEN ACCURACY METRICS

Total examples evaluated: 170
Mean token accuracy: 0.9804
Median token accuracy: 1.0000
Std dev: 0.1339
Min token accuracy: 0.0000
Max token accuracy: 1.0000
Assessment: EXCELLENT - High token-level accuracy

SEQUENCE ACCURACY METRICS

Total examples evaluated: 170
Mean sequence accuracy: 0.9765
Median sequence accuracy: 1.0000
Perfect matches (seq_acc = 1.0): 166/170

Partial matches (seq_acc $>$ 0.5): 166/170
Assessment: EXCELLENT - Most sequences are correctly generated
==========

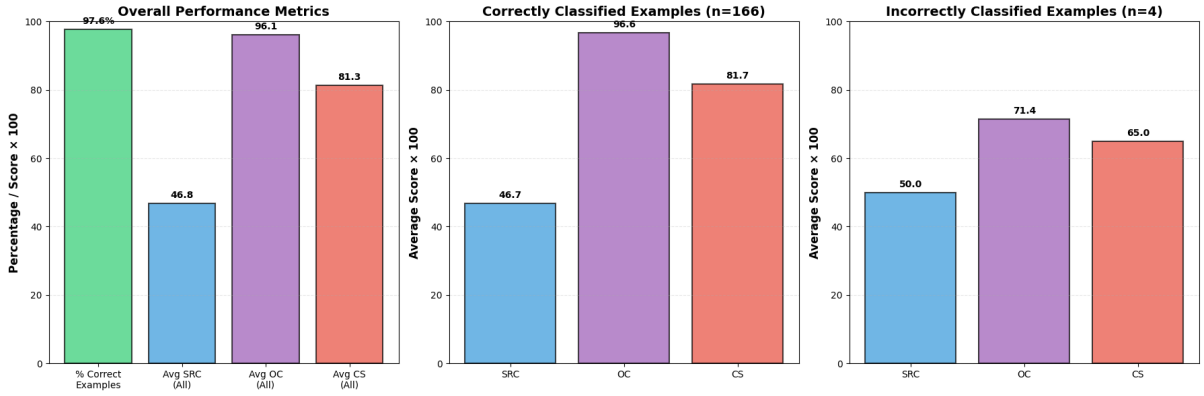


Figure 64: Real system test with testing dataset: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

What is immediately evident is that, although the average Observed Consistency score is good, the Self-reflection Certainty score is very low, in particular it tells us that, in the best case, the model is not sure if the response is correct, and on top of that the median score is zero, so the score is just bringing down the confidence score.

As mentioned previously in section 4.4, the SRC score is unreliable even when used on very large models. Once one realizes that the score is unreliable for one's purposes, one should simply abandon its use. This should certainly have been done in this case, but it wasn't only to demonstrate and justify this finding.
However, we saw in section 7.1 (which we will call PoC for brevity, while R will be used describing the real system application) how applying the same approach resulted in a significantly higher SRC score.
It was seen that the SRC score was unreliable even in section 7.1. However, it is worth pointing out the reasons why there is a certain difference between the two systems:

- Names in PoC are more self-descriptive, and make the model more confident that the sequence fits the prompt, while in R the names are tags that are linked to the BTs' mission type but do not clearly point to tasks at the linguistic level.
- Operations in PoC are more variegated and modular and their presence in the sequence is directly linked to the context, while in R most of the times for each request a single mission is the output and most of the time context is not used.
- The context in PoC is closely tied to the prompt and provides useful information in a much more descriptive and concise manner, helping the model make the right choices. In R, however, the context always has the same format for each prompt, which leads the model to give it less importance and therefore treat it more like noise.

- In PoC, a detailed analysis was done of the answers that the model had difficulty with, writing clarifying examples that confirmed correct answers and corrected incorrect answers. While in R the system has less difficulty giving the right answer, so mainly confirmatory clarifying examples were provided for a portion of the batches to build confidence, which led to an improvement from a practically zero SRC score to the current one.

Beyond this matter, some considerations on the test results are reported below:

- Sequence accuracy is high, therefore most of the example are correctly classified.
- The model makes up few mission names, but in a very small amount.
- Unfortunately, most errors have a high degree of confidence, however the overall score is lower in average with respect to the examples that are instead correctly classified, meaning that one can adjust the system confidence threshold accordingly.
- Some of the errors in many batches are due to the model responding with "skip", which is a precautionary measure because in this way the model expresses its awareness that it does not know how to perform a task.
- The OC score is fairly consistently aligned with the percentage of correctly classified examples in that batch. However, in cases where there are errors, the score remains higher. Therefore, we could talk about a correlation on average, but not for each specific case.
- The model has slightly less confidence in expressing the "make move" mission's plan B, but it did not make any mistakes.

This tells us that the model is certainly effective in classifying, however its level of confidence in its answers could be significantly improved by optimizing the prompt with an automatic construction that is more effective in providing targeted information based on the request, so as to cause less confusion for the model. Furthermore, it would be good to provide the information in the most conversational way possible.

Small version
An additional test was performed to show the difference in performance with the lighter version of the model, FLAN-T5-small. The only difference introduced was the complete removal of the "memory" field from the training and testing examples. The results are shown below.

- Number of epochs = 7
- Batch size = 8
- Gradient accumulation steps = 2
- Initial learning rate = $4 \cdot 10^{-4}$
- Warm-up steps = 30
- Early stopping patience = 3
- Early stopping threshold = 0.05

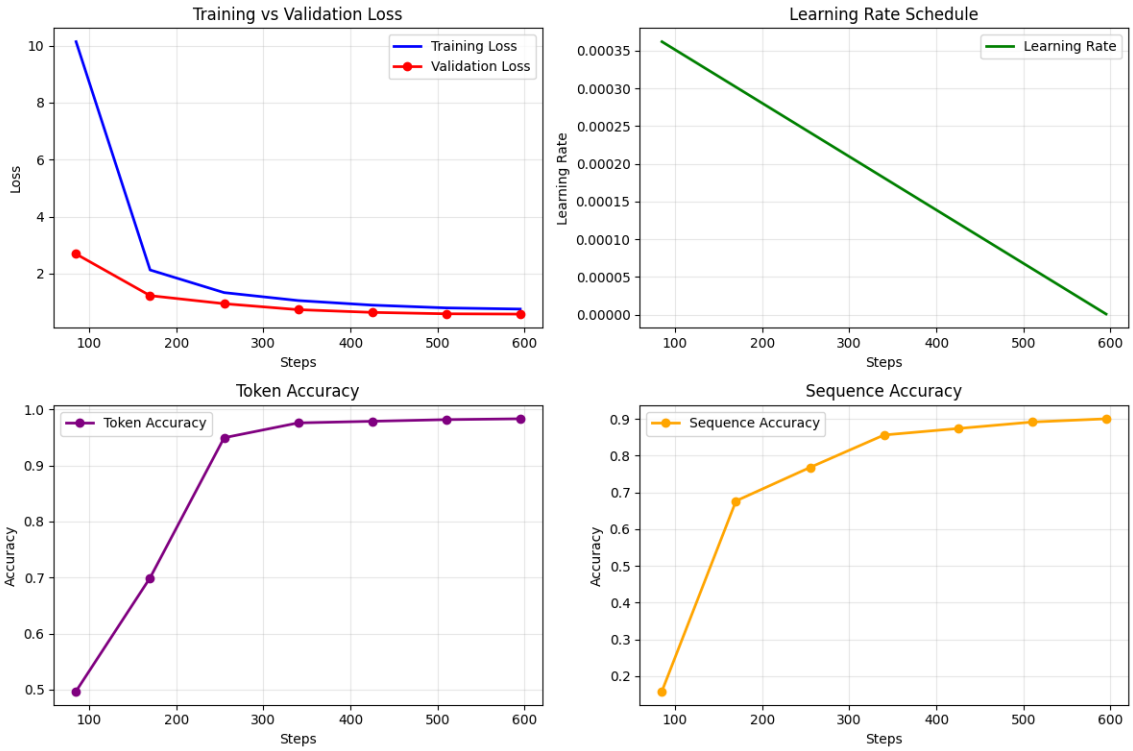


Figure 65: Metrics trend during training of small version of the model.

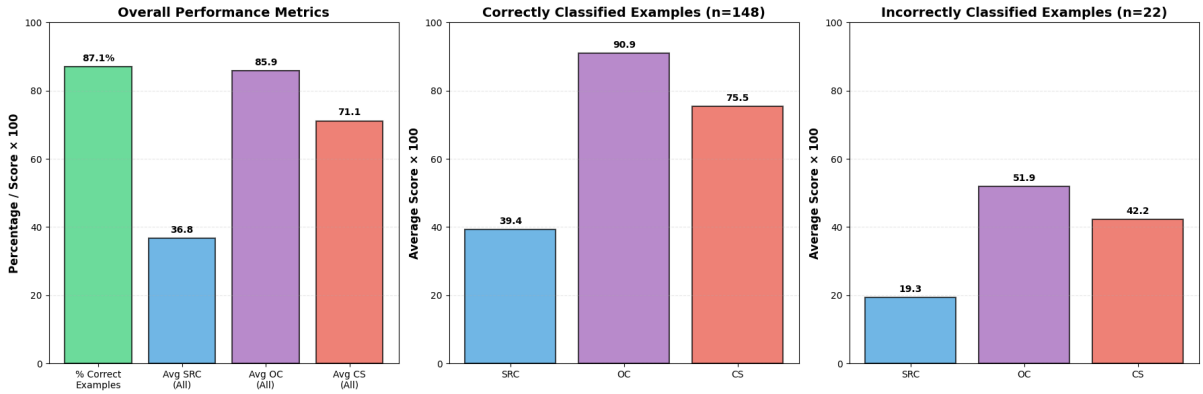


Figure 66: Real system test with testing dataset on small version of the model: The first graph shows the comparison between the percentage of correctly classified examples and the average confidence scores described in section 4.4. The other two describe how these scores change on average for correctly and incorrectly classified examples.

We can see that the small version model is generally less accurate, but confidence management is in line with the basic version. In particular, we can see that the OC score is on average similar to the percentage of correctly classified examples, as is the case for the basic version. Furthermore, as expected, the SRC score is even lower.

7.3 Experiments in simulation

This session will describe the experiments conducted using the rules outlined in section 6.6.

7.3.1 Performance experiments

First set: 4 sessions were executed using training commands.

SUCCESS RATE:

- SLM manager: 100%
- Rule-based: 100%

Mission Type	Mean (%)	Std. Dev. (%)	Min (%)	Max (%)
Overall	**83.71**	**8.68**	**67.10**	**90.00**
go to received goal	90.00	0.00	90.0	90.0
cross gate	90.00	0.00	90.0	90.0
map buoy area A	81.45	5.70	78.6	90.0
map buoy area B	87.15	5.70	78.6	90.0
make move A	69.98	5.75	67.1	78.6

Table 3: Confidence Score Statistics by Mission Type (4 Sessions)

First and second set: other 4 sessions were executed with testing commands:

SUCCESS RATE:

- SLM manager: 100%
- Rule-based: 100%

Mission Type	Mean (%)	Std. Dev. (%)	Min (%)	Max (%)
Overall	**82.96**	**10.03**	**55.70**	**90.00**
cross gate	90.00	0.00	90.00	90.00
go to received goal	87.14	8.10	67.10	90.00
map buoy area B	84.29	8.65	67.10	90.00
map buoy area A	78.57	12.23	55.70	90.00
make move A	73.66	9.02	67.10	90.00

Table 4: Confidence Score Statistics by Mission Type (8 Sessions)

All three sets: other 4 sessions were executed varying missions' order.

SUCCESS RATE:

- SLM manager: 93.8%
- Rule-based: 96.9%

Mission Type	Mean (%)	Std. Dev. (%)	Min (%)	Max (%)
Overall	**82.67**	**10.27**	**55.70**	**90.00**
go to received goal	84.28	11.44	55.70	90.00
cross gate	87.14	7.11	67.10	90.00
map buoy area A	81.43	11.04	55.70	90.00
map buoy area B	82.38	10.16	67.10	90.00
make move A	76.65	10.74	67.10	90.00
make move B	86.20	6.58	78.60	90.00
stop mission	90.00	0.00	90.00	90.00

Table 5: Confidence Score Statistics by Mission Type (12 Sessions, 65 Samples)

BT managers failures:

- During a gate crossing mission, the search for the second buoy failed. This is because the survey to find the second buoy depends on the point where the vehicle finds the first one, and this point in turn depends on the direction from which the vehicle arrived to perform the first survey.
- In one session, the yellow buoys were found before the gate crossing mission. When the mission was started, there was a problem with saving the buoys, specifically, the vehicle passed between one of the yellow buoys and another buoy.

SLM manager only failures:
The model responded to command ”determine the colours of each buoy within the area and execute actions in the designated response mode” with mission ”map buoy area A” (CS = 67.1%) instead of ”make move A”, which had already been completed, so a request for clarification regarding the repetition of the mission was triggered, to which the response was ”skip this”, which again led the model to respond with mission ”map buoy area A” (CS = 67.1%) instead of ”skip”.

The conclusion we can draw from these experiments is that the model, as seen in the previous section, has more difficulty with the ‘map buoy area’ and ‘make move’ missions, probably due to the meaning field of the two missions and the similarity of the terms that identify the command. In any case, the use of the SLM manager proved to be fallacious only in one particular case of ambiguous command for move performing and with a clarification command for which the model was not trained. However, we can see that, in general, the confidence score is lower (although not below the 50% threshold) than the average.

Reported below are resources usage and latency statistics:

Metric	Mean	Std. Dev.	Min	Max
Average CPU Usage (%)	2.24	1.08	1.14	5.23
Peak CPU Usage (%)	102.91	2.88	98.80	110.00
Average Memory (MB)	1464.93	14.36	1435.03	1485.65
Peak Memory (MB)	1478.40	8.45	1464.64	1493.16

Table 6: System Resource Usage Statistics (12 Sessions)

Metric	Mean (ms)	Std. Dev. (ms)	Min (ms)	Max (ms)
Communication Latency	139.88	273.13	11.32	1567.69

Table 7: WSL22→WSL18 `/send_mission` Latency Statistics (12 Sessions, 88 Samples)

Measurements show that, on average, the SLM manager consumes approximately 1.5 GB of RAM and 2% of the CPU of the machine used, as expected [9].
Deployment on the onboard computer on Zeno would therefore certainly be possible, but it would certainly require analysis of resource consumption by the control system and other systems, such as the neural network used for buoy recognition.

7.3.2 User accessibility experiments

Average operator experience with rule-based system:

- The operator must open the mission file and read all the missions before deciding what to do.
- All operators except one chose to send the 'go to received goal' mission first and then the 'cross gate' mission. The other operator decided to do a central survey first, but then decided to send the same two missions anyway.
- In one case, the "cross gate" mission failed, so the operator sent the mission again, which was then successful.
- After crossing the gate, it was chosen at random whether to use the "map buoy area A" or "map buoy area B" mission. Four operators chose the second one directly, which led directly to success. The operator who chose mission A saw it fail and sent mission B.
- Three operators sent all three missions to perform the move for each buoy, while the other two assumed that it was sufficient to do so for the red buoy.
- All operators conducted surveys in all areas of the map to find the pipeline, unaware that the system cannot recognise a pipeline structure. After surveying all quadrants of the map they all terminated their session.
- An average of 13 commands were sent to the system and an average of 12 missions were performed per session.

Average operator experience with SLM manager:

- Each operator simply takes the request from the RAMI task list and rephrases it to write the message on the system chat.
- For each request to reach the goal and pass the gate, the correct mission was selected with a sufficient level of confidence.
- In one case a cross gate mission failed, triggering replanning with the same mission, which led to success.
- For each request to map the buoy area, plan A was selected, which failed because the buoys could not be found, but plan B was then selected and the mission was completed without further issues.

- For four operators, the request to perform the moves for each buoy was mapped to the alias that leads back to plan A of the sequence of three move performing missions for all colours.
- In the case of one operator, plan B of the same mission was selected with a low confidence coefficient, but it was sufficient for the operator to confirm their choice to allow the model to make the same selection with sufficient confidence, and then send the mission and complete it.
- For each pipeline survey request, the model responded with “skip”, indicating to the operators that it was not a feasible task, leading them to terminate the mission.
- An average of 5 commands were sent to the system and an average of 7 missions were performed per session.

The conclusions we can draw are that the rule-based system places the entire decision-making burden on the operator, opening up the possibility of errors due to an incomplete understanding of the tasks to be performed or the capabilities of the system. The SLM manager, on the other hand, relieves the operator of the need to understand how the system works and its capabilities, reducing their role to that of relaying requests and supervising the progress of the mission. The number of commands sent and the choices that the operator has to make are also reduced. Also the lower number of error from the operator allows to reduce the session duration, since useless missions are not pursued.

8 Conclusions

The formal approach described in section 4 has been adapted to the purpose-built system described in section 5 and to the real vehicle system described in section 6, succeeding in its aim of acting as a simplifying interface between the user and the execution system, in particular by managing to map natural language requests into operations useful to the latter.

There are some important considerations that need to be made, so that the application of the approach can be done as effectively as possible:

- The T5 architecture is text-to-text, so it's a good idea to make the information included in the prompt and the names of the operations that will be output as conversational as possible, so as to facilitate understanding of the content of the request and the self-assessment that includes both the request and the generated response.
- The prompt should be constructed with as little content as possible to allow the model to make the correct choice, possibly filtering the prompt fields based on the request. Otherwise, the model may recognize some information as useless during training, leading it to ignore it even when it is actually useful for making one decision rather than another.
- When the execution system is designed to receive a limited number of operations, it's a good idea to assign each one a short, self-descriptive name and always present the model with options, along with instructions to choose between them. When these operations are a significant number and they would make the prompt too large is better to avoid adding all the pobbible operations, but it's a good idea to train the model with responses that contain all the information needed to define a unique operation.
- A system of this kind is more effective if the operations are modular, since during training the model is able to better understand the function of that operation depending on the context, thus being able to generalise better.
- It's a good idea to run a few training attempts, so that with each attempt one can see which types of examples the model struggles with the most. From this information, one can more easily add clarifying examples to the training dataset. These help increase the confidence level of correctly classified examples that the model is unsure about, and correct incorrect examples using the clarifying format, even if the confidence level of the incorrect answer is high.
- Experiments suggest that the system using the SLM manager performs similarly to the rule-based system but is a much simpler and more user-friendly platform, which also reduces execution errors and therefore session time.

It can therefore be concluded that the approach is versatile and applicable to robot control systems, effectively replacing mission management systems. For its application, the most important phases involve analyzing potential requests, efficiently constructing the prompt, and simply and scalably constructing the training dataset.

8.1 Possible future improvements

This work primarily defines an approach aimed at creating an effective and efficient mission manager. An application has been described in section 6, but as described in the same section,

this application is not intended to be ready-to-use for real-world missions. The idea is to lay the foundation for adapting the approach to each context, making the mission manager as useful as possible for any type of mission.
Possible continuations of this research could be found in the following areas:

- A survey of various language models of similar size to find the most suitable one for this purpose. Looking even further, the construction of a specialized architecture to specifically handle this type of situation.
- An effective method for managing the construction of contextual elements starting from the mission environment, followed by an appropriate system for targeted selection of the information needed to satisfy the request presented to the system.
- A method for dividing system capabilities into operations that can be assigned a name tag, allowing the model to call the operations. These operations must be unique and have a clear name tag so that the model can understand how to construct sequences of operations based on context.
- Further analyze the relationship between these systems and behavior trees, possibly using more modular trees to replace the operations mentioned above.
- An optimized system for constructing datasets, both for training and testing, that distinguishes between different types of prompts and their ratio to the total number, also based on their meaning field.
- A data collection system suitable for the platform on which the experiments are performed, which makes it possible, in post-processing, to easily generate new examples that add functionality to the model, increase the model's confidence in correct answers, or correct errors.
- User accessibility tests should be carried out on a large number of actual operators to verify the results obtained in this work. It would also be useful to have a personal assessment from each operator on the ease of use of the two systems.

References

[1] Ruochu Yang, Mengxue Hou, Junkai Wang, and Fumin Zhang. Oceanchat: Piloting autonomous underwater vehicles in natural language, 2023. arXiv:2309.16052.

[2] Ruo Chen, David Blow, Adnan Abdullah, and Md Jahidul Islam. Word2wave: Language driven mission programming for efficient subsea deployments of marine robots, 2025. arXiv:2409.18405.

[3] Ehsan Khorrambakht, Paolo Marinelli, Estelle Gerbier, Stefano Chiaverini, Andrea Caiti, Paolo Di Lillo, and Andrea Munafò. Composable and modular autonomy for maritime robotics: Bridging human-robot collaboration, 2025. URL: https://program-brest25.oceanstechnical.org.

[4] Gawon Choi and Hyemin Ahn. Can only llms do reasoning?: Potential of small language models in task planning, 2024. arXiv:2404.03891.

[5] Yuxuan Chen, Yixin Han, and Xiao Li. Fastnav: Fine-tuned adaptive small-language-models trained for multi-point robot navigation, 2024. arXiv:2411.13262.

[6] Jiuhai Chen and Jonas Mueller. Quantifying uncertainty in answers from any language model and enhancing their trustworthiness, 2024. arXiv:2308.16175.

[7] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai: An introduction, 2022. arXiv:1709.00084.

[8] Hyung Won Chung, Le Hou, Shayne Longpre, et al. Scaling instruction-finetuned language models, 2022. arXiv:2210.11416.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019. arXiv:1910.10683.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. arXiv:1706.03762.

[11] Jason Wei, Maarten Bosma, Vincent Zhao, et al. Finetuned language models are zero-shot learners, 2022. arXiv:2201.08239.

[12] Nicholas Kluge Corrêa, Sophia Falk, Shiza Fatimah, Aniket Sen, and Nythamar de Oliveira. Teenytinyllama: Open-source tiny language models trained in brazilian portuguese, 2024. arXiv:2401.16640.

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. arXiv:2106.09685.

[14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. arXiv:2305.14314.

[15] Zhizhong Liu et al. A metric for token-level calibration on large language models, 2024. arXiv:2406.11345.

[16] DigitalOcean. Automated metrics for evaluating the quality of text generation, 2024. URL: https://www.digitalocean.com/community/tutorials/automated-metrics-for-evaluating-generated-text.

[17] Garvesh Raskutti, Martin J. Wainwright, and Bin Yu. Early stopping and non-parametric regression: An optimal data-dependent regularization method, 2014. URL: https://jmlr.csail.mit.edu/papers/volume15/raskutti14a/raskutti14a.pdf.

[18] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering for large language models. *Patterns*, 2025. In press. URL: https://arxiv.org/abs/2310.14735.

[19] Dang Anh-Hoang et al. Survey and analysis of hallucinations in large language models. *Frontiers in Artificial Intelligence*, 8:1622292, 2025. URL: https://www.frontiersin.org/articles/10.3389/frai.2025.1622292, doi:10.3389/frai.2025.1622292.

[20] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2024. URL: https://arxiv.org/abs/2401.11817.

[21] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, and Tim Rocktäschel. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2020. arXiv:2005.11401.

[22] Fei Kang, Dong Chen, Junjie Li, Gang Wan, and Zhe Li. Automated underwater concrete crack width measurement system based on dual lasers and deepcrack network. *Advanced Engineering Informatics*, 2025. Validates precise underwater measurements at 0.5-2 meter distances with millimeter accuracy. doi:10.1016/j.aei.2025.103713.

[23] Guangliang Xu et al. Vision-based underwater target real-time detection for autonomous underwater vehicle. *Frontiers in Marine Science*, 10, 2023. Proposes lightweight CNN for AUV real-time detection with 0.053s computational time. doi:10.3389/fmars.2023.1175625.

[24] E. Leivada et al. Llms lack embodied referents and lived experience. *Linguistics and Philosophy*, 2023. LLMs process text without direct sensorimotor contact with the world, lacking embodied referents and lived experience.

[25] E. Ferrara. Bias in llms: Fundamental effects on cognitive and behavioral simulation. *First Monday*, 2023. Demonstrates how bias in LLMs fundamentally affects their cognitive and behavioral simulation capabilities through systematic misrepresentation.

[26] Qichen Wang, Yuxuan Zhang, Yile Liu, et al. What limits llm-based human simulation: Llms or our design? *arXiv preprint arXiv:2501.08579*, 2025. Identifies fundamental challenges in LLM-based human simulations, including inherent limitations of LLMs and simulation framework design challenges.

[27] Sarah Schröder, Thekla Morgenroth, Ulrike Kuhl, Valerie Vaquet, and Benjamin Paaßen. Large language models do not simulate human psychology. *arXiv preprint arXiv:2508.06950*, 2025. Demonstrates that LLMs fail to react like humans to small but semantically meaningful rewordings of stimuli, showing inconsistency across simulations.

[28] L. Rossi et al. The problems of llm-generated data in social science research. *Sociologica*, 2024. Discusses limitations of LLM-generated data including inability to reproduce human diversity and variance in responses.

[29] L. Xiang et al. The divergence between human and llm-generated tasks: Value-driven and embodied cognition. *arXiv preprint arXiv:2508.00282*, 2024. Shows LLMs fail to exhibit core behavioral signatures of human goal generation despite being prompted with individual human profiles, demonstrating lack of internal mechanisms for value prioritization.

[30] W. Ai et al. Llms struggle to replicate distinct personalities. *Conference on Neural Information Processing Systems*, 2024. Research showing that LLMs struggle to replicate distinct personalities and maintain consistent cognitive patterns.

[31] J. S. Park et al. Generative agents: Interactive simulacra of human behavior. *ACM Symposium on User Interface Software and Technology*, 2023. LLMs face difficulties in replicating authentic group behaviors and maintaining consistent personas across multiple interactions.

[32] T. Gui and O. Toubia. The challenge of using llms to simulate human behavior: A fundamental trade-off between unconfoundedness and ecological validity. *arXiv preprint*, 2023. Identifies fundamental challenges in using LLM-simulated subjects who are blind to experimental design, showing violations of unconfoundedness assumptions.

[33] J. Kim et al. Chatgpt does not replicate human moral judgments. *Nature Scientific Reports*, 2025. Shows systematic biases where ChatGPT overestimates morality of moral behavior and immorality of immoral behavior compared to humans, with limited unique response values.