MARCO: Multi-Agent Real-time Chat Orchestration

Anubhav Shrimal[1], **Stanley Kanagaraj**[1], **Kriti Biswas**[1],
Swarnalatha Raghuraman[1], **Anish Nediyanchath**[1],
Yi Zhang[2] **and Promod Yenigalla**[1]
[1]Retail Business Services, Amazon
[2]AWS Bedrock, Amazon
{shrimaa, kstanly, kritibw, rswarnal, anishned,
yizhngn, promy}@amazon.com

Abstract

Large language model advancements have enabled the development of multi-agent frameworks to tackle complex, real-world problems such as to automate tasks that require interactions with diverse tools, reasoning, and human collaboration. We present MARCO, a Multi-Agent Real-time Chat Orchestration framework for automating tasks using LLMs. MARCO addresses key challenges in utilizing LLMs for complex, multi-step task execution. It incorporates robust guardrails to steer LLM behavior, validate outputs, and recover from errors that stem from inconsistent output formatting, function and parameter hallucination, and lack of domain knowledge. Through extensive experiments we demonstrate MARCO's superior performance with 94.48% and 92.74% accuracy on task execution for Digital Restaurant Service Platform conversations and Retail conversations datasets respectively along with 44.91% improved latency and 33.71% cost reduction. We also report effects of guardrails in performance gain along with comparisons of various LLM models, both open-source and proprietary. The modular and generic design of MARCO allows it to be adapted for automating tasks across domains and to execute complex use-cases through multi-turn interactions.

1 Introduction

Advancements in LLM technology has led to a lot of interest in applying Agents framework to realise solutions which require complex interactions with the environment including planning, tools usage, reasoning, interaction with humans. Recent works (Wang et al., 2024; Huang et al., 2024) demonstrate potential of LLMs for creating autonomous Agents while there are numerous challenges to overcome and provide a seamless experience for end users who interact with the system at a daily basis. LLMs are probabilistic next token prediction systems and by design, non-deterministic which can introduce inconsistencies in the output generation that can prove challenging for features like function calling, parameter value grounding, etc. There are also challenges of domain specific knowledge which can be an advantage and disadvantage at the same time. LLMs have biases inherent in them which can lead to hallucinations, at the same time it may not have the right internal domain specific context which needs to be provided to get the expected results from an LLM.

We present our work on building a real time conversational task automation assistant framework with the following emphasis, (1) **Multi-turn Interface** for, (a) User conversation to execute tasks (b) Executing tools with deterministic graphs providing status updates, intermediate results and requests to fetch additional inputs or clarify from user. (2) **Controllable Agents** using a symbolic plan expressed in natural language task execution procedure (TEP) to guide the agents through the conversation and steps required to solve the task (3) **Shared Hybrid Memory** structure, with Long term memory shared across agents which stores complete context information with Agent TEPs, tool updates, dynamic information and conversation turns. (4) **Guardrails** for ensuring correctness of tool invocations, recover for common LLM error conditions using reflection and to ensure general safety of the system. (5) **Evaluation** mechanism for different aspects and tasks of a multi-agent system.

This is demonstrated in the context of task automation assistant which supports adding usecase tasks to provide users a conversational interface where they can perform their intended actions, making it easier for them to refer to informational documents, interact with multiple tools, perform actions on them while unifying their interfaces. We provide detailed comparison across multiple foundational LLMs as backbone for our assistant tasks like Claude Family models (Anthropic, 2024), Mis-

tral Family models (Jiang et al., 2023, 2024) and Llama-3-8B (AI@Meta, 2024) on Digital Restaurant Service Platform (DRSP) conversations and Retail conversations (Retail-Conv) datasets.

2 Related Work

Improvements to LLM technology through the release of foundational LLMs like GPT-4 (OpenAI et al., 2024), Claude (Anthropic, 2024) and Mixtral (Jiang et al., 2024) has led to a flurry of research around autonomous agents and frameworks (Wang et al., 2024; Huang et al., 2024). Zero shot Chain-of-Thought (COT) reasoning (Kojima et al., 2023) allows LLMs to perform task reasoning by making it think step by step. LLMs can invoke external tools based on natural language instructions. HuggingGPT (Shen et al., 2023b) can coin series of model invocations to achieve complex tasks mentioned by the user. Toolformer (Schick et al., 2023) demonstrates how LLMs can be used as external tools through API invocations selecting the right arguments to be passed from few examples and textual instructions. Agents framework (Zhou et al., 2023) discuss using natural language symbolic plans called (Standard Operating Procedures) SOPs which define transition rules between states as the agent encounters different situations to provide more control over agent behavior along with memory to store relevant state information within the prompt (Fischer, 2023; Rana et al., 2023) or long term context externally (Zhu et al., 2023; Park et al., 2023). Amazon Bedrock Agents [1] provide interface to quickly build, configure and deploy autonomous agents into business applications leveraging the strength of foundational models, while the framework abstracts the Agent prompt, memory, security and API invocations. LangGraph [2] is an extension of LangChain which facilitates the creation of stateful, multi-actor applications using large language models (LLMs) by adding cycles and persistence to LLM applications thus enhancing their Agentic behavior. It coordinates and checkpoints multiple chains (or actors) across cyclic computational steps. While these frameworks present novel ways for LLMs to act in a desired behaviour, they often have accuracy-latency trade-off where to improve on the accuracy the system latency increases due to multi-step planning and thinking (Yao et al., 2023; Wei et al., 2023). Our proposed solution, MARCO, not only interacts with user in a multi-turn fashion but also has multi-turn conversation with deterministic multi-step functions which comprises of pre-determined business logic or task execution procedure (TEP) requiring agents only at intelligent intervention related steps. Along with the usecase TEPs, multi-step functions and robust guardrails to steer LLM behaviour, MARCO is able to perform complex tasks with high accuracy in less time as detailed in subsequent sections.

[1] Amazon Bedrock Agents User Guide
[2] LangGraph library

3 MARCO: Multi-Agent Real-time Chat Orchestration

In this section, we discuss our approach for MARCO. Section 3.1 formulates the problem statement in terms of Task Automation via real-time chat, followed by components of MARCO in section 3.2 and the evaluation methods on performance and latency for MARCO in section 3.3.

3.1 Problem Statement

Given an user (*Actor*), who wishes to perform a task with intent $I \in \{$*OOD, Info, Action*$\}$; where *Out-Of-Domain (OOD)* intent is defined as any user query which is not in scope of the system such as malicious query to jailbreak (Shen et al., 2023a; Rao et al., 2024) the system, foul language or unsupported requests, *"Info"* intent is defined as getting information from predefined data-sources and indexed documents (D_{index}), and *"Action"* intent is defined as a performing a usecase related task (u_x) which involves following a series of instructions/steps (Task Execution Procedure, TEP_x) defined for the usecase and accordingly invoking the right set of tools/functions ($F^x_* = \{F^x_1, F^x_2, ..., F^x_n\}$) with the identified required parameters ($P^x_* = \{P_{F^x_1}, P_{F^x_2}, ..., P_{F^x_n}\}$) for each function respectively. The objective for a task automation system is to, (1) interpret the user intent I for each query, (2) identify the relevant usecase u_x, (3) understand the steps mentioned in its TEP_x, (4) accordingly call the right sequence of tools F^x_* with required parameters P^x_*, (5) correlate TEP_x, tool responses and requirements and conversation context to communicate back with the user, and (6) be fast and responsive for a real-time chat.

An example scenario is shown in figure 1 where User first asks "*The sale of certain item is going down in my restaurant. Can you please help me find out why?*", i.e. $I = Action$ for which

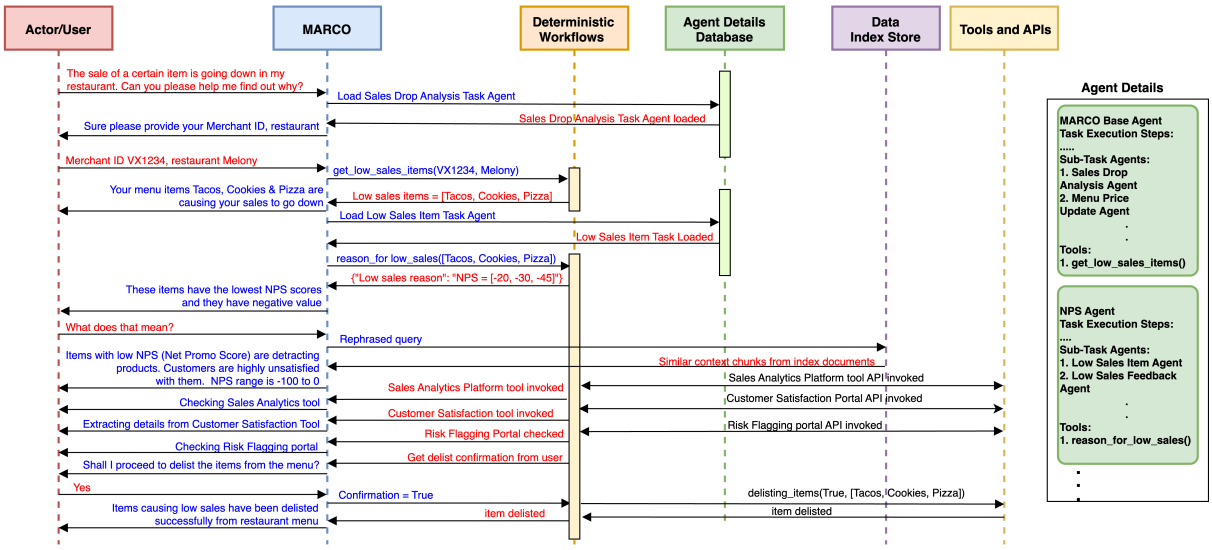


Figure 1: Multi-Agent Conversation Flow in MARCO Framework. This diagram illustrates the complex interactions within the *MARCO* system as it addresses a user's query about declining sales. It showcases MARCO's orchestration of multiple components including the MARCO Base agent, specialized task agents, deterministic multi-step workflows, data stores, and external tools/APIs. The figure demonstrates MARCO's capability to manage multi-turn communications with both the user and various system components, highlighting its process of task decomposition, information gathering, analysis, and action execution in response to real-world business scenarios.

MARCO then loads the agent with TEP for Sales Drop Analysis usecase (TEP_{sd}) and then goes on to call relevant function F=[*get_low_sales_item*, *reason_for_low_sales*] with respective required parameter values *merchant_id* and *restaurant_name*. It is worth noting that the interaction with MARCO is multi-turn, both with the user as well as the functions being called where the functions may provide intermediate communications or ask for information to proceed further (for example, *confirmation=True*).

3.2 MARCO – Components

MARCO built for task automation has 4 primary LLM components, (i) Intent Classifier, (ii) Retrieval Augmented Generation (RAG) to answer domain related informational queries, (iii) MARS for tasks orchestration and execution, and (iv) Guardrails. The sections below cover each of the component, except for RAG where the implementation details are out of scope for this paper.

3.2.1 Intent Classifier

Intent Classifier's (IC) primary role is to understand the intent behind an incoming user message considering the conversation context, and to seamlessly orchestrate between RAG for answering informational queries and Multi-Agents system (MARS) to execute supported tasks. IC also takes the role of first level guardrails to identify and gracefully reject queries to protect the underlying modules from harmful jailbreak instructions and *Out-Of-Domain* (OOD) queries. At a high level IC performs intent classification into one of the three supported classes *{Info, Action, OOD}*, leveraging language understanding capability of LLMs. Major challenges faced by intent classifier can be found in Appendix A.6.

3.2.2 Multi-Agent Reasoner and Orchestrator (MARS)

When a user query is classified as an I = *Action* intent, the chat conversation history is redirected to *MARS* (Multi-Agent Reasoner and Orchestrator) module which is a Multi-Agent system responsible for (1) understanding the user's request and tool responses in the chat context (2) planning and reasoning for the next action according to the Task Execution Procedure (TEP) steps, (3) selecting relevant LLM Agent for the task, and (4) invoking the relevant tools/tasks with their required parameters. The key component of MARS are the LLM Agents, which we call *Task Agents*. These Task Agents comprise of their own TEP steps, tools/-functions also known as *Deterministic Tasks*, *Sub-Task-Agents* (dependent Task Agents) and common instructions for reasoning and output formatting. We will explain each of these in detail:

Deterministic Tasks: Task Execution Procedure

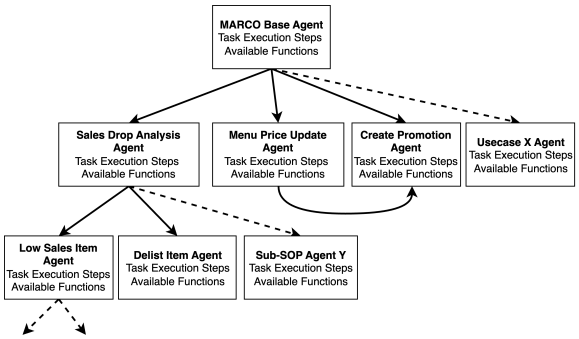


Figure 2: Multi-Agents Hierarchy example for Digital Restaurant Service Platform dataset. A directed acyclic graph in which each agent has it's own Task Execution Procedure (TEP) steps, functions and dependent Sub-Task Agents.

(TEP) steps can be very complex with multiple instructions and steps to follow based on a given usecase scenario. While some of these steps require high judgement and reasoning (understanding natural language to parse required arguments, intents, performing checks defined in plain text without writing explicit code), most of the steps in the TEP are deterministic sequence of API calls, processing and propagating the output gathered from API_1 to API_2 and so on. Such sequence of deterministic steps can be encapsulated as a single tool to the LLM Agent, which when called performs the sequence of these deterministic steps and communicates with the agent intermittently with updates or any high judgement reasoning or inputs required by the underlying APIs (for example refer to Appendix A.9).

Task Agents: A usecase TEP can be divided into multiple Sub-Tasks which are logical abstractions of complex steps inside the TEP. For example, if a usecase u_x has sub-branches {a, b, c}, each with their own set of steps to follow, then each can be created as a Task Agent (A) where Agent A_x has agents $\{A_a, A_b, A_c\}$ as it's child Task-Agents. Each of these child Agents may further have their own children Agents based on their TEP complexity. A Task Agent has the steps comprised in its TEP along with the list of available determinisitic tasks/functions that the particular Agent can utilize, for e.g. The "Sales Drop Analysis" usecase Agent (A_{sd}) may have a function named *get_low_sales_items(merchant_id, restaurant_name)* function but will not have *update_menu_item_price(menu_item, price)* function as it is not a valid dependency. An Agent also has the information of the immediate child Sub-Task-Agents in its hierarchy so that it can invoke another child agent if required during its planning. Figure 2 shows an example multi-agent hierarchy for DRSP dataset where *MARCO Base Agent*, using which the system is first initialized, is the main agent with its own TEP steps, tools and Sub-Agents i.e. the usecases which are added onto the platform.

Agent's LLM input prompt has sub-agents, tools, reasoning and formatting instructions and chat history embedded using which it has to auto-regressively generate the output. We prompt the underlying LLM to generate the "*message*" which is to be conveyed to the *Actor* and the corresponding "*action*" which could be to invoke a deterministic task with the arguments the Agent provides or switching to a child Task-Agent. Appendix A.7 provides more details on input and output formatting.

3.2.3 Guardrails

LLMs exhibit stochastic behavior, generating varying outputs for the same input. They are susceptible to hallucination (Bang et al., 2023; Guerreiro et al., 2023), producing responses with fabricated or inaccurate information. It is crucial to establish mechanisms to steer LLMs in the desired direction for reliable systems. We introduce guardrails to identify issues and prompt the LLM-Agents to reflect on their mistakes, correcting their responses. Common issues and proposed guardrail solutions are: (1) **Incorrect Output Formatting:** Generating incorrect formats despite detailed instructions, causing parsing issues. If parsing fails, a reflection prompt is added to the Agent's chat history, and sent for a retry. (2) **Function Hallucination:** Hallucinating non-existent function names, even when prompted to use only existing tools. Our guardrails checks if the generated function name exists in the available tools and Sub-Agents. If not, reflection prompt is added. (3) **Function Parameter Value Hallucination:** When making function calls with required parameters, LLMs sometimes hallucinate parameter values instead of asking relevant questions to the user. This often occurs due to pre-trained dataset biases because they have seen this pattern frequently during pre-training, making it challenging to unlearn using prompting techniques. For each function parameter p, the module checks if p is part of the function schema; if not, p is removed (e.g., for *get_low_sales_items(merchant_id, restaurant_name)*, the Agent also generated *menu_item* as a parameter). The parameter value for non–

boolean parameters is grounded to be present in the *Actor* message history; if not, it is classified as hallucination (e.g., *Actor* said, "update menu price of item X to $50" and Agent generated *marketplace="US"* which was not mentioned by the Actor). (4) **Lack of Domain Knowledge:** Although pre-trained LLMs possess good general world knowledge, they may lack certain domain-specific knowledge, especially in lesser–known domains. We define a list of static rules for each parameter based on the type, constituent values, length and more (e.g., "merchant_id value has a minimum_length=6 and maximum_length=8, is an alphanumeric string"). The guardrails module checks if the generated value satisfies these rules; if not, a reflection prompt with rule failures is added. Parameter properties and definitions are also introduced in the Agent prompt as *<helpful_definitions>...</helpful_definitions>* to provide explicit in-domain knowledge for e.g., "<helpful_definitions>merchant_id is 6-8 character alphanumeric string, restaurant_code is 4-5 character alphanumeric string</helpful_definitions>", which helps the Agent to then disambiguate these values when provided without names by the *Actor* (e.g. "VX1234, BL123"). The number of retries with reflection is limited to 2 (*NUM_RETRIES=2*) for real-time chat system latency. Appendix Algorithm 1 provides detailed flow of guardrails.

3.2.4 Context Sharing

As MARCO has multiple components (IC, MARS, RAG) and is a multi-turn multi-agent conversation system, it needs a mechanism to share the context amongst each other. Along with the usual roles of *[[SYSTEM], [USER], [AGENT]]* similar to Bedrock's Claude messages API format[3], we introduce separate roles for function responses and guardrails, *[FUNCTION_RESPONSE], [GUARDRAILS]*, respectively. This allows LLM-Agents to better differentiate each message in the chat history as the conversation is multi-turn from both *Actor* and *Deterministic tasks* (for example Figure 1 *reason_for_low_sales()* task communicates multiple times to MARCO), and it prevents jailbreaking by malicious *Actors*. When a Parent Agent ($Agent_p$) loads its Child Agent ($Agent_c$), the *[SYSTEM]* prompt is updated with $Agent_c$ details and a message is added to the chat history to capture that an agent switch has occurred. The common chat history thread is shared among all Task-Agents for a chat session, as any information provided to $Agent_p$ by the *Actor* or a *function response* might be useful for the executing $Agent_c$'s task execution procedure (TEP) steps.

[3] Bedrock Claude messages API documentation

3.3 Evaluation Methods

A real-time task automation system should have highly accurate execution as well as fast turnaround time. Keeping these tenets in mind, we evaluate MARCO components on quality and accuracy of generated responses along with time taken to produce such outputs. For evaluating *MARS* we compare the expected function call and parameter $(F_i^x, P_{F_i^x})$ with the generated function call and parameter $(\hat{F}_i^x, \hat{P}_{F_i^x})$ whenever an action is expected in test data. We also implemented an LLM response evaluation prompt which takes in two response messages (m_1, m_2) and returns *True* if semantics of m_1 and m_2 are the same else *False*. An manual audit based evaluation is also performed to validate the efficacy of our LLM response evaluation prompt (LLM evaluation prompt detailed in Appendix A.8). Both, the generated function call and response message semantics, should be evaluated as correct with the ground truth to mark the complete generated output as valid. We calculate the accuracy as the number of test cases where MARS's complete generated output is valid. For each component we also calculate and compare the latency and cost of response generation as it is a real-time chat system.

4 Experimental Setup

Dataset: For our experiments, we curated two conversational orchestration test datasets, Digital Restaurant Service Platform (DRSP-Conv) and Retail-Conv, each with 221 and 350 multi-turn conversations in the restaurant services and retail services domain respectively. These conversations are a mix of *Out-Of-Domain (OOD), Action* and *Info* queries with multi-turn interactions with both, User and Deterministic Tasks (an example conversation flow in the dataset is shown in figure 1 for DRSP-Conv). The dataset covers usecases along with their natural language Task Execution Procedure (TEP) steps, supported functions (deterministic tasks and utility tools [4]) and sub-task agents. Each test conversation has multiple *Assistant* (Agent) messages (replying to the user, loading

[4] utility tools are simple functions to get specific data, e.g., get_menu_item_name(), get_menu_item_price(), etc.

an agent, calling a deterministic task, or answering an informational query). We use these datasets for evaluating MARCO on the defined performance metrics. Hyper-parameter details mentioned in Appendix A.2.

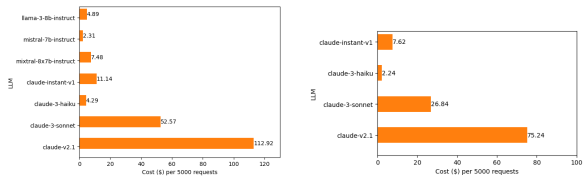
Baseline: We implement MARCO with a single agent-based prompt as a baseline to compare with our multi-agent proposed solution, on performance, latency and cost. To achieve this, the usecase subtask TEP steps in the datasets were combined into the parent agent TEP steps to create a single agent TEP and the datasets were modified accordingly to support the single agent baseline.

	DRSP-Conv dataset			
	With All Reflection Guardrails (retries=2)		No Guardrails (retries=0)	
Model Name	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)
llama-3-8b-instruct	42.44 ± 2.01	3.75	15.93 ± 0.98	**1.9**
mistral-7b-instruct	66.33 ± 1.04	4.92	59.28 ± 1.06	2.9
mixtral-8x7b-instruct	40.64 ± 1.51	17.77	32.67 ± 0.38	15.55
claude-instant-v1	74.38 ± 1.4	3.25	53.12 ± 3.83	2.53
claude-3-haiku	84.8 ± 0.88	**2.14**	**75.2 ± 0.87**	2.24
claude-v2.1	88.51 ± 0.76	8.44	64.52 ± 1.04	6.61
claude-3-sonnet	**94.48 ± 0.59**	5.61	66.34 ± 0.82	4.07
	Retail-Conv dataset			
	With All Reflection Guardrails (retries=2)		No Guardrails (retries=0)	
Model Name	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)
llama-3-8b-instruct	49.68 ± 1.55	3.44	17.82 ± 1.12	**1.64**
mistral-7b-instruct	55.32 ± 0.77	4.89	50.72 ± 0.66	3.06
mixtral-8x7b-instruct	48.31 ± 0.60	12.94	40.49 ± 0.93	5.96
claude-instant-v1	76.61 ± 0.81	4.14	60.56 ± 0.24	2.94
claude-3-haiku	87.82 ± 0.44	**2.45**	77.66 ± 1.01	2.43
claude-v2.1	92.34 ± 0.49	8.2	**78.87 ± 0.61**	6.95
claude-3-sonnet	**92.74 ± 0.49**	5.85	60.89 ± 0.81	4.61

Table 1: LLMs performance comparison for MARCO with and without guardrails on DRSP-Conv and Retail-Conv datasets averaged across 5 runs.

5 Experiments & Results

In this section we detail the various experiments to evaluate our proposed solution, MARCO, on task specific performance, operational performance (latency, run-time cost), scalability and ablations.



(a) Cost of MARS. (b) Cost of Intent Classifier.

Figure 3: Cost ($) of MARCO components for every 5000 requests using various LLMs.

MARS Operational Performance: We compare the accuracy, latency of *MARS* (Multi-Agent Reasoner and Orchestrator) using various open-source (llama-3-8B, mistral-7B, mixtral-8x7B) and proprietary instruction-tuned LLMs (claude-instant-v1, claude-v2.1, claude-v3-haiku, and claude-v3-sonnet) in Table 1. We observe that *claude-3-sonnet* performs best with 94.48% and 92.74% accuracy and 5.61 and 5.85 seconds latency including all reflection guardrails for DRSP-Conv and Retail-Conv datasets respectively. Sonnet is also 30% faster and 60% cheaper than *claude-v2.1*, making it cost-effective as shown in figure 3a for MARCO implementation costs using various LLMs assuming 5000 requests with average input and output tokens calculated empirically (refer Appendix A.5). Open-source LLMs underperform even with reflection guardrails, suggesting the need for fine-tuning as future work. We found a Cohen's kappa (Cohen, 1960) of 0.65 (96.66% agreement) between human auditors and our LLM semantics similarity matching prompt for evaluation, indicating a high level of agreement. Intent Classifier has 94.53% using *claude-v3-sonnet* 3-class classification accuracy with a latency of 1.98 seconds (refer Appendix Table 5).

	DRSP-Conv dataset			
	With All Reflection Guardrails (retries=2)		No Guardrails (retries=0)	
Model Name	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)
MARCO Single-Agent (claude-3-sonnet)	82.71 ± 0.68	6.89	**70.63 ± 2.77**	5.72
MARCO Multi-Agent (claude-3-sonnet)	**94.48 ± 0.59**	**5.61**	66.34 ± 0.82	**4.07**
	Retail-Conv dataset			
	With All Reflection Guardrails (retries=2)		No Guardrails (retries=0)	
Model Name	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)
MARCO Single-Agent (claude-3-sonnet)	88.38 ± 0.90	9.77	**80.07 ± 0.77**	8.81
MARCO Multi-Agent (claude-3-sonnet)	**92.74 ± 0.49**	**5.85**	60.89 ± 0.81	**4.61**

Table 2: Comparing MARCO single-agent and multi-agent with and without guardrails on DRSP-Conv and Retail-Conv datasets averaged across 5 runs.

Single-Agent Baseline vs Multi-Agent (MARS) performance: Through this experiment, we aim to demonstrate the effectiveness of MARS against a Single-Agent baseline covering all usecases. Table 2 shows that our proposed multi-agent system, MARCO, outperform single-agent baseline by +11.77% and +4.36% with all guardrails included on respective datasets. Also, the latency of Single-Agent baseline is on average 44.91% higher and increases the cost by 33.71% ($70.29 per 5k requests) compared to MARS ($52.57 per 5k requests) due to longer prompt length for the Agent.

Effects of Reflection Guardrails: Through this experiment, we compare MARS's performance when all reflection guardrails, as discussed in section 3.2.3, are added vs without adding any guardrails. As shown in table 1 adding reflection guardrails provides a +28.14% and +31.85% boost in accuracy while increasing the latency only by

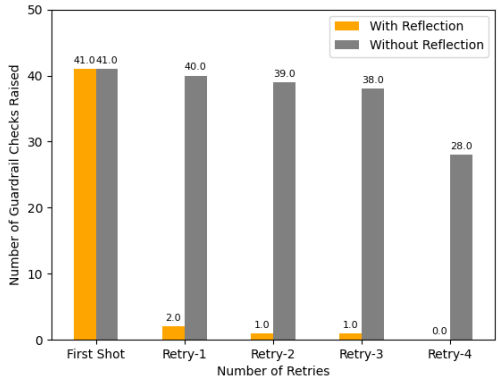


Figure 4: Impact of Reflection Prompts on Guardrail Error Recurrence During Retries. This graph compares the number of guardrail errors persisting across multiple retry attempts, with and without the use of reflection prompts. It demonstrates that incorporating reflection prompts significantly reduces error recurrence, typically resolving issues within the first retry. In contrast, retrying without reflection shows a gradual decrease in errors but fails to eliminate them entirely even after four attempts.

1.54 and 1.24 seconds on average for DRSP-Conv and Retail-Conv respectively. Figure 4 illustrates the impact of our proposed reflection guardrails, where the first retry with reflection resolves all but two errors, whereas without any reflection prompt (using the original prompt on retires), error rates remain high even after four retries. Appendix A.3 shows the effects of removing each reflection type. On further deep dive we observe that claude-3-haiku has better performance than larger counterparts (claude-3-sonnet and claude-v2.1) when no guardrails are applied primarily due to its effectiveness in following output formatting instructions and generating correct outputs more often. Hence Haiku could be a viable option when cost of retries and latency have to be reduced further.

Effects of Temperature, Input & Output Token Lengths: Increasing temperature hyperparameter allows an LLM to be more creative while generating a response. We observed that setting the value *temparature=0* gives the best accuracy for MARS (Appendix Table 5), which is understandable as Task Execution Procedure (TEP) instruction following and function calling should be reliable and should not vary. Also, with increasing number of input and output tokens, the latency of MARCO increases (Appendix A.4).

Conclusion

We presented MARCO, a multi-agent real-time chat orchestration framework for automating tasks using large language models (LLMs) addressing key challenges in utilizing LLMs for complex, multi-step task execution with high accuracy and low latency including reflection guardrail prompts for steering LLM behaviour and recover from errors leading to +30% accuracy improvement. We demonstrated MARCO's superior performance with up to +11.77% and +4.36% improved accuracy against single agent baseline for two datasets, DRSP-Conv and Retail-Conv, and improved latency by 44.91% and 33.71% cost reduction. The modular and generic design of MARCO allows it to be adapted for automating tasks across various domains wherever complex tasks need to be executed through multi-turn interactions using LLM-powered agents.

References

AI@Meta. 2024. Llama 3 model card.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *Preprint*, arXiv:2302.04023.

Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46.

Kevin A. Fischer. 2023. Reflective linguistic programming (rlp): A stepping stone in socially-aware agi (socialagi). *Preprint*, arXiv:2305.12647.

Nuno M. Guerreiro, Duarte Alves, Jonas Waldendorf, Barry Haddow, Alexandra Birch, Pierre Colombo, and André F. T. Martins. 2023. Hallucinations in large multilingual translation models. *Preprint*, arXiv:2303.16104.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of llm agents: A survey. *Preprint*, arXiv:2402.02716.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud,

Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *Preprint*, arXiv:2401.04088.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners. *Preprint*, arXiv:2205.11916.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, and et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Preprint*, arXiv:2304.03442.

Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. 2023. Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning. *Preprint*, arXiv:2307.06135.

Abhinav Rao, Sachin Vashistha, Atharva Naik, Somak Aditya, and Monojit Choudhury. 2024. Tricking llms into disobedience: Formalizing, analyzing, and detecting jailbreaks. *Preprint*, arXiv:2305.14965.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. 2023a. "do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. *Preprint*, arXiv:2308.03825.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023b. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Preprint*, arXiv:2303.17580.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, Shiding Zhu, Jiyu Chen, Wentao Zhang, Xiangru Tang, Ningyu Zhang, Huajun Chen, Peng Cui, and Mrinmaya Sachan. 2023. Agents: An open-source framework for autonomous language agents. *Preprint*, arXiv:2309.07870.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. 2023. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*.

A Appendix

A.1 Discussion

While as part of this work our experiments are focused towards Digital Restaurant Service and Retail task automations, the design for MARCO is generic LLM Agents based framework and can be adapted to any domain where the system is required to follow standard task execution steps to solve for a usecase while using set of available tools and interacting with an end user. Also, the guardrails and evaluation methods are generic for such a framework. As Intent Classifier, RAG and MARS are independent modules, we execute them in parallel to reduce the latency of our real-time chat system. The output from MARS or RAG is picked according to IC's classification.

A.2 Hyper-parameters:

For all experiments, unless specified otherwise, we used the underlying LLM as *claude-3-sonnet* with *temperature=0*, *max_output_tokens=1000* and *Top-P*, *Top-K* values as defaults. We use LLM APIs provided by Amazon Bedrock dated July 1, 2024 for output generation. The maximum number of retires on any guardrail failure was set to 2, and if the issue still persisted, a constant "Facing Technical Issue" response was sent back. We ran each experiment five times and published the average and standard deviation for the results. We publish

Algorithm 1: MARCO Reflection Guardrails

Input: $F^x_* = \{F^x_1, F^x_2, ..., F^x_n\}$, $P^x_* = \{P_{F^x_1}, P_{F^x_2}, ..., P_{F^x_n}\}$, /* list of available tools, Sub-Agents & respective parameters in $Agent_x$ */

R /* LLM Agent generated response string */

Output: $Agent_x$ updated context

if *invalid_output_format(*R*)* **then**
  $Agent_x$.add_to_context("Output R is not as per required formatting guidelines.")
else
  $\hat{F}^x_i, \hat{P}_{F^x_i} \leftarrow$ parse_llm_response(R) /* LLM generated Function & corresponding parameters */
  if $\hat{F}^x_i \notin F^x_*$ **then**
    $Agent_x$.add_to_context("Function F^x_i not present in Agent tools and Sub-Agents.")
  for $p \in \hat{P}_{F^x_i}$ **do**
    if $p \notin P_{F^x_i}$ **then**
      $\hat{P}_{F^x_i} \leftarrow \hat{P}_{F^x_i} \setminus p$ /* remove p from generated parameters set */
    else if *p.value()* $\notin$ Agent$_x$*.user_messages()* **then**
      /* parameter value not present (grounded) in user messages */
      $Agent_x$.add_to_context("Value of p not provided by the user.")
    else
      rules $\leftarrow$ get_predefined_rules_errors(p) /* example "length(p) should be $\leq$ 10" */
      $Agent_x$.add_to_context("Following rules not satisfied by p: *rules*.")

the cost calculation numbers with AWS Bedrock pricing [5] in this work.

A.3 Reflection Guardrails Ablation

Table 3 performs an ablation of each of the reflection prompts discussed in section 3.2.3. The results show that each reflection prompt contributes to the performance enhancement of MARCO without which the performance drops significantly on DRSP-Conv and Retail-Conv datasets. The latency also does not increase much due to re-trying with reflection with an average increase of only 1.54 and 1.24 seconds respectively when adding all guardrails to the system in *claude-3-sonnet*.

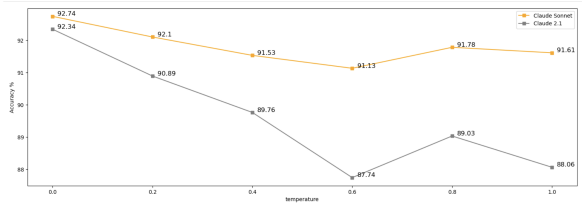


Figure 5: Effect of temperature hyper-parameter on MARS performance.

A.4 Effects of Temperature, Input & Output Token Lengths:

Effects of Temperature: We vary the temperature hyper-parameter at an increment of +0.2 from 0 to 1 and compare the performance accuracy of MARS using *claude-3-sonnet* and *claude-v2.1*. The results suggest that *temperature=0* performs the best for MARCO.

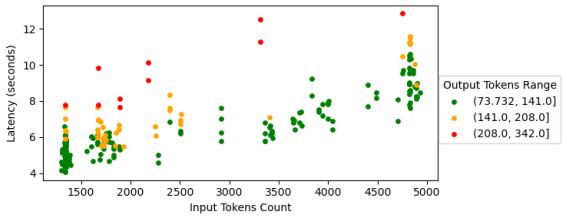


Figure 6: Correlation between number of input and output tokens in LLM prompt and response latency for MARS using *claude-3-sonnet*.

Effects of Input and Output Tokens on Latency: In figure 6 we plot the latency of MARS using *claude-3-sonnet* with respect to input tokens (x-axis). We further color code each instance on the plot based on the number of output tokens generated within a given range. The results show a correlation between the growing number of input tokens leading to an increase in the latency while also having large number of output tokens for similar input token length leading to further increase in the latency.

A.5 Cost Analysis

To calculate the cost of various LLM version we assume that the task automation system has on

[5] Bedrock API Pricing Documentation

Model Name	With All Reflections		Without Incorrect Formatting Reflection		Without Function Hallucination Reflection		Without Parameter Grounding Reflection		Without Parameter Static Rules Reflection		Without A Reflection (retries = 0)	
	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)	Accuracy (%) ± Std dev	Latency (secs)
	DRSP-Conv dataset											
llama-3-8b-instruct	42.44 ± 2.01	3.75	16.29 ± 0.64	4.38	41.18 ± 0.96	3.72	41.09 ± 1.41	3.7	41.18 ± 1.2	3.73	15.93 ± 0.98	**1.9**
mistral-7b-instruct	66.33 ± 1.04	4.92	64.52 ± 0.61	5.17	66.88 ± 1.3	5.01	64.34 ± 0.81	5.02	65.79 ± 0.52	5.16	59.28 ± 1.06	2.9
mixtral-8x7b-instruct	40.64 ± 1.51	17.77	39.46 ± 1.08	20.42	41.54 ± 0.98	24.15	40.82 ± 2.43	23.61	40.81 ± 1.26	20.93	32.67 ± 0.38	15.55
claude-instant-v1	74.38 ± 1.4	3.25	72.5 ± 1.4	3.37	74.38 ± 1.4	3.14	74.38 ± 2.61	2.85	75.0 ± 0.0	2.9	53.12 ± 3.83	2.53
claude-3-haiku	84.8 ± 0.88	**2.14**	**84.43 ± 1.65**	**2.13**	83.98 ± 1.3	**2.54**	78.73 ± 0.78	**2.37**	81.09 ± 1.08	**2.25**	**75.2 ± 0.87**	2.24
claude-v2.1	88.51 ± 0.76	8.44	68.42 ± 1.34	9.49	88.42 ± 0.68	8.22	86.24 ± 1.09	8.35	86.15 ± 1.45	8.19	64.52 ± 1.04	6.61
claude-3-sonnet	**94.48 ± 0.59**	5.61	73.39 ± 0.5	6.23	**94.03 ± 0.59**	5.41	**91.04 ± 1.08**	5.5	**91.86 ± 0.46**	5.26	66.34 ± 0.82	4.07
	Retail-Conv dataset											
llama-3-8b-instruct	49.68 ± 1.55	3.44	20.32 ± 0.46	4.99	48.47 ± 0.53	3.45	46.77 ± 1.76	3.41	47.58 ± 1.18	3.36	17.82 ± 1.12	**1.64**
mistral-7b-instruct	55.32 ± 0.77	4.89	54.68 ± 1.29	4.96	54.03 ± 0.57	4.55	55.16 ± 1.04	4.74	55.24 ± 1.56	4.82	50.72 ± 0.66	3.06
mixtral-8x7b-instruct	48.31 ± 0.60	12.94	47.34 ± 1.5	11.82	48.87 ± 2.18	10.35	50.32 ± 1.75	11.24	48.87 ± 2.82	10.22	40.49 ± 0.93	5.96
claude-instant-v1	76.61 ± 0.81	4.14	68.95 ± 0.57	4.32	75.56 ± 0.61	4.23	60.56 ± 0.34	**2.94**	74.03 ± 1.05	4.28	60.56 ± 0.24	2.94
claude-3-haiku	87.82 ± 0.44	**2.45**	87.58 ± 0.78	**2.29**	86.21 ± 1.47	**2.28**	82.74 ± 0.33	3.1	85.08 ± 0.57	**3**	77.66 ± 1.01	2.43
claude-v2.1	92.34 ± 0.49	8.2	**88.31 ± 0.57**	8.6	90.32 ± 1.14	6.22	**87.98 ± 0.44**	8.63	89.68 ± 0.67	8.48	**78.87 ± 0.61**	6.95
claude-3-sonnet	**92.74 ± 0.49**	5.85	66.53 ± 0.81	7.93	**91.53 ± 0.64**	8.55	83.39 ± 2.91	6.61	**90.89 ± 0.54**	6.16	60.89 ± 0.81	4.61

Table 3: LLMs performance comparison for MARCO by removing different type of reflection guardrails on DRSP-Conv and Retail-Conv datasets averaged across 5 runs.

average:

1. 100 active users per day,
2. 50 messages per chat,
3. X input tokens per LLM request (calculated empirically from our experiments in table 1),
4. Y output tokens per LLM request (calculated empirically from our experiments in table 1).
5. $\$Z_i/1000$ input tokens and $\$Z_o/1000$ output tokens cost of LLM API invocation.

Then the cost of the system (C) in product to serve 5k requests ($100 * 50 = 5000$) is calculated as follows:

$$C = (5000 * X * Z_i/1000) + (5000 * Y * Z_o/1000) \quad (1)$$

Single-Agent baseline has on an average 3946 input and 148 output tokens which leads to a total of $70.29 per 5k requests cost using *claude-3-sonnet*.[6] Pricing for MARCO components (IC and MARS) for various LLMs is shown in figure 3. The results state that using *claude-v2.1* is 2.14 times costly compared to *claude-3-sonnet*. Similarly, for Intent Classifier using *claude-3-sonnet* followed by *claude-instant-v1* is an ideal choice to keep latency and cost in mind while also comparing the performance (refer table 5).

A.6 Intent Classifier prompting techniques

In this section we explain the various prompting techniques that we employed to improve the performance of Intent Classifier. The primary objective of the Intent Classifier is to classify between *I=Info, I=Action* intents, while also adeptly managing casual conversational contexts such as greetings, out-of-domain inquiries, and potential jailbreak attempts. Major challenges that we have addressed for IC are:

1. Disambiguate closely related queries that can have different meaning and should be classified accordingly. For e.g., "What is the menu price of *a* food item?" and "What is the menu price of *my* food item?", while the former is an *I=Info* query to understand the definition of menu price, the latter is to know the existing menu price of user's food item which needs to fetch the details from a tool and hence should be classified as *I=Action*.
2. Multi-turn Conversation understanding: User

[6] Pricing of Bedrock API Documentation

Prompting Technique	Average Accuracy (%) ± Std dev	Average Latency
Zero Shot	89.26% ± 0.47	2.99
Chain of Thought	89.68% ± 1.56	**1.98**
One Vs. All	91.37% ± 0.47	**1.98**
Few Shot	**94.32%** ± **0.94**	2.43

Table 4: Intent Classifier performance comparison based on varying prompting techniques.

Model Name	Average Accuracy (%) ± Std dev	Average Latency
mixtral-8x7b-instruct	65.47% ± 0.008	**1.62**
mistral-7b-instruct	75.58% ± 0.004	1.96
claude-3-haiku	90.32% ± 0.88	1.98
claude-v2.1	92.42% ± 0.47	5.02
claude-instant-v1	94.32% ± 0.94	2.43
claude-3-sonnet	**94.53%** ± **0.88**	1.98

Table 5: Comparing Intent Classifier performance and latency using various LLMs.

can ask an action query and switch to informational query in the middle or vice-versa. The follow-up user messages can be partial and derive from the conversation context heavily (e.g., "What does *this* mean?"). This requires IC to have nuanced conversation understanding to classify user message accurately.

3. Handling domain specific acronyms: Conversation and tasks can refer to internal keywords and acronyms not present in common language usage. Knowledge of these are required to understand the context of conversation to act on it accurately.

4. Context length: Conversations can be lengthy and run into several hundreds of tokens. Classifier needs to account for the complete context to make decisions.

Table 4 provides a comprehensive comparative analysis of the effectiveness of each prompting technique. Initially, we established a zero-shot prompt as our baseline, achieving an accuracy of 89.26% with a latency of approximately 3 seconds using the *claude-instant-v1* model. Subsequently, we investigated the efficacy of chain-of-thought prompting. This method involved presenting a sequence of yes/no questions within the same prompt to steer the Intent Classifier towards the accurate intent selection. (An illustrative example from the prompt is as follows: "*Is the context directly related to digital restaurant platform or business? If Yes , Go to next step, If no Intent = Out of Context, Is the User asking the meaning or definition of retail terminologies?, If Yes, Intent = Information, If No, Go to next step*"). Despite its implementation, this prompting technique yielded a negligible uplift of less than 0.5% in accuracy. Another approach explored was one-vs-all prompting. Herein, we explicitly defined one intent (e.g. *I=Info*) while categorizing the remainder as another intent. This technique proved efficient in mitigating ambiguity in the instructions, consequently yielding a 2% improvement from the original baseline. Furthermore, by formulating a prompt with explicit instructions and examples for ambiguous scenarios (few shot prompting), we achieved the most significant enhancement thus far, with a 5% uplift from the baseline performance.

In another experiment, we evaluated the performance of various instruct-tuned large language models (LLMs), the outcomes of which are delineated in Table 5. The *claude-3-sonnet* model emerged with the highest accuracy slightly exceeding 94%, whereas the Mixtral model exhibited superior latency measures fine-tuning which will be a future work for improved accuracy.

A.7 LLM Agents Input Prompts & Output Formatting

In this section we go deeper into the details of how we prompt our Task-Agents (LLM Agents in MARS) to get desired reasoning and output.

LLM Input Prompt: Below mention is a sample LLM Agent's prompt using which we intialise all our Task-Agents where details like *agent_name, agent_purpose, agent_task_execution_steps, sub_task_agents, tools, history, user_message* are dynamic variables replaced with the actual values on the fly using Agent's internal state. We employ techniques like Chain-of-thought reasoning, guiding LLM to complete the prefix string (*[Agent]<thinking>*) so that it steers in the required direction, output formatting instructions and XML tags to define segments in the prompts carefully.

```
{{agent_name}}, {{agent_purpose}}
<TEP_STEPS>
{{agent_task_execution_steps}}
</TEP_STEPS>
Sub-Tasks:
<sub_tasks>
{{sub_task_agents}}
</sub_tasks>
Tools:
<tools>
{{agent_tools}}
</tools>
Place to Add important instructions:
<instructions>
{{instructions}}
</instructions>
Placeholder for chat history
<history> {{history}} </history>
```

LLM Output: We prompt the LLM to generate the following output format, which is then parsed to get relevant actions:

```
<response>{
  "content": "The message to be conveyed
      back to the user.",
  "function_call": {
    "name": "function name",
    "arguments": "{\"Arg1\": \"
       Arg1_value\"}"
  }
}</response>
```

A.8 LLM Evaluation Prompt

In this section we detail the LLM based semantic similarity matching LLM prompt for evaluating

MARS Agents' responses. While verifying the generated function call and corresponding parameters is easy as they can be matched after parsing from the string with the ground truth deterministically, it can be challenging to match whether the LLM generated response back to the *Actor/User* is same as the intended string in ground truth test set. Traditionally a manual audit is conducted to look at the generated string and ground truth string to identify if both have the same semantics or meaning. This can be a time taking and costly task depending on the size of your test dataset. We employ an LLM based task evaluation strategy where we prompt *claude-instant-v1* to evaluate if two responses (sentence1 and sentence2) are semantically same or not. We conducted a manual audit as well and found a Cohen's Kappa score of 0.65 (96.66% agreement) between auditors and LLM generated evaluations establishing the effectiveness of our approach.

A.9 Digital Restaurant Service Platform Conversation Dataset

Each usecase has their own set of task execution procedure (TEP) steps in natural language, deterministic multi-step execution task and utility queries. Deterministic tasks (functions) are defined as JSONSchemas to the LLM prompt as input. A sample of TEP steps and a function JSONSchema is mentioned below:

Sample Function JSONSchema for Restaurant Menu Update:

```
{
    "name": "menu_price_update_task",
    "description": "update the price for
        a menu item of a restaurant",
    "parameters": {
      "type": "object",
      "properties": {
        "merchant_id": {
          "type": "string",
          "description": "Unique
              identifier for a merchant"
        },
        "restaurant_name": {
          "type": "string",
          "description": "name of the
              restaurant"
        },
        "current_price": {
          "type": "string",
          "description": "current price
              of the menu item"
        },
        "new_price": {
          "type": "number",
          "description": "new price to
              be updated for the menu
              item"
        },
        "item_name": {
          "type": "string",
          "description": "name of the
              menu item for which the
              price needs to be updated"
        }
      },
      "required": [
        "merchant_id",
        "restaurant_name",
        "current_price",
        "new_price",
        "item_name"
      ]
    }
}
```