Pontifícia Universidade Católica
do Rio de Janeiro

Polyana Bezerra da Costa

# An LLM-based Workflow for Ad Hoc Teamwork with Human-Robot Teams in Real World Scenarios

**Tese de Doutorado**

Thesis presented to the Programa de Pós–graduação em Informática, do Departamento de Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática.

Advisor: Prof. Sérgio Colcher

Rio de Janeiro
April 2025

Pontifícia Universidade Católica
do Rio de Janeiro

**Polyana Bezerra da Costa**

# An LLM-based Workflow for Ad Hoc Teamwork with Human-Robot Teams in Real World Scenarios

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática. Approved by the Examination Committee:

**Prof. Sérgio Colcher**
Advisor
Departamento de Informática – PUC-Rio

**Prof. Alberto Barbosa Raposo**
Departamento de Informática – PUC-Rio

**Prof. Edward Hermann Haeusler**
Departamento de Informática – PUC-Rio

**Prof. Júlio Cesar Duarte**
Instituto Militar de Engenharia

**Dr. Antonio José Grandson Busson**
BTG Pactual

Rio de Janeiro, April 25th, 2025

**Polyana Bezerra da Costa**

Holds a master's degree (2020) in Computational Modeling, and a bachelor's degree in Computer Science from the Federal University of Maranhão (UFMA).

Bibliographic data

Costa, Polyana Bezerra da

An LLM-based Workflow for Ad Hoc Teamwork with Human-Robot Teams in Real World Scenarios / Polyana Bezerra da Costa; advisor: Sérgio Colcher. – 2025.

75 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Informática – Teses. 2. Colaboração Ad Hoc. 3. Agentes Autônomos. 4. Modelos de Linguagem de Grande Escala. 5. Planejamento Automático. 6. Interação Humano-Computador. I. Colcher, Sérgio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, my niece Maju, my cat Cambota,
and my lover Alexandre

## Acknowledgments

I want to start by thanking God for giving me the strength to start, persist, and complete my PhD. Without His help and support, this work would not have been possible. I am deeply grateful to God for guiding me and for placing generous and inspiring people in my path who contributed greatly to this journey.

I thank my dear parents, Florise and José Carlos, for always believing in me and supporting my ideas (even the craziest ones, like pursuing a PhD in a city and at a university where I didn't know anyone). Thank you for your emotional and financial support, for your kind words, and for having so much faith in me that I felt I had no choice but to keep going, for you. I also thank my dear brother, João Pedro, who has always expressed how proud he is of me, and for giving us Maju, who constantly motivates me to be a better person and to set a good example. Thank you, Maju, for keeping me company, helping me relax during tense moments and for being an endless source of love and affection.

I would like to thank Professor Sérgio Colcher for his immense care and patience during these nearly five years of my PhD. Thank you for never giving up on me, even when I missed deadlines or was frustrated with my progress. Thank you for always treating your students with respect and for listening to our concerns with a genuine intent to help. I also thank Professor Júlio Duarte for his continuous support, care, and respect for students, always willing to review papers, discuss results, and offer valuable suggestions, all with elegance and grace.

To my lab mates, I offer special thanks to Pedro Thiago for his incredible help with experiments, his excellent suggestions, and generosity, we always knew we could count on you. I also thank José Boaro, who contributed with insights and suggestions to this work, and Paulo Vitor for keeping me updated on new models and techniques and for supporting my work. Thank you to Álvaro for taking good care of the students and for being attentive to our needs. I'm especially grateful to Daniel Moraes, who was not only my lab partner but also my housemate during part of the PhD. Thank you for your generosity, motivation, and for always believing we would make it, we did!

A warm thank you to my Rio family: Isabela, Matheus, Baguete, and Cuca! Thank you for everything! For welcoming me into your home so many times, for your affection and hospitality, for your unwavering encouragement, and for always believing in me. I also thank my beloved cats, Roxinho, Branquinho, and their mom, as well as my cat angels who are no longer with us: Cambota, Jujuca, Loirinha, and Amarelinha.

I'm also grateful to the professors and friends throughout my academic journey who motivated and inspired me to pursue a research career. Special thanks to Professor Anselmo for welcoming me, mentoring me, and teaching me most of what I know about research, and, most importantly, for being someone who inspires others with your generosity, authenticity, competence, and sense of humor. I also thank Professor Dallyson for his guidance and patience during my master's; Professors Geraldo and Simara for being sources of inspiration; my former boss, Márcio Moreno, for encouraging me to apply for a PhD; and Wallas, Sandro, and Rodrigo for being brilliant PhDs who motivated me to aim higher.

Finally, I thank my lover, friend, partner, and soon-to-be PhD, Alexandre. We've come a long way! When I met you ten years ago, I could never have imagined the path we would take together. You've stood by me during the most important moments of my life, and even when we were apart, you were a constant source of inspiration. I am so grateful to have you in my life, to share my dreams with you, and for our mutual support in our interests and careers. My love, thank you for being the best partner I could ask for. You inspire me every day with your competence, generosity, simplicity, and above all, kindness. Thank you for everything, it's hard to put into words. Thank you for always believing in me, for encouraging me to keep going, and to try again and again. You always said everything would work out, and it did. So here's a reference for you: Believe in victory!

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; and by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ), under grant number 141809/2020-5. In addition, this material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-22-1-0475.

# Abstract


Costa, Polyana Bezerra da; Colcher, Sérgio (Advisor). **An LLM-based Workflow for Ad Hoc Teamwork with Human-Robot Teams in Real World Scenarios**. Rio de Janeiro, 2025. 75p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As robotic platforms become more accessible, they are gradually being integrated into diverse real-world scenarios, requiring adaptive collaboration to support human activities effectively. Ad Hoc Teamwork (AHT) research addresses this challenge by enabling autonomous agents to collaborate dynamically without prior coordination, predefined rules, or rigid protocols. This requires the agent to infer tasks, plan actions, and adapt to dynamic environments in real-time, often with incomplete knowledge of the task or their teammates. While most existing AHT research has focused on agent collaboration in virtual environments or constrained human-robot interactions, real-world AHT presents additional challenges, such as unpredictable human behavior, implicit task definitions, and physical constraints. To operate effectively in such environments, ad hoc agents must have robust planning mechanisms and be highly adaptable, inferring tasks, recognizing collaboration opportunities, and making real-time decisions without imposing constraints on human teammates. In this context, this thesis explores the integration of Large Language Models (LLMs) into ad hoc agents to enhance their planning capabilities in real-world scenarios with humans. Given the vast knowledge embedded in LLMs and their strong generalization abilities, they may provide valuable insights that enhance the planning process for tasks the agent was not explicitly trained for. Unlike prior approaches that rely on predefined task libraries or environmental assumptions, we propose an LLM-based workflow that enables on-the-fly planning of tasks, allowing the agent to infer tasks, identify collaboration opportunities, and adapt to dynamic scenarios using contextual information such as verbal instructions and environmental cues. Our method combines task decomposition into sub-steps using Zero-Shot Chain-of-Thought prompting, the in-context learning capabilities of LLMs through an adapted method for retrieving meaningful examples, and self-refinement via LLMs to improve the quality of the final plan. Using this workflow, we achieved 90.20% similarity with reference plans for single tasks without plan refinements, and 82.20% for multi-tasks with refinements, an improvement of 2 percentage points over the previous best result for single tasks and 4 percentage points for multi-tasks. These results highlight the potential of using LLMs to generate feasible action plans for previously unseen tasks. By bridging the gap between virtual and real-world AHT research, this work contributes to the development of ad hoc agents capable of flexible collaboration with humans in unstructured environments. Our findings underscore the potential of using LLMs across multiple stages of the AHT pipeline, making our workflow suitable for deployment with human supervision.

## Keywords

Ad Hoc Teamwork; Autonomous Agents; Large Language Models; Automated Planning; Human-Computer Interaction.

## Resumo

Costa, Polyana Bezerra da; Colcher, Sérgio. **Uma Arquitetura baseada em LLMs para Colaborações Ad Hoc em Times com Humanos e Robôs em Ambientes do Mundo Real**. Rio de Janeiro, 2025. 75p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

À medida que plataformas robóticas se tornam mais acessíveis, seu uso em diversos cenários do mundo real tem se popularizado. Para que essas plaformas possam dar o devido suporte às atividades humanas, é desejável que a colaboração e interação com elas seja flexível e sem muitas regras. A área de pesquisa Ad Hoc Teamwork (AHT) aborda esse desafio ao explorar o desenvolvimento de agentes autônomos que colaborem dinamicamente sem coordenação prévia, regras predefinidas ou protocolos rígidos. Isso requer que o agente infira tarefas, planeje suas ações e se adapte a ambientes dinâmicos em tempo real, geralmente com conhecimento incompleto da tarefa atual ou das pessoas que estão no ambiente. Enquanto a maioria das pesquisas em AHT foca na colaboração agente-agente em ambientes virtuais ou em interações humano-robô em ambientes controlados, o AHT do mundo real apresenta vários desafios adicionais, como comportamentos imprevisíveis por partes dos humanos, tarefas sem definições claras e restrições físicas ao interagir no mundo real. Para atuar efetivamente em tais ambientes, agentes ad hoc devem ter mecanismos de planejamento robustos e ser altamente adaptáveis, inferindo tarefas, reconhecendo oportunidades de colaboração e tomando decisões em tempo real sem impor restrições aos humanos que estejam no ambiente. Neste contexto, esta tese explora a integração de *Large Language Models* (LLMs) em agentes ad hoc para aprimorar suas capacidades de planejamento e colaboração em cenários do mundo real com humanos. Por terem sido treinados num conjunto vasto de dados, as LLMs apresentam grande capacidade para generalização e conhecimento de senso-comum. Dessa forma, as LLMs podem fornecer *insights* valiosos que auxiliem o processo de planejamento de tarefas para as quais o agente não foi explicitamente treinado. Ao contrário de abordagens anteriores que dependem de bibliotecas de tarefas predefinidas ou pré suposições sobre o ambiente em que o agente está inserido, propomos um workflow baseado em LLMs que usa *Prompt Engineering* para melhorar a qualidade dos planos gerados. Nosso método combina a decomposição de tarefas em subetapas usando o método Zero-Shot Chain of Thought, as capacidades de aprendizagem contextual das LLMs, utilizando um método adaptado para buscar exemplos relevantes a serem incluídos no prompt, e auto refinamento por meio de LLMs para melhorar a qualidade do plano final. Com esse workflow, alcançamos 90.20% de similaridade com planos de referência para tarefas únicas sem aplicar refinamentos nos planos, e 82.20% para multitarefas, aplicando refinamentos nos planos, o que mostra um aumento de dois pontos percentuais em relação ao melhor resultado anterior para tarefas únicas e um aumento de 4 pontos percentuais para multitarefas. Esses resultados destacam o potencial das LLMs para gerar planos de ação viáveis para tarefas que o agente não foi especificamente treinado para fazer. Ao abordar o AHT em ambientes reais com humanos, este trabalho contribui para o desenvolvimento de agentes ad hoc capazes de colaboração flexível com humanos em ambientes não estruturados. Nossos resultados ressaltam o potencial de usar LLMs em vários estágios do pipeline AHT, tornando nosso workflow desejável para implantação com supervisão humana.

## Palavras-chave

Colaboração Ad Hoc; Agentes Autônomos; Modelos de Linguagem de Grande Escala; Planejamento Automático; Interação Humano-Computador.

# Table of contents

## List of figures

## List of tables

## List of algorithms

## List of Abreviations

AHT – Ad Hoc Teamwork

API — Application Programming Interface

CoT – Chain-of-Thought

GPT — Generative Pre-Trained Transformer

LEMMA – LEarning Multi-agent Multi-task Activities

LLaMA – Large Language Model Meta AI LLMs – Large Language Models

RAG — Retrieval Augmentation Generation

PDDL – Planning Domain Definition Language

ReAct – Reasoning and Act

ToT – Tree of Thoughts

*"Beauty lies neither in the morning light nor in the evening shadow; it is in the twilight, in that halftone, in that uncertainty."*

**Lygia Fagundes Telles**, *Os contos.*

# 1
# Introduction

As robotic platforms become more accessible, they are increasingly expanding beyond industrial applications to assist humans in diverse contexts, such as educational settings, healthcare, entertainment, household tasks, personal assistance, and retail (TASAKI et al., 2015; BELPAEME et al., 2018; PRATTICÒ; LAMBERTI, 2020; TRAINUM et al., 2023). According to a study on the future of robots, the presence of collaborative robots with humans will be even more prevalent in the upcoming years. It is estimated that 34% of all robots sold globally in 2025 will be collaborative robots (BORBONI et al., 2023).

Recent advancements in artificial intelligence have allowed collaborative robots to perform more complex tasks and adapt better to dynamic environments (BROHAN et al., 2023). As robots become more present in our daily activities, efficient and simple collaboration with humans is crucial to facilitate its adoption by general users. Besides, in many of the aforementioned scenarios, adaptive collaboration is not only a matter of making human-robot interaction more appealing but also a need, as some tasks inherently require flexibility and cannot rely on predefined protocols or rigid coordination. This type of on-the-fly colaboration is the focus of the Ad Hoc Teamwork (AHT) research area (STONE et al., 2010).

In AHT, a robot or virtual agent (called the ad hoc agent) interacts with teammates to achieve a common goal without prior coordination, quickly adapting to new situations and behaviors (RAVULA, 2019). The ad hoc agent enters an environment and helps its teammates (humans or not) without any pre-defined rules on how they will help each other. The agent has to make decisions, engage in activities, and cooperate without prior knowledge about the surrounding world, the teammates, or at times, even the task they are addressing (RIBEIRO et al., 2023). All inference, planning, and coordination must happen on the fly, in real time. In addition, the ad-hoc agent cannot impose constraints on its teammates (MIRSKY et al., 2022).

Most AHT research has focused on agent collaboration in virtual environments, such as in soccer, chess, and predator-prey games (BARRETT; STONE, 2015; BARRETT et al., 2017a; RIBEIRO et al., 2022). More recent studies have explored human-robot (or virtual agent) collaboration, but primarily in virtual settings like in the games Minecraft and Overcooked, where all possible actions and tasks are typically predefined (FOSONG et al., 2024).

These works often rely on prior information, such as a task library with action plans or predefined teammate types and their expected behaviors. Although the ad hoc agent might enter the environment without knowing the current task, it usually knows that the task belongs to its predefined library and uses prior experience to guide its behavior.

However, when collaborating with humans in real-world settings, such control over the environment is no longer guaranteed (RAHMAN et al., 2023). As noted by Ribeiro et al. (2024), humans do not always behave optimally or in a rational way. A human teammate may interrupt a task midway to start another, introduce unexpected actions, or deviate from predefined plans. Defining a set of tasks and designing action plans for human-robot collaboration in real-world environments can be time-consuming and restricting, as humans may easily deviate from the plan or perform tasks outside the agent's predefined scope. Additionally, the teammate may not always explicitly state the task, requiring the ad hoc agent to infer tasks and identify opportunities for collaboration. Real-world environments also introduce physical constraints and complex dynamics when interacting with and manipulating objects. As a result, AHT with human teammates is highly dynamic and often unpredictable, and to cooperate in such an environment, ad hoc agents must be highly adaptable.

The versatility and adaptability required in AHT environments call for mechanisms to enhance the ad hoc agent's planning abilities. In this context, Large Language Models (LLMs) could play a valuable role, since LLMs have been used in many scenarios where generalizations and adaptations are needed. Given the wide range of data on which these models are trained, LLMs embed such a vast knowledge that enables them to achieve high generalization capabilities and perform tasks in diverse contexts, including acting as planning agents, coordinating tasks, and real-time decision making (ZHAO; JIN; CHENG, 2023; ZHENG; WANG; JIA, 2023; XU et al., 2023; HUANG et al., 2023; ZHANG et al., 2024).

By leveraging these capabilities and their broad commonsense knowledge, LLMs could aid in ad hoc human-robot collaboration in real-world settings, helping eliminate the need for predefined task lists/action plans, or prior environmental assumptions.

### 1.0.1 Objectives

In this context, this work aims to investigate the use of LLMs into ad hoc agents to enhance their planning and cooperation capabilities with human teammates on the fly. We investigate current prompt techniques and language models available for planning/executing tasks, aiming to develop a workflow for ad hoc teamwork with humans in real-world scenarios without relying on predefined tasks, that is adaptive to changes, low-cost and that works on real-time. With these main goals defined, the following research questions guided the development of this work:

*R.Q 1 - How can we leverage the use of Large Language Models to build an ad hoc agent that can plan for tasks it was not trained for in an unknown real-world scenario?*

Our main hypothesis is that combining contextual cues alongside the common sense knowledge embedded in LLMs enables an ad hoc agent to identify tasks, assess collaboration opportunities, understand task progression, and generate feasible action plans. Given the vast knowledge encoded in LLMs, they may provide valuable insights that enhance the planning process for tasks the agent was not explicitly trained for. Based on this assumption, to assess the performance of LLMs and their feasibility of use in real-world scenarios, we ask the second research question:

*R.Q 2 - How can we ensure that an ad hoc agent using LLMs for real-time task planning is reliable and cost-effective?*

To assess the feasibility of using LLMs in real-time ad hoc teamwork, we explore strategies that balance reliability, response time, and cost. By experimenting with different prompting techniques, we can identify which ones lead to the most accurate and consistent plans. We also investigate ways to optimize model usage, such as focusing on open-source alternatives to mitigate rate-limiting issues of commercial APIs and reduce costs. Prompt engineering, along with self-refinement mechanisms can help improve the performance of open-source models, making them viable options for task planning.

In this context, we propose an LLM-based workflow for on-the-fly planning of tasks involving humans in real-world environments. Unlike prior work, our workflow is not limited by predefined task libraries. Instead, it uses the agent's capabilities and contextual information (such as teammates' actions, verbal instructions, objects present in the environment, etc.) to identify opportunities for collaboration and plan for tasks in real-time. With this work, we want to bridge the existing gap in AHT research by focusing on mixed human-robot teams for tasks in the real world. We address tasks where the agent has to interact with both humans and physical objects. Since we do not have an actual robot to execute these tasks, the execution is simulated. The agent receives a task instruction and autonomously selects the appropriate actions from its pool of capabilities.

We will evaluate the suitability of LLMs as planning tools for ad hoc environments, assessing performance, cost, response time, and exploring how their vast knowledge enhances adaptability in dynamic settings. This research is part of the project *Trustworthy Ad Hoc Teamwork* funded by the Air Force Office of Scientific Research under award number FA9550-22-1-0475. Two previous research projects were developed under this theme: a middleware of core services for robotic platforms that provides contextual information about the environment and a SLAM-based localization system for robots that builds a semantic map of the environment.

### 1.0.2 Outline

The remainder of this thesis is organized as follows: Chapter 2 presents the related work in Ad Hoc Teamwork and LLMs for Planning, highlighting possible research gaps and opportunities for contributions with this work. Chapter 3 presents important concepts for understanding the developed method, while Chapter 4 presents our proposed workflow for building an agent for ad hoc contexts. Chapter 5 describes our experiments and a detailed analysis of the results. Finally, in Chapter 6 we present our conclusions, answer the research questions, and present ideas for future work.

# 2
# Related Work

This chapter presents important work in the main research fields addressed in this thesis, highlighting research gaps and potential contributions. The first section covers the main approaches to Ad Hoc Teamwork, considering both virtual agent collaboration and human-robot collaboration. The following section presents the most recent work on using Large Language Models for planning tasks for robots, virtual agents, and agent collaboration. We conclude each section of this chapter with a comparative table between these works and ours.

The first part of our related work covers the state of the art in Ad Hoc Teamwork. To find relevant works in this area, we searched the databases Scopus, ScienceDirect, IEEE, Springer, ACM Digital Library, and arXiv. The search string used to retrieve these works was *"ad hoc teamwork" OR "zero-shot coordination" OR ("human-machine teaming" AND "zero-shot coordination").*

The following part of our related work shows how LLMs have been applied to planning tasks. For this section, we consulted the GitHub repository for Reasoning and Planning with LLMs[1], which is frequently updated with relevant works. We also used two search strings to retrieve additional works: *(("Large Language Model" OR "Large Language Models" OR "LLM" OR "LLMs") AND ("MULTI-TASK" OR "MULTITASKING" OR "MULTI-TASKS" OR "MULTI TASK" OR "MULTI TASKS") AND "PLANNING")* and *("LLM" and "PDDL").*

### 2.0.1
### Ad Hoc Teamwork

To manage the uncertainty and adaptability required in ad hoc environments, most work in Ad Hoc Teamwork incorporates some sort of prior information. Some approaches model a set of possible tasks or teammate types, while others store past interactions for comparison with new ones.

The PLASTIC Model, a classical AHT approach, models both the environmental dynamics and teammate behaviors (BARRETT et al., 2017a). PLASTIC has a library of previous teammates' behaviors and corresponding action policies for the ad hoc agent. This model assumes that the current teammate's behavior can be similar to past ones. At runtime, the ad hoc agent compares its teammate's actions to those in its library to select an appropriate

[1]https://github.com/GT-RIPL/Awesome-LLM-Robotics

policy. However, this assumption may not always hold. The authors tested PLASTIC on two benchmark: the pursuit domain and robot soccer in the RoboCup 2D simulation domain.

Another approach is to model all possible tasks in a given context. Although the ad hoc agent may not know the task performed at the moment, in these works it usually has access to a library of all possible tasks (RIBEIRO et al., 2021). In Melo e Sardinha (2016), the authors define the set of possible tasks as matrices of actions, and the AHT problem is treated as a partially observable Markov decision problem. In this context, the agent must observe the teammates' behavior to identify the task. The disadvantages of this approach are having to characterize each task in advance and expecting the teammates to behave accordingly and perform only tasks specified in the library.

So far, most of these works discuss ad hoc teamwork collaboration as a theoretical problem under relatively ideal conditions in 2D environments. As noted in Ribeiro et al. (2021), expanding ad hoc teamwork to real-world scenarios presents significant challenges. These include the ad hoc agent's lack of prior knowledge about the tasks, limited perception capabilities to understand teammates' actions, and the absence of explicit or implicit reward signals during interaction. Additionally, dealing with mixed teams of humans and robots can be even more difficult, as human teammates may not behave rationally or optimally, or follow a model of actions. To top all that, the processing and decision-making must happen in real time.

In the context of human-robot AHT, Ribeiro et al. (2024) introduced the first framework for real-world human-robot collaboration. Their system comprises two modules: (1) a decision module, responsible for identifying the task, recognizing teammates, and planning actions, and (2) a communication module, which employs NLP techniques to parse instructions from human teammates and determine the task's current state. The robot, equipped with a task library, uses this prior knowledge to infer the human's actions and provide assistance. The framework was tested on a single task in an open space.

More recent approaches to AHT are using Large Language Models. For example, Liu et al. (2024) used LLMs for ad hoc heterogeneous multi-robot collaboration. In this research, the authors used LLMs for coordinating robots with different capabilities where the ad hoc robot improves the efficiency of the whole team and does not interfere with the actions of the other robots. The authors proposed a hierarchical dynamic planning model called IRoT, where an LLM interprets a given task plan, decomposes it into sub-tasks, and assigns them based on each robot's capabilities, predicting the most suitable sub-task at any given moment.

Similarly, (DODAMPEGAMA; SRIDHARAN, 2025) combined knowledge-based reasoning, LLMs, and data-driven learning for ad hoc teamwork. Their architecture uses LLMs to anticipate future tasks and goals of the human teammates. Based on these predictions, it computes an action plan using non-monotonic logical reasoning. The system also incorporates commonsense domain knowledge to get a basic understanding of each task. The LLM has access to a set of possible tasks to choose from, and examples of sequences of tasks previously performed by the human. The architecture capabilities were evaluated in VirtualHome, a realistic 3D simulation environment.

Table 2.1 summarizes the main works in Ad Hoc Teamwork (AHT) and their characteristics, highlighting how it compares to our proposed method. Most approaches to AHT often rely on explicitly modeling possible tasks and teammate behaviors. In contrast, our approach uses an LLM to autonomously analyze the context and plan how the ad hoc agent can collaborate based on its capabilities. This allows the agent to dynamically respond to evolving scenarios without relying on predefined tasks or behavior specifications.

### 2.0.2
### LLMs for Planning Tasks

With the recent breakthroughs in the LLMs field, they have been applied across diverse contexts, including planning for robotics. Works with earlier language models such as BERT already catered to robotic-planning-related problems (DEVLIN et al., 2019). In this scenario, FILM presented a modular method based on BERT that processes language instructions and organizes visual input into a semantic map to fulfill a given task, outputting a series of navigation commands and actions as a plan (MIN et al., 2021).

Similarly, in LangSuitE, the authors built a semantic mapping module to capture what is happening in the environment (ZHENG; WANG; JIA, 2023). LangSuitE specifies the task to be executed, characterizes the environment, and defines the robot's actuation all in the same prompt. They also included a task example and how it should be performed in the prompt for few-shot learning. The authors used OpenAI's GPT 3.5[2] for controlling and planning the robot's actions and for communication between agents.

Recently, more robust models for solving robotic problems were introduced, such as RT-2, a visual-language-action model (BROHAN et al., 2023). RT-2 is an end-to-end model based on PALM-2 that maps robot observations

[2]https://platform.openai.com/docs/models/gpt-3-5

Table 2.1: Comparison of Existing Approaches to Ad Hoc Teamwork in the Literature

| Work | Technique | Environment | Prior Info |
|---|---|---|---|
| (BARRETT et al., 2017a) | Models environmental dynamics and teammate behaviors | Virtual | X (Library of tasks and teammate behaviors) |
| (MELO; SARDINHA, 2016) | Models AHT as a partially observable MDP | Virtual | X (Library of tasks) |
| (RIBEIRO et al., 2024) | Framework with a decision and a communication module | Real World | X (Reference task plan) |
| (LIU et al., 2024) | LLMs for ad hoc heterogeneous multi-robot collaboration | Virtual | X (Library of tasks) |
| (DODAMPEGAMA et al., 2025) | Combined knowledge-based reasoning, LLMs, and data-driven learning for AHT | Virtual | X (Library of tasks) |
| Ours | LLMs: task decomposition, in-context learning, action plan generation and plan refinements | Real World (simulated) | No |

to actions using textual and visual data. RT-2 is able to receive a command in text, extract information from image data, reason over this information, and output generalized instructions for robotic control, such as vectors for positional and rotational change.

Also combining different types of information, integrated texts, sensor data, prompts, LangChain's Agents, and LLMs to model the context in which the robot is inserted to plan its actions (XIONG et al., 2023). The authors define categories of requests and examples of question-answering for each category. At first, the LMM has to classify the user's request based on previously defined categories using the right prompt template. With the final prompt, the LLM outputs a sequence of actions in natural language that fulfill the user's request.

Along with planning the robot's action, some works use LLM to generate actual code to execute those actions. RoboTool is a system geared towards robotic arms and mobile robots that uses LLMs to map commands in natural language to executable code (XU et al., 2023) . Similarly, Instruct2Act also uses an LLM to generate Python code for robotic manipulation, distinguishing itself by receiving multimodal instructions (HUANG et al., 2023). The system comprises a perception module that uses foundation models to locate objects in a scene, classify and segment them, and an action module that interprets a given command along with the perception information. Then, it generates actions that are transformed into executable code.

Vemprala et al. (2023) applied OpenAi's ChatGPT[3] to a pipeline for solving robotic tasks, also outputting Python code. At first, the authors define a basic Python API with the robot's actions, the function's names, and their parameters, but no actual code within these functions. Then, in the prompt, the authors provide a clear description of the task and the functions defined in the API (and their descriptions). ChatGPT is then asked to output a Python code that can fulfill the specified task. The authors tested this pipeline in ten different scenarios that involved logical, geometrical, and mathematical reasoning, navigation, and manipulation, among others. With extensive tests, the authors showcased the potential of ChatGPT for robotic tasks and how it can be able to reason and plan in the most different contexts.

Regarding planning for collaborative tasks, Singh, Traum e Thomason (2024) used LLMs to manage multi-agent planning, combining the strengths of classical planning (PDDL) with LLMs for two-agent planning goal decomposition. Based on a task plan defined in PDDL, the authors used an LLM to decompose a global goal into independent and non-concurrent sub-goals for each agent, in a way that the two agents in the domain can execute them simultaneously. In this context, the agents have the same capabilities. The authors noted that LLM-based goal decomposition led to faster planning times with fewer execution steps.

Similarly, Zhang et al. (2024) developed a novel framework that uses LLMs to enhance cooperation between agents and to dynamically adapt to teammates. Before the cooperative task begins, the LLM accesses a library of tasks and transforms raw tensor state information into language-based state descriptions. Based on these prior observations, the LLM analyzes the current scene to infer the teammate's intention and selects the most suitable action. Experiments within the Overcooked-AI environment demonstrated ProAgent's superior performance in comparison to traditional methods.

In a work that is the most similar to this one, Wang et al. (2023) uses LLMs for multi-task planning for long-term tasks in open-world environments

[3]https://openai.com/blog/chatgpt/

(tested on Minecraft). In this work, the authors tackled several challenges in task planning, such as planning for tasks with long sequences of subgoals and handling tasks in open worlds with different object types and complex dependencies. Given a task and image observations, the authors used an LLM to generate an initial plan, along with a description of the plan (also generated by an LLM), executed the plan on the virtual environment, and used another LLM to provide feedback if the plan failed. To refine the initial plan, the authors trained a neural network that ranks parallel sub-goals of a task based on the estimated steps of completion, which guarantees that the agent will choose optimal actions. Training this neural net was possible because of the amount of knowledge available about the Minecraft game. Applying this optimal-action selector might not be as easy in real-world environments with unseen tasks.

Table 2.2 presents an overview of recent works that use LLMs in different stages of the planning process, highlighting their key characteristics in comparison to our approach. Similarly to these works, we use LLMs to process diverse contextual information and generate actionable plans for robots or virtual agents. However, unlike most AHT approaches, our agent has no explicit exposure to tasks and relies entirely on real-time contextual understanding and the commonsense knowledge of LLMs. We eliminate the use of predefined task libraries, enabling our ad hoc agent to operate in a zero-shot manner, using contextual cues such as human actions, environmental information, and verbal instructions to dynamically plan and adapt. Additionally, while most prior work focuses on controlled virtual environments with predefined actions and interactions, we shift toward real-world, human-robot collaboration, where the agent must handle the unpredictability of human behavior, physical constraints, and unscripted interactions.

Table 2.2: Comparison of Existing Approaches that use LLMs for Planning

| Work | Technique | Environment | Prior Info |
|---|---|---|---|
| (SINGH; TRAUM; THOMASON, 2024) | LLMs + classical planning (PDDL) | Virtual | X (Library of tasks defined in PDDL) |
| (ZHENG; WANG; JIA, 2023) | Uses LLMs to transform raw tensor state information into language-based state descriptions | Virtual | X (Library of tasks) |
| (WANG et al., 2023) | LLMs for multi-task planning for long-term tasks (generate an initial plan and provide feedbacks) + NN that ranks sub-goals for selecting optimal actions (plan refinement) | Virtual | X (Sub-goals of a given task) |
| Ours | LLMs: task decomposition, in-context learning, action plan generation and plan refinements | Real World (simulated) | No |

# 3
# Theoretical Background

This chapter presents the theoretical background needed for understanding the main problem addressed in this work and the techniques that support the developed workflow. It begins by defining what is Ad Hoc Teamwork, and what are the main approaches to it. Next, it introduces Large Language Models and their applications in planning. Then, it discusses the concept of Prompt Engineering and the main prompting strategies that were applied and tested in this work.

## 3.1
## Ad Hoc Teamwork

Ad Hoc Teamwork (AHT) is a research area introduced by Stone et al. (2010) to address the challenges of collaboration without prior coordination. In AHT, agents cooperate to achieve a common goal without previous opportunities to establish how they will collaborate or what each team member will do (GRUDA; MCCLESKEY, 2022). An ad hoc agent should be able to adapt to the context of the task dynamically, responding to changes on the fly (STONE et al., 2013). All inference, planning, and coordination must happen at runtime, with the ad hoc agent adjusting its behavior based on what it observes in the moment.

According to Mirsky et al. (2022), three main assumptions characterize the AHT problem. The first one is the lack of prior coordination. The agent is expected to cooperate when the task begins without any prior opportunities to establish or specify mechanisms for coordination. Secondly, the ad hoc agent has no control over teammates. The agent has to reason and act under the given conditions and cannot impose its knowledge on the team. Lastly, all agents have a common goal and must collaborate towards it.

Along with these characteristics, there are four main subtasks used to tackle AHT, as also described by (MIRSKY et al., 2022). The first task, *Knowledge Representation*, requires a representation of the domain knowledge. This includes knowledge about the environment, the agent's capabilities, and potential teammates. The representation of domain knowledge strongly influences the solution approach used in the other subtasks. This information can be obtained from human experts, prior knowledge of past teammates, through self-play, and/or learning from data.

The second task, *Modeling Teammates*, covers the part of modeling the teammates' behavior (if available), such as classifying teammates by predefined types to aid in the adaptation process and improve decision-making. Next, *Action Selection* includes planning methods and expert policies that are learned or based on expert knowledge. Finally, the fourth task, *Adapting to Changes*, is responsible for gathering new information from the environment, teammates, or tasks obtained during the interaction process. The agent must adapt its behavior to enhance collaboration based on this additional information.

So far, research in ad hoc teamwork has focused primarily on virtual agent interaction scenarios or human-agent interactions in virtual settings (BARRETT et al., 2017b). Dealing with mixed teams of humans and robots in real-world scenarios brings new challenges. A robot's perception can be incomplete at times. Robots have to deal with perceptual and actuation challenges, so the robot's perception of its state is often imperfect (RIBEIRO et al., 2024). Human teammates may not behave rationally or optimally or follow a predictable model of actions. Besides, without prior or complete knowledge about tasks, it is difficult to define reward mechanisms that can tell the agent if it is on the right path. However, collaboration with humans can have its advantages. As humans can communicate through natural language, such communication channels can be rich and informative if the agent can take advantage of them. Although communication between agents might be seen as a form of coordination, its use is not condemned, and the ad hoc agent can leverage that to increase cooperation.

In this thesis, we focus on two of the subtasks to achieve AHT: *Knowledge Representation*, by gathering contextual cues about the surroundings of the agent and using the common sense knowledge embedded in LLMs; and *Action Selection*, by exploring prompt strategies to improve on the fly task plan generation.

## 3.2 Large Language Models

Language Models (LMs) are computational models designed to understand and generate human-like text (RASCHKA, 2024). They capture the statistical patterns of human language, which allows them to perform several Natural Language Processing (NLP) tasks. Recent breakthroughs in Deep Learning, particularly with the advent of Transformer models, have allowed for significant advancements in LMs (VASWANI et al., 2017).

Large Language Models (LLMs) are a type of LM characterized by their massive scale. These models are trained on vast amounts of textual data and contain billions of parameters. LLMs are typically pre-trained in an unsupervised fashion on large-scale text corpora, learning contextual representations of language and demonstrating remarkable generalization capabilities. As their scale grows, LLMs show significant performance improvements and abilities that are not present in smaller language models, such as in-context learning and cross-domain applicability without fine-tuning (ZHAO et al., 2023).

The versatility of LLMs has led to their extensive adoption in various NLP applications. One prominent use of these models is text generation, where LLMs produce coherent outputs for tasks like summarization, dialogue generation, machine translation, and content creation (BORGES et al., 2024). Models such as OpenAI's GPT and Meta's LLaMA are also applied to traditional NLP tasks, including sentiment analysis and named entity recognition (TOUVRON et al., 2023). Additionally, LLMs have also been used for improving search engines, information retrieval systems, and recommendation algorithms by enhancing the understanding of user queries and document relevance.

## 3.3 Planning with Large Language Models

Since LLMs have been applied across various contexts with great success, they have also been tested in several steps of the planning process. In a survey on the use of LLMs for planning tasks, Huang et al. (2024) noted that LLMs have shown significant performance in reasoning tasks, tool usage, plan generation, and instruction-following. As planning requires advanced decision-making, reasoning abilities, and understanding of the surrounding context in which the agent operates, LLMs can be used for improving planning abilities in agents.

For the authors, planning with LLMs can be defined as generating a sequence of actions, which can be formalized as follows: at time step $t$, given environment denoted as $E$, the action space as $A$, the task goal as $g$, the action at step $t$ as $a_t \in A$, the parameters of the LLM as $\Theta$, and the prompt for the task as $\mathcal{P}$, an LLM-generated plan is the sequence of actions $p$:

$$p = \text{plan}(E, g; \Theta, \mathcal{P});$$

$$p = (a_0, a_1, \cdots, a_t)$$

The authors also categorized the use of LLMs for planning into four main approaches. First, LLMs can be used for *Task Decomposition*, where the model decomposes a complex task into sub-tasks, applying a divide-and-conquer paradigm and planning for each sub-task individually. This method is extremely useful for real-world scenarios, as real-life tasks are typically multi-step. Another approach is to use LLMs for *Multi-plan Selection*, where the model generates several alternative plans or options for the same task, and a search algorithm selects the most suitable one. LLMs have also been used for plan *Reflection and Refinement*, improving plan quality by identifying potential failures, suggesting refinements, and providing feedback. Lastly, there is *Memory-augmented Planning*, where relevant information is stored in a memory module that the LLM can access during planning. Past experiences, common sense knowledge, and domain-specific knowledge can be stored in memory. In the workflow presented in this thesis, we use LLMs for task decomposition and plan refinement.

## 3.4
## Prompt Engineering

Prompt Engineering is the process of curating and optimizing prompts to improve the performance of LLMs, leading them into generating the desired response (GIRAY, 2023). This is one of the key techniques used to enhance the performance and adaptability of Large Language Models (LLMs) in specific tasks or domains. It can help shape the behavior and output of LLMs to better suit particular applications .

Researchers have noted that the choice of prompt strongly influences the outputs of advanced LLMs, where even minor input changes can lead to different responses (BROWN et al., 2020). This observation has motivated the study and development of prompting techniques to use LLMs more effectively and maximize their potential across diverse domains. By developing specialized prompts, we can improve model performance, mitigate hallucinations, reduce computational costs, and ensure that LLMs behave safely.

Prompt engineering involves optimizing instruction formats, selecting appropriate structures, and choosing the right keywords to better guide models toward desired outputs (BORGES et al., 2024). The effectiveness of prompt engineering has been demonstrated across several applications, including question-answering, summarization, and dialogue generation. The main prompting techniques used in our experiments are presented in the following sections.

### 3.4.1
### Zero-Shot and Few-Shot Prompting

While traditional machine learning models require retraining for new tasks, sufficiently large language models can adapt to unseen tasks from

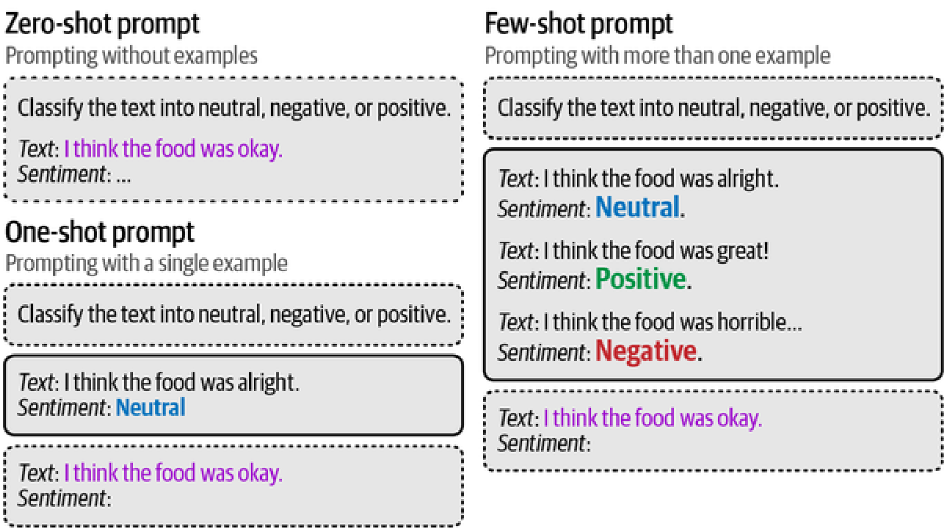



Figure 3.1: Zero-shot and few-shot prompting examples. Source:(ALAMMAR; GROOTENDORST, 2024)

examples or instructions provided in the prompt, not requiring weight updates (LIU et al., 2023b; LIU et al., 2023a). This is a form of continual learning called In-Context Learning (HUYEN, 2025). As shown by Brown et al. (2020), as the scale of language models grows, their task-agnostic performance greatly improves, showing zero and few-shot capabilities.

Zero-shot prompting allows LLMs to perform tasks without specific training, making them highly adaptable and efficient. In this approach, the model is given a natural language description of a task without any examples or additional guidance, relying solely on its pre-trained knowledge to generate responses. While this method highlights LLMs' generalization abilities, zero-shot prompting may struggle with complex tasks due to incomplete context understanding (BORGES et al., 2024). Few-shot prompting addresses this limitation by providing examples, where each example constitutes a "shot." Common variations of this method include one-shot (one example) and five-shot (five examples) prompting, as illustrated in Figure 3.1.

With few-shot prompting, we can provide the model with a few input–output examples to illustrate the task we want to perform. This has been successful for a wide range of tasks and tends to improve the performance of LLMs, providing more accurate responses. The main advantage of few-shot prompting is not needing extensive task-specific data and allowing the same model to perform different tasks (BROWN et al., 2020).

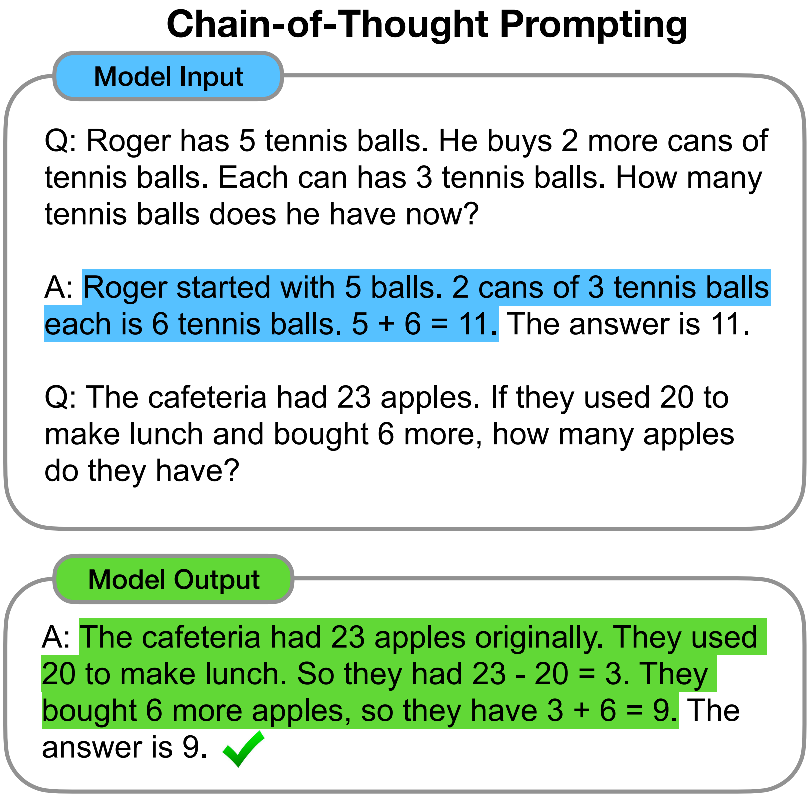



Figure 3.2: Chain-of-Thought prompting example. Source:(WEI et al., 2022)

### 3.4.2 Chain of Thought

Taking inspiration from the way humans solve complex problems by breaking them down into smaller steps, Wei et al. (2022) introduced Chain of Thought (CoT) prompting, a technique that encourages LLM to break problems into intermediate steps. Instead of directly answering a question, the model has to explain its "thought" process before giving the final answer.

The core idea behind CoT is to augment the prompt with input/output pairs that contain a chain of thought, a sequence of natural language steps that guides the model to the final answer. This is a form of few-shot learning, known as few-shot CoT. This approach contrasts with standard prompting, where the model is expected to generate the final answer directly without any intermediate reasoning. Figure 3.2 shows an example of how to provide those chains of thoughts on the prompt.

Another popular way to CoT prompting is to simply include the phrase "think step by step" in the input. This is a form of zero-shot CoT proposed by Kojima et al. (2022). The authors noticed that by including the keywords "Let's think step by step", LLMs outperformed zero-shot approaches on diverse benchmarks, including reasoning and logical tasks.

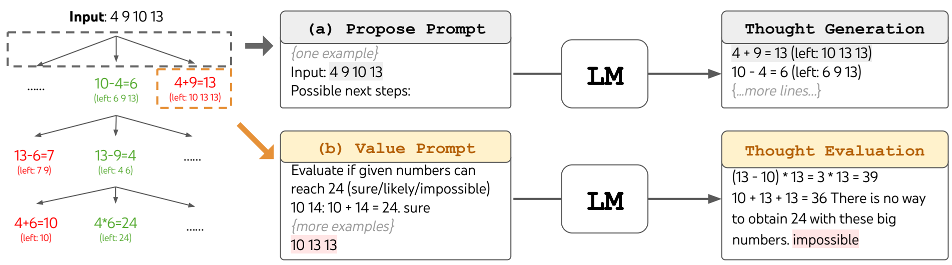



Figure 3.3: Tree-of-Thoughts framework. Source:(YAO et al., 2023)

Besides allowing LLMs to perform more complex tasks, the task decomposition abilities of CoT increases explainability, by providing a look into the model's decision-making process. This makes it easier to understand how the model arrived at a particular answer and where it went wrong. This makes CoT a powerful tool for enhancing the reasoning capabilities of LLMs across various domains.

### 3.4.3
### Tree of Thoughts

Proposed by Yao et al. (2023), Tree of Thoughts (ToT) is a framework designed to enhance the problem-solving capabilities of large language models (LLMs) by enabling them to explore multiple reasoning paths during inference. Similar to CoT, ToT allows LLMs to consider multiple intermediate steps (thoughts), that lead to a solution. This approach also took inspiration from human problem-solving strategies, where we explore various options, evaluate their potential results, and change paths if needed.

The ToT framework generalizes CoT by introducing a tree-like structure where each node represents a partial solution, and the branches correspond to possible next steps. This structure allows the model to explore different reasoning paths, evaluate their outcomes, and then decide which path to follow. The framework involves four key components: thought decomposition, where the problem is broken down into intermediate steps; thought generation, where the model proposes potential next steps; state evaluation, where the model assesses the progress of each thought; and search algorithms, such as breadth-first search (BFS) or depth-first search (DFS), which guide the exploration of the tree. By combining these components, ToT enables LLMs to perform more strategic problem-solving, particularly in tasks that require planning, exploration, or backtracking. Figure 3.3 depicts the classical ToT process.

Inspired by the ToT framework, Hulbert (2023) proposed Tree-of-Thoughts (ToT) Prompting, a technique that encourages the LLM to explore

```
Imagine three different experts are answering this
question.
All experts will write down 1 step of their thinking,
then share it with the group.
Then, all experts will go on to the next step, etc.
If any expert realizes they're wrong at any point,
then they leave.

The question is...
```

Figure 3.4: Tree-of-Thoughts prompting template. Source:(HULBERT, 2023)

and evaluate multiple possibilities and paths before concluding a task. In this technique, the prompt leads the model to elaborate different answers, allowing for a broader exploration of solutions (BORGES et al., 2024). This is particularly useful for contexts where there is more than one correct answer. Figure 3.4 shows the ToT prompting template.

### 3.4.4 Self-Refine

A crucial part of planning with LLMs is refinement. Although LLMs have performed well in the most diverse tasks and contexts, they can still hallucinate, especially in the context of planning, where tasks can be more complex (HUANG et al., 2024). A reflecting and refinement module can improve error correction capabilities of LLMs, helping it identify failures and learn from it, leading to more accurate responses.

One of the simplest forms of LLM-based refinement is Self-refine, an approach for improving initial outputs from LLMs through iterative feedback proposed by Madaan et al. (2023). The authors suggested an iterative process to improve and correct LLMs' responses. Self-refine uses the same model to generate an initial output, to give feedback on its own output, and to refine it. Figure 3.5 shows some use-cases of Self-refinement.

Using a single LLM, Self-Refine generates an initial response based on the user's request; then, the model analyzes its response and provides feedback. This feedback is then used by the same model to refine its initial response and correct possible errors. This process is repeated until a stopping criteria is met. After each generation, feedback for the previous response facilitates adjustments for the next generation. The authors evaluated the Self-Refine method on 7 different tasks and noticed a 20% improvement in performance on each task. The outputs generated with SELF-REFINE performed better

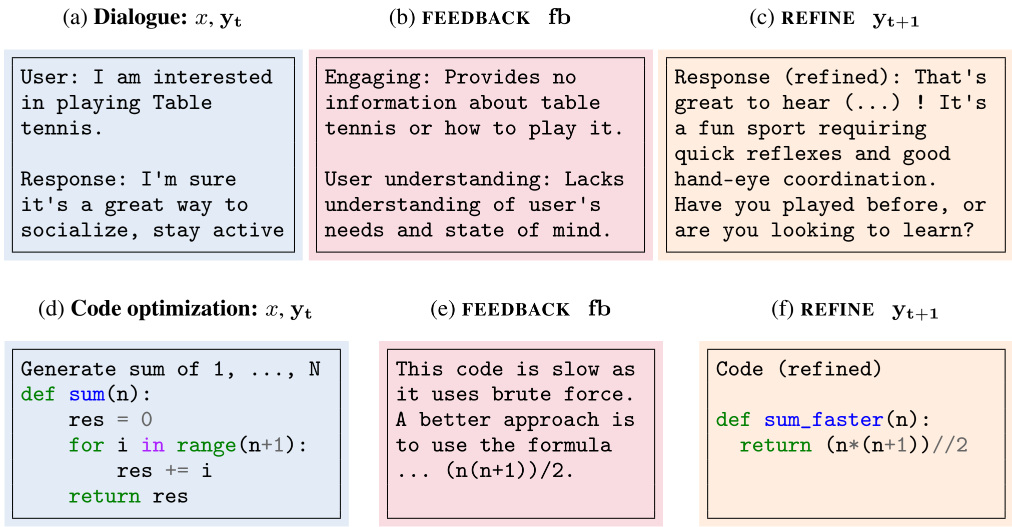



Figure 3.5: Example of a process of self-refinement with LLMs. The blue boxes represent the initial response of the LLM, the pink boxes show feedback for refinement, and the yellow boxes show the improved response based on feedback. Source:(MADAAN et al., 2023)

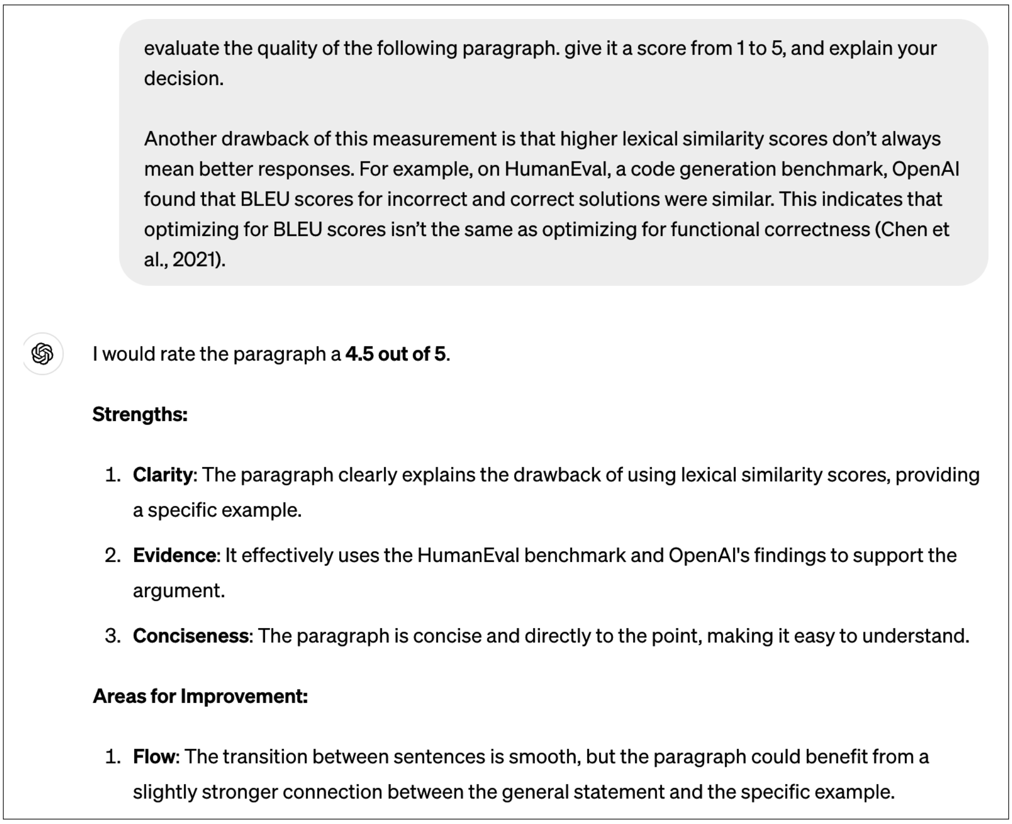
than the same LLM using conventional one-step generation. In our workflow, we used self-refining to correct possible errors in the LLM-generated plans.

### 3.4.5 LLM as a Judge

Another context in which LLMs have been extensively applied is in the evaluation of other AI systems. LLMs can evaluate, rank, and provide feedback on responses generated by other models, an approach known as LLM as a Judge or AI Judge (ZHENG et al., 2023). This has become one of the most popular methods for evaluating AI systems, as human evaluations are often costly and impractical in many scenarios (HUYEN, 2025). LLM judges offer a cost-effective, fast, and scalable alternative to human expert evaluation. Figure 3.6 shows an LLM Judge in action.

One of the main advantages of LLM judges is their explainability since they can evaluate responses and justify their decisions. As noted by Huyen (2025), this makes them particularly valuable for various applications and, in some cases, the only viable automated solution.

Although LMs have memorized some world knowledge during training they still tend to hallucinate and create imaginary content (Maynez et al., 2020; Zhou et al., 2021). Augmenting LMs with retrieval components that look up relevant information from external knowledge resources is a promising direction to address hallucination


evaluate the quality of the following paragraph. give it a score from 1 to 5, and explain your decision.

Another drawback of this measurement is that higher lexical similarity scores don't always mean better responses. For example, on HumanEval, a code generation benchmark, OpenAI found that BLEU scores for incorrect and correct solutions were similar. This indicates that optimizing for BLEU scores isn't the same as optimizing for functional correctness (Chen et al., 2021).

I would rate the paragraph a **4.5 out of 5**.

**Strengths:**

1. **Clarity**: The paragraph clearly explains the drawback of using lexical similarity scores, providing a specific example.
2. **Evidence**: It effectively uses the HumanEval benchmark and OpenAI's findings to support the argument.
3. **Conciseness**: The paragraph is concise and directly to the point, making it easy to understand.

**Areas for Improvement:**

1. **Flow**: The transition between sentences is smooth, but the paragraph could benefit from a slightly stronger connection between the general statement and the specific example.

Figure 3.6: Example of an LLM being used to rate a response. Source:(HUYEN, 2025)

Retrieval augmented LMs commonly use a retrieve-and-generate setup where they retrieve documents based on the user's input, and then generate a complete answer conditioning on the retrieved documents

These single-time retrieval augmented LMs outperform purely parametric LMs, particularly for shortform knowledge-intensive generation tasks such as factoid question answering (QA) (Kwiatkowski et al., 2019; Joshi et al., 2017), where the information needs are clear in the user's input, and it is sufficient to retrieve relevant knowledge once solely based on the input.

Retrieval-Augmented Generation (RAG) refers to the retrieval of relevant information from external knowledge bases before answering questions with LLMs. RAG has been demonstrated to significantly enhance answer accuracy, reduce model hallucination, particularly for knowledgeintensive tasks. By citing sources, users can verify the accuracy of answers and increase trust in model outputs. It also facilitates knowledge updates and the introduction of domain-specific knowledge

The LLM Judge can be used to make evaluations based on any criteria. They can be used as classifiers, categorizing a response according to possible labels (good, neutral, bad). They can be used as a scoring system, generating discrete or continuous scores based on a given range, evaluating degrees of similarity, etc. LLM judges can also evaluate responses based on the occurrence of hallucinations, the tone of the response, or any other criteria that the user may imagine. In addition to those contexts, Zheng et al. (2023) categorizes LLM judges in three main uses: for *pairwise comparison*, where the judge receives two responses and has to pick one; for *single answer grading*, when the judge has to assign a score to a single answer; and for *reference-guided grading*, where the judge has reference data and has to compare or grade the LLM response with the reference. In our experiments, we used an LLM Judge for reference-guided grading, comparing the LLM-generated plans against reference plans.

# 4
# A Workflow for Ad Hoc Teamwork with Human-Robot Teams

This chapter presents our LLM-based workflow for an ad hoc agent to plan tasks on the fly, detailing each part of the process. Our proposed method relies on task decomposition, in-context examples, and plan refinements rather than depending on predefined task plans. The user specifies the task, and the agent plans for it in real time. Our hypothesis is that by combining the agent's capabilities with the environment's current state, the user's request, and prompt engineering, LLMs can generate feasible plans for tasks in the real world.

We begin by defining the ad hoc agent's capabilities and acquiring contextual information about its operating environment. This contextual information, along with the task to be performed, is passed to our Contextual Planner, which generates an initial plan by decomposing the task into sub-goals. Based on this outline, the planner searches for and retrieves the most representative examples to augment the prompt and produces a final plan. Another LLM then reflects on the generated plan, refining it where needed. Figure 4.1 depicts this

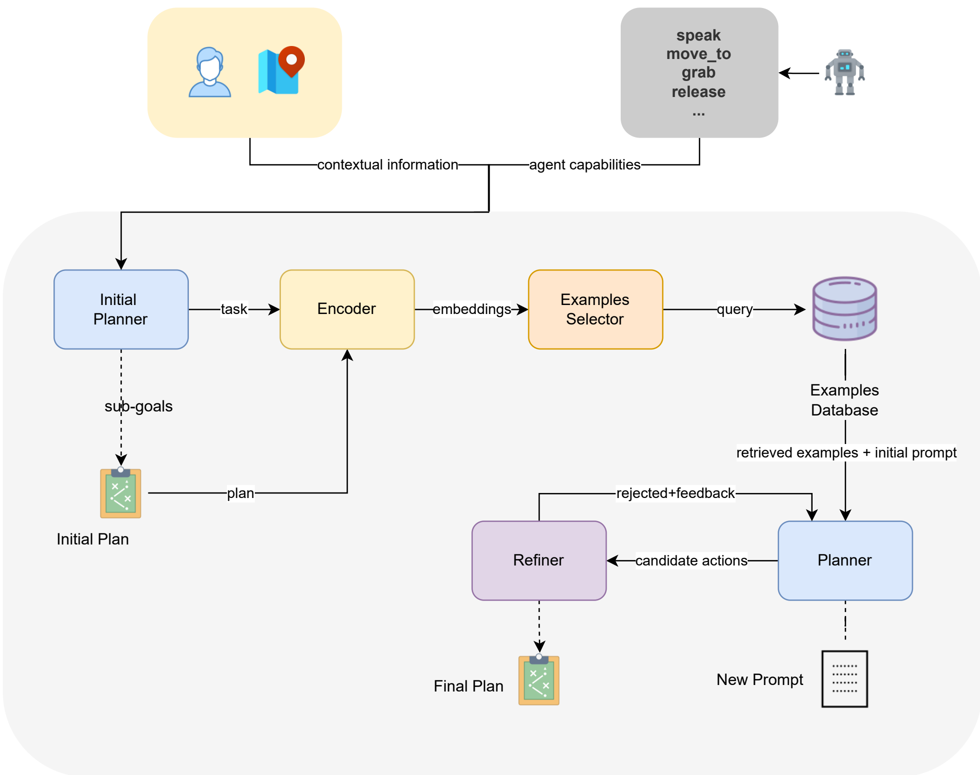



Figure 4.1: Overview of the main modules of our proposed workflow.

process, and each part of the workflow is described in the following sections.

## 4.1 Modeling the Agent's Capabilities

We define our agent's capabilities as the atomic actions it can perform. These capabilities are listed in natural language and as Python functions, including parameters and usage examples. Our agent's capabilities align with the action templates in the LEMMA dataset, which contains the tasks used in our experiments (see Section 5.1) (JIA et al., 2020). However, these capabilities can be easily adapted to other robots/agents. Figure 4.2 illustrates each possible action in this dataset. In this context, our ad hoc agent has human-like capabilities, with some actions involving complex object/tool pairings and up to three parameters.

| Verb | Template | | | | | Example | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| blend | blend | targets | | | with tools | blend | coffee | | | with | spoon |
| clean | clean | targets | | | with tools | clean | cup | | | with | sink |
| close | close | targets | | | | close | drawer | | | | |
| cook | cook | targets | in | location | with tools | cook | meat | in | pan | with | fork |
| cut | cut | targets | on | location | with tools | cut | watermelon | on | cutting-board | with | knife |
| drink | drink | targets | | | with tools | drink | milk | | | with | cup |
| eat | eat | targets | | | with tools | eat | meat | | | with | fork |
| fill | fill | targets | | | with tools | fill | juicer | | | with | sink |
| get | get | targets | from | location | with tools | get | spoon, fork | from | drawer | using | hand |
| open | open | targets | | | | open | closet | | | | |
| play | play | targets | | | with tools | play | game-console | | | with | controller |
| pour | pour | targets | into | location | with tools | pour | coffee | into | cup | with | spoon |
| put | put | targets | to | location | with tools | put | meat | to | pan | with | fork |
| sit-on | sit on | | | location | | sit on | | | sofa | | |
| sweep | sweep | targets | | | with tools | sweep | floor | | | with | vacuum |
| switch | switch | targets | | | with tools | switch | TV | | | with | remote |
| throw | throw | targets | into | location | | throw | wrapping | into | trash-can | | |
| turn-off | turn off | targets | | | with tools | turn off | TV | | | with | remote |
| turn-on | turn on | targets | | | with tools | turn on | microwave | | | with | hand |
| watch | watch | targets | | | | watch | TV | | | | |
| wash | wash | targets | | | | wash | cup, spoon | | | | |

Figure 4.2: Examples of the actions available to our agent when completing tasks in the LEMMA dataset. Adapted from (JIA et al., 2020).

These capabilities are sent to the Contextual Planner to be included in the system prompt. Figure 4.3 shows their representation in the prompt. There are 20 possible atomic actions our agent can perform.

## 4.2 Contextual Information

To enable contextual awareness, we represent the agent's environment using a semantic map, following approaches similar to Min et al. (2021) and Zheng, Wang e Jia (2023). This semantic map consists of a structured list containing the names and positions of all objects and agents in the scene at a given time.

```
Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of
capabilities and choose which objects that surround me I can use to solve those tasks. Those are my
capabilities:

1. blend: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format
blend(target,tool). For example: blend(coffee,spoon).
2. clean: to clean targets. For example: clean hand. Input should be in the format clean(target). For example:
clean(hand).
3. close: to close targets. For example: close drawer.  Input should be in the format close(target). For
example: close(drawer).
4. cook: to cook a target on tool or location. For example: coo meat on a pan. Input should be in the format
cook(target,tool). For example: cook(meat,pan).
5. cut: to cut targets with tools. For example: cut tomato with knife. Input should be in the format
cut(target,tool). For example: cut(tomato,knife).
6. drink: to drink targets with tools. For example: drink milk with cup. Input should be in the format
drink(target,tool). For example: drink(milk,cup).
7. eat: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format
eat(target,tool). For example: eat(sandwich,hand).
8. fill: to fill targets with tools/something. For example: fill tank with water. Input should be in the format
fill(target,tool). For example: fill(tank,water).
9. get: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input
should be in the format get(target,location, tool). For example: get(spoon|fork,drawer,hand).
10. open: to open targets. For example: open closet. Input should be in the format open(target). For example:
open(closet).
11. play: play targets with tools. For example: play game-console with controller. Input should be in the
format play(target,tool). For example: play(game-console(game),controller).
12. pour: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in
the format pour(target,location,tool). For example: pour(milk,cup,hand).
13. put: put targets to location with tools. For example: put meat to pan with fork. Input should be in the
format put(target,location,tool). For example: put(meat,pan,fork).
14. sweep: to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the
format sweep(target, tool). For example: sweep(floor, vacuum).
15. switch: to switch targets. For example: switch TV. Input should be in the format switch(target). For
example: switch(TV).
16. throw: to throw targets into location. For example: throw wrapping into trash-can. Input should be in the
format throw(target,location). For example: throw(wrapping,trash-can).
17. turn-off: to turn off targets with tools. For example: turn off TV with remote. Input should be in the
format turn-off(target,tool). For example: turn-off(TV,remote).
18. turn-on: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the
format turn-on(target,tool). For example: turn-on(microwave,hand).
19. watch: to watch targets. For example: watch TV. Input should be in the format watch(target). For example:
watch(TV).
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example:
wash(cup).
```

Figure 4.3: System prompt with the agent's capabilities and usage examples.

In addition to the semantic map, we incorporate the user's task request and brief instructions. The instructions are general, just detailing the main goals of the task. For our experiments, the semantic map is derived from the ground truth data of each task in the dataset. In a real-world deployment, this information would ideally be obtained from the robot's sensors (cameras and microphones). However, since our focus is on plan generation rather than physical execution, the contextual data is hardcoded and passed to the Contextual Planner for proof of concept. A list of all possible tasks in the dataset, along with their general instructions, is provided in Section 5.1 of the experiments chapter.

## 4.3 Contextual Planner

Using the contextual information obtained in the previous steps (the agent's capabilities, the objects in the environment, and the user's request), we use an LLM to analyze this data and generate an initial plan with general subgoals. The user's request is decomposed into sub-steps based on the agent's capabilities, applying the Chain of Thought (CoT) prompting technique described in Section 3.4.2.

### 4.3.1
### Generating an Initial Plan

To construct the CoT prompt, we first include the agent's capabilities shown on Figure 4.3. Next, we add an example of a user requesting a task, the available objects in the environment and their locations, and a directive for the LLM to "think step-by-step." We then include an example of the expected output: first breaking the task into main subgoals, then generating the plan with atomic actions. We also specify the output format to simplify function and parameter retrieval for further comparisons with the reference plan for the task. Through testing, we observed that LLMs formatted outputs more reliably with XML than JSON, as JSON often led to misplaced semicolons and structural errors. Thus, we ask for the final plan to be in XML. Figure 4.4 illustrates this prompt. Here, the system prompt contains task execution instructions, while the user prompt includes the task request itself.

With this initial plan, we then use in-context learning to further refine it. The next step involves retrieving meaningful examples to include in the prompt for few-shot learning.

### 4.3.2
### Selecting Examples

For example selection, we adapted the work of Gao et al. (2023), who proposed DAIL-SQL, a few-shot strategy for Text-to-SQL tasks. Since providing a few examples in the input prompt can significantly improve LLM performance, the authors of DAIL-SQL developed a method to gather the best examples for their task context. The authors had access to a database of triples (queries in natural language, corresponding SQL queries, and resulting tables). Their strategy selects examples based on their similarity to the input query in natural language. They masked the information on the tables, such as column names and values, leaving only SQL clauses, as to limit the influence of domain-specific information. Then, they generated embeddings of this data and retrieved examples via similarity search, implementing a form of Retrieval-Augmented Generation (LEWIS et al., 2020). Based on the user's query in natural language, a model generated an initial corresponding query in SQL. Both query formats were compared to the ones in the dataset, and the most similar query (with its masked SQL response) was selected and included in the final prompt for a new query generation. Since this method significantly

```
... System prompt with capabilities of the agent ...

When asked to complete a task, the final output plan should be in
xml tags, like the one shown below.

<plan><action>get(target,location,tool)</action>
<action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>

For example, let's say you receive the following request:

Input
You are a helpful assistant and have one task to complete: put a
winebottle on the shelf. Use the capabilities defined earlier. Use
the objects and tools listed below.
Objects and tools: winebottle, countertop, shelf, cabinet,
coffeemachine. Winebottle is over the countertop, coffeemachine is
inside the cabinet.
To complete this task, I want you to think step by step. The final
output plan should be in xml tags, like the one shown below.

Output
1 - First, I have to get the winebottle.
2 - After that, I need to go to the shelf.
3 - Finally, I need to place the winebottle on the shelf.
<plan><action>get(winebottle,countertop,hand)</action>
<action>put(winebottle,shelf,hand)</action></plan>


Your task is to watch tv. Here are things to know before you start:
the items you need for this task are mentioned in the environment
map.  Keep the area tidy, and return any used items to their
original place after completing the task.

Objects on the environment: TV, remote, table. Additional
information: remote at/in/on table.
Use the capabilities and objects informed. Think step by step.
```

Instructions for output formatting

One-Shot Chain of Thought Prompt

Requested Task

Figure 4.4: Prompt used for generating an initial plan based on the user's request.

improved the performance of LLMs on Text2SQL tasks, we decided to adapt this method to our context and see if it would improve plan generation.

We started by searching for a database of examples relevant to our context. As the dataset in which we performed our experiments is comprised of humans performing daily tasks in a household, we looked for datasets of indoor activities. We chose ALFWorld as our base of examples, a 3D simulator for robots to perform basic domestic tasks in a virtual house (SHRIDHAR et al., 2021). ALFWorld tasks are much simpler than the ones in LEMMA, (averaging five steps per task) and involve different agent capabilities and task structures. We chose a dataset with different tasks performed by an agent with fewer and simpler capabilities but in the same context. This helps to avoid task-specific bias while providing domain-specific context. We hope these examples can help the LLM understand the structure of task plans and the precedence of actions.

After removing duplicated task plans, we have 678 examples in our database. Figure 4.5 shows an example of a task plan in ALFWorld.

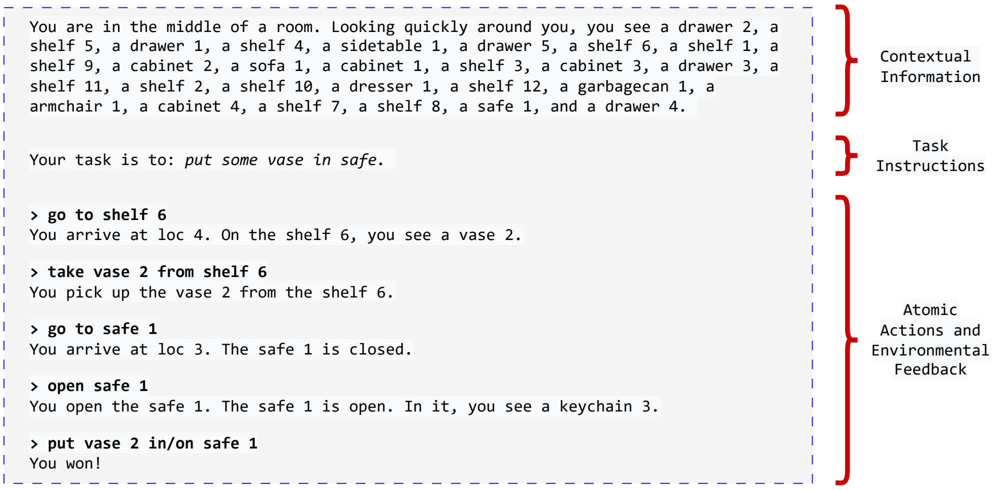



Figure 4.5: Example of a task plan in the ALFWorld dataset. Adapted from Shridhar et al. (2021)

We pre-processed the plans in ALFWorld, converting the actions from natural language to functions format, only using the task instructions and the atomic actions, and discarding the contextual information. Then, we prepared two formats of plans: 1 - a plan with actions and parameters and 2 - a masked plan, with only actions. With a masked plan, we want to limit the input of task-specific information, similar to Gao et al. (2023). Additionally, those two plan formats will help us understand what matters most in the example given to the LLM: the atomic actions combined with the affected objects or just the sequence of actions, independent of parameters. Figure 4.6 shows how the plans presented in Figure 4.5 look after this processing step.

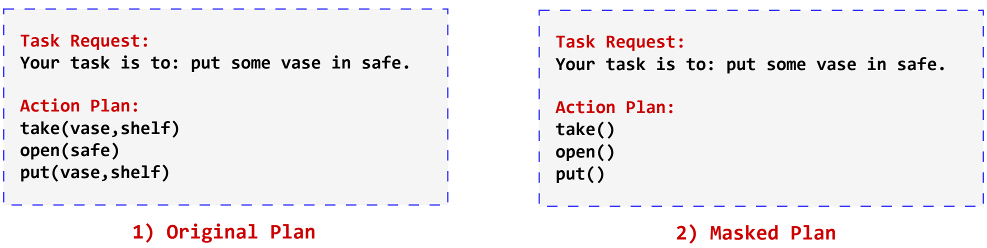



Figure 4.6: Formats of plans used in example retrieval.

With the tasks request and their corresponding formatted plans, we generate embeddings for the request and each plan format (plain or masked), storing them in memory. We used *all-MiniLM-L6-v2*[1], a popular Sentence

[1]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Transformer model for embedding generation. Now we have our vector store of examples and use this in the future for semantic searching.

Based on the user's task request and the initial plan obtained in the previous module of the workflow, we can now search for the most similar examples. We also format the initial plan in plain (functions with parameters) and masked (just the functions) formats, similar to the ones in ALFWorld. Now, we generate embeddings for the task request and the initial plans, and use those embeddings to search for meaningful examples based on cosine similarity, combining the results of the similarity of each search (similarity of request + similarity of the corresponding plan). After this, we build a rank of examples, picking the top $N$ most similar ones. The pseudo-code depicted on Algorithms 1 and 2 exemplify this process.

**Algorithm 1:** Retrieve Similar Examples

**Input:** task embedding $E_T$, plan embedding $E_P$, number of examples $n$, dataset `ALFWorld`, internal memory

**Output:** similar tasks $T_{sim}$, similar plans $P_{sim}$

$S_T \leftarrow$ CosineSimilarity($E_T$, `ALFWorld.task_embeddings`);<br>
$S_P \leftarrow$ CosineSimilarity($E_P$, `ALFWorld.plan_embeddings`);<br>
$S_{comb} \leftarrow S_T + S_P$;<br>
$r \leftarrow$ SortIndicesDescending($S_{comb}$);<br>
$T_{sim} \leftarrow \emptyset$;<br>
$P_{sim} \leftarrow \emptyset$;<br>
$counter \leftarrow 0$;<br>
$idx \leftarrow 1$;<br>
**while** $counter < n$ **do**<br>
    **if** $Skeleton(r[idx]) \notin P_{sim}$ **then**<br>
        Append `ALFWorld.tasks[`$r[idx]$`]` to $T_{sim}$;<br>
        Append `ALFWorld.plans[`$r[idx]$`]` to $P_{sim}$;<br>
        $counter \leftarrow counter + 1$;<br>
    $idx \leftarrow idx + 1$;<br>
    **if** $idx = |$*ALFWorld.task_embeddings*$|$ **then**<br>
        **break**<br>
**return** $T_{sim}, P_{sim}$

The $N$ selected examples are added to the final prompt, and the LLM generates another plan, now using a Few-Shot + Chain of Thought approach. Figure 4.7 shows the final prompt combining the retrieved examples using the masked plan format.

**Algorithm 2:** Generate Plan with Example Retrieval

**Input:** task request $T$, number of examples $n$, model $M$, flag $useSkeletons$

**Output:** final action plan $P$

$E_T \leftarrow \text{Embedding}(T)$;
$A \leftarrow \text{GenerateInitialPlan}(T,\ M)$;
**if** $useSkeletons$ **then**
  $E_P \leftarrow \text{Embedding}(\text{Skeleton}(A))$;
**else**
  $E_P \leftarrow \text{Embedding}(A)$;
$T_{ex}, P_{ex} \leftarrow \text{RetrieveExamples}(E_T,\ E_P,\ n)$;
$P \leftarrow \text{GenerateFinalPlan}(T,\ T_{ex},\ P_{ex},\ M)$;
**return** $P$

## 4.4 Refiner Module

Upon receiving the final plan, we use the same LLM to evaluate the generated output, analyzing the feasibility of the plan and whether the main goal was achieved. If needed, the model provides feedback to improve the plan, correcting possible errors and optimizing the output.

We tested several prompts for plan correction but found that as we added more guidelines for evaluation, the model tended to overanalyze plans, proposing too many changes even to plans that were already good, reducing the similarity score. Thus, we find that using a concise prompt led to better performance from the LLM, providing more nuanced feedback. After this step, we arrive at the final plan. Figure 4.8 shows the prompt template used for plan refinement.

```
... System prompt with capabilities of the agent ...
```

Instructions for output formatting

```
When asked to complete a task, the final output plan should be in
xml tags, like the one shown below.

<plan><action>get(target,location,tool)</action>
<action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>
```

Few-Shot Prompt

```
Here are some examples of task requests and their respective action
plans. These examples are from a robot with slightly different
capabilities, but it should help you. Keep in mind that these
examples don't have any additional rules imposed in the request.

Request: Your task is to clean some cup and put it in shelf.
Action Plan:
take(cup,countertop)
clean(cup,sinkbasin)
put(cup,shelf)

Request: Your task is to clean some cup and put it in cabinet.
Action Plan:
take(cup,countertop)
clean(cup,sinkbasin)
open(cabinet)
put(cup,cabinet)

Request: Your task is to clean some cup and put it in sidetable.
Action Plan:
take(cup,countertop)
clean(cup,sinkbasin)
put(cup,sidetable)
```

Task Request and Chain of Thought Prompt

```
Your task is to drink water. Here are things to know before you
start: the items you need for this task are mentioned in the
environment map.  Keep the area tidy, and return any used items to
their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser.
Additional Information: cup at/in/on table.
Use the capabilities and objects informed. Think step by step.
```

Figure 4.7: Final prompt with 3 examples from ALFWorld for the task *drink water*.

```
You are a helpful agent who analyses tasks plans and gives feedback for
improvements.

An LLM agent received the following instructions to complete a task: { prompt }.
Based on this request, the agent generated this action plan: { current plan }.
Analyze the feasibility of this plan, check if it's viable, and provide feedback
for improvements if needed.

If the task goal was achieved, or if don't have any additional feedback, output
<status>Ok</status>. If you want to propose a new plan, the output should be in
xml tags, like the one shown below.
Output:
<plan><action>get(target,location,tool)</action><action>fill(target,thing)
</action><action>put(targets,location,tools)</action></plan>
```

Figure 4.8: Prompt used for evaluating and refining the generated plan.

# 5
# Experiments and Results

This chapter describes all the experiments conducted to develop our proposed workflow. We begin by presenting the dataset of tasks used for our tests. We then introduce the LLMs used in our experiments and the evaluation metrics. Next, we begin our experimentation process by assessing the performance of these LLMs with the main prompting techniques presented in Chapter 3.

These experiments guided our selection of which techniques and models would be part of our final workflow. Finally, we present the experiments and results of our proposed workflow, as well as ablation studies to analyze the impact of different configurations for single-task and multi-task planning, and we discuss the results.

## 5.1
## LEMMA Dataset

To evaluate how LLMs would perform in generating plans for tasks in real-world scenarios with humans, we searched for datasets of tasks performed in the real-world. We found LEMMA, a dataset for video action recognition in multi-agent, multi-task activities with humans (JIA et al., 2020). Since this dataset primarily focuses on video action recognition, we conducted extensive pre-processing to get the final action plans and basic task instructions.

Each video on the dataset comes with annotations listing the objects present in the environment and the sequence of actions the users performed to complete a given task, allowing us to use this information to build reference action plans for each task. We analyzed every action plan to exclude invalid ones and selected a set of examples where the user successfully completed the task. After manually validating the users' plans for each task, we sent these plans (typically ranging from 5 to 13 examples per task) to an LLM to generate a reference plan. This reference plan combines the most common actions across the plans and integrates them with the available objects in each scenario. We thoroughly analyzed the LLM-generated reference plans, correcting errors and validating them. We also had to annotate general instructions for the tasks, as tasks on LEMMA have several sub-goals.

There are 16 possible tasks in this dataset: Sweep floor, Watch TV, Water Plant, Drink Water, Play Games, Make Cereal, Make Noodles, Eat Watermelon, Change Water (from fish tank), Heat Milk, Make Coffee, Make

Orange Juice, Make Apple Juice, Make Watermelon Juice, Make Sandwich, and Cook Meat. The tasks appear in different configurations:

- Single-agent Single-task **(1x1)**: One agent independently performs one task asked by the user;
- Single-agent Multi-task **(1x2)**: One agent simultaneously performs two tasks;
- Multi-agent Single-task **(2x1)**: Two agents cooperatively perform one task;
- Multi-agent Multi-task **(2x2)**: Both agents collaboratively complete two multi-step tasks.

In our experiments, we used the **1x1** and **1x2** task configurations to test our method and several prompting techniques on single task planning and multi-task planning. Additionally, we categorize LEMMA's tasks based on the level of mandatory steps required for completion. Easy tasks require 5 actions at most; medium-level tasks require up to 15 steps for completion. Tasks requiring more than 15 steps are classified as Complex tasks and can be considered long-horizon tasks with many sub-goals. Table 5.1 shows each category with its corresponding tasks and their minimum required steps for completion.

| Task Level | Tasks |
|---|---|
| **Easy** | Sweep floor (3), Watch TV (3), Water Plant (3), Drink Water (4) and Play Games (5) |
| **Medium** | Make Cereal (11), Make Noodles (11), Eat Watermelon (11), Change Water (from fish tank) (15) |
| **Complex** | Heat Milk (17), Make [Orange\|Apple\|Watermelon] Juice (16,17,18), Make Coffee (17), Make Sandwich (19), Cook Meat (21) |

Table 5.1: Types of tasks on the LEMMA dataset and the minimum number of steps required to complete each task.

Beyond the number of steps required for completion, these tasks involve manipulation of objects in the real world and action precedence before manipulating objects, which can be challenging for most LLMs. The possible actions available for task completion in this dataset were previously presented in Section 4.1 of Chapter 4.

## 5.2 Experimental Setup

For our initial tests, we selected several models from different providers, including both proprietary and open-source options. In this initial step, we aimed to select the best models in terms of both performance and cost. Beginning with open-source models, we tested two Llama models from Meta AI[1], two models from Mistral AI[2], and one model from Deepseek [3]. We excluded the reasoning model from Deepseek (R1) due to its extensive processing time, which would not be feasible for real-world tasks. For proprietary models, we tested three models from OpenAI[4]. Details about the models per provider along with their pricing are presented in Table 5.2.

| Provider | Model | Input Price | Output Price |
|---|---|---|---|
| MetaAI | Llama-3.1-8B-Instruct-Turbo | $ 0.18 | $ 0.18 |
| MetaAI | Llama-3.1-70B-Instruct-Turbo | $ 0.88 | $ 0.88 |
| MistralAI | Mistral-7B-Instruct-v0.3 | $ 0.20 | $ 0.20 |
| MistralAI | Mixtral-8x7B-Instruct-v0.1 | $ 0.60 | $ 0.60 |
| Deepseek | deepseek-chat (V3) | $ 0.135 | $ 0.550 |
| OpenAI | gpt-4o | $ 2.50 | $ 10.0 |
| OpenAI | gpt-4o-mini | $ 0.150 | $ 0.60 |
| OpenAI | o3-mini | $ 1.10 | $ 4.40 |

Table 5.2: Prices of LLMs calls per one million tokens. Accessed on April 6th, 2025.

Although open-source models are technically free and can run on personal computers or clusters, due to hardware limitations, we tested all open-source models via paid APIs. The Meta and Mistral models were accessed via the TogetherAI API[5], and Deepseek via their own API.

### 5.2.1 Evaluation Procedure

To grade the performance of LLMs on generating task plans, we adopted the LLM-as-a-Judge paradigm presented in Chapter 3. Due to the absence of a physical robot for real-world execution of tasks and feedback, we use an LLM Judge to compare the plans generated by the other LLMs against the reference plan mentioned in Section 5.1. The LLM Judge assigns a similarity score ranging from 0 to 1, indicating how closely each plan matched the reference.

[1]https://www.llama.com/
[2]https://mistral.ai/
[3]https://platform.deepseek.com/
[4]https://platform.openai.com/
[5]https://www.together.ai/

We tested multiple prompt variations to ensure the Judge could recognize equivalent actions and tool usage; the final evaluation prompt is shown in Figure 8.1 in the Appendix 8.

For a consistent evaluation, the LLM designated as the Judge was excluded from plan generation to avoid bias, ensuring all evaluated LLMs were scored under the same conditions. We selected the latest Claude Sonnet model (Claude 3-7 Sonnet) as our Judge [6].

## 5.3 Experiments with Main Prompting Techniques

This section presents the experiments conducted to evaluate the effectiveness of different prompting techniques and large language models (LLMs) for planning generation. We assessed performance, response time, and cost per task to determine the most suitable approaches for our workflow.

We applied the key prompting techniques introduced in Chapter 3, including: Zero-Shot prompting, Few-Shot Learning, Zero-Shot Chain of Thought (ZS-CoT), Standard Chain of Thought (CoT) and Tree of Thoughts (ToT). Figure 5.1 presents the average similarity score per task, grouped by LLM and prompting approach.

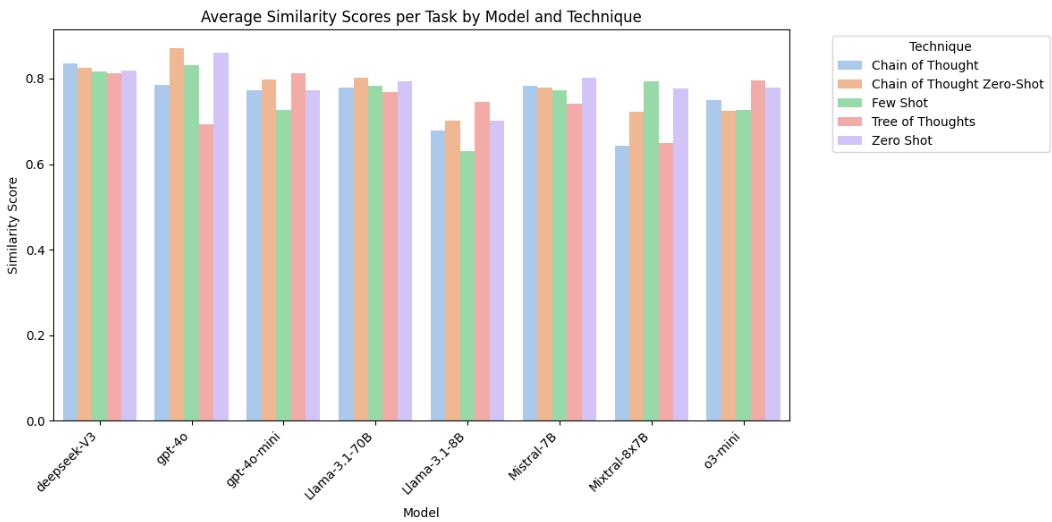



Figure 5.1: Average similarity score per task grouped by model and prompting technique.

The results, illustrated in Figure 5.1, reveal that Zero-Shot Chain of Thought (ZS-CoT) paired with GPT-4o achieved the highest similarity score at 87.08%. Notably, DeepSeek demonstrated consistency across all techniques, surpassing the 80% similarity score regardless of the prompting method used. This suggests its robustness for diverse applications in planning generation.

[6]https://www.anthropic.com/

When combining all prompting approaches and obtaining the average similarity score per model, we note that the best performing models across all techniques are DeepSeek-V3, GPT-4o and Llama-3.1-70B, respectively. Figure 5.2 shows the performance per model.

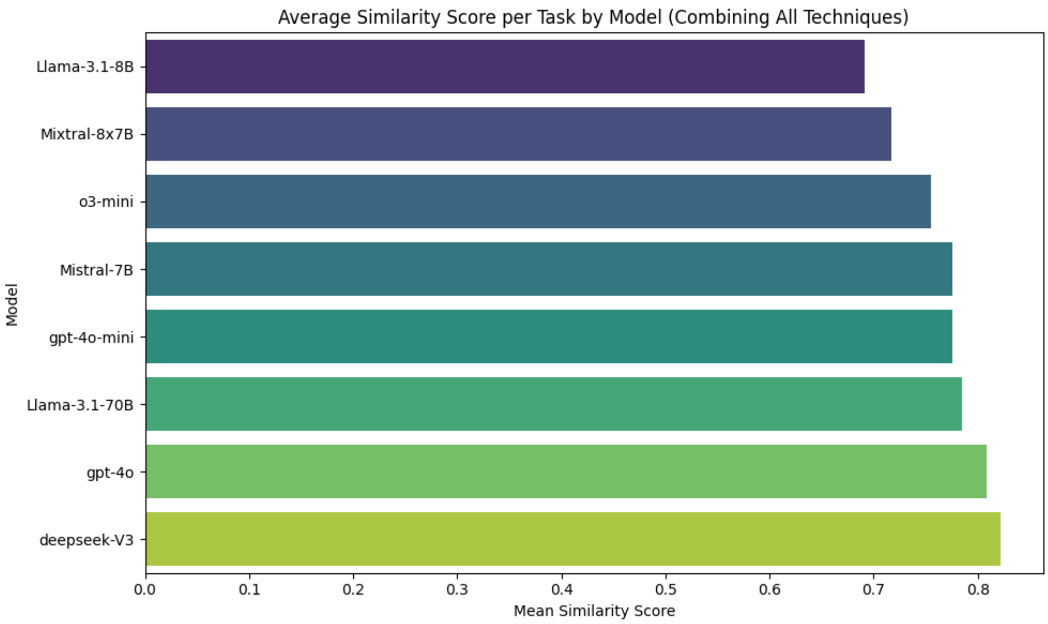



Figure 5.2: Average similarity score across all techniques per model.

Another important factor for an LLM-based agent that generates plans in real-time is the response time of the models. Figure 5.3 shows the average time for task completion by each model and prompting techniques. While most models generate plans under the 10 seconds marks, DeepSeek-V3 takes more than 20 seconds to complete a task.

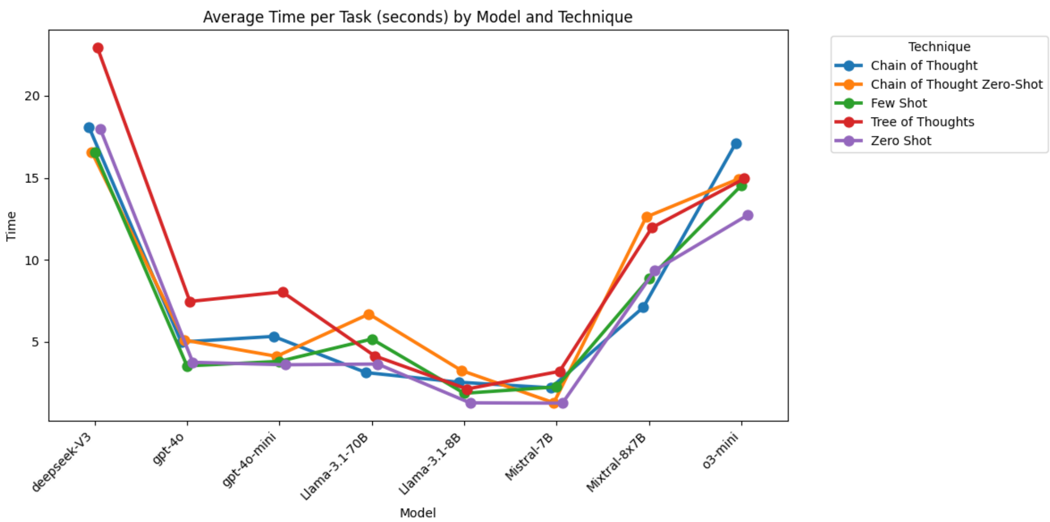



Figure 5.3: Average time in seconds for task completion.

Regarding the cost of generating task plans with LLMs, Figure5.4 shows that it is relatively cheap for most models, excluding the reasoning model o3-mini.

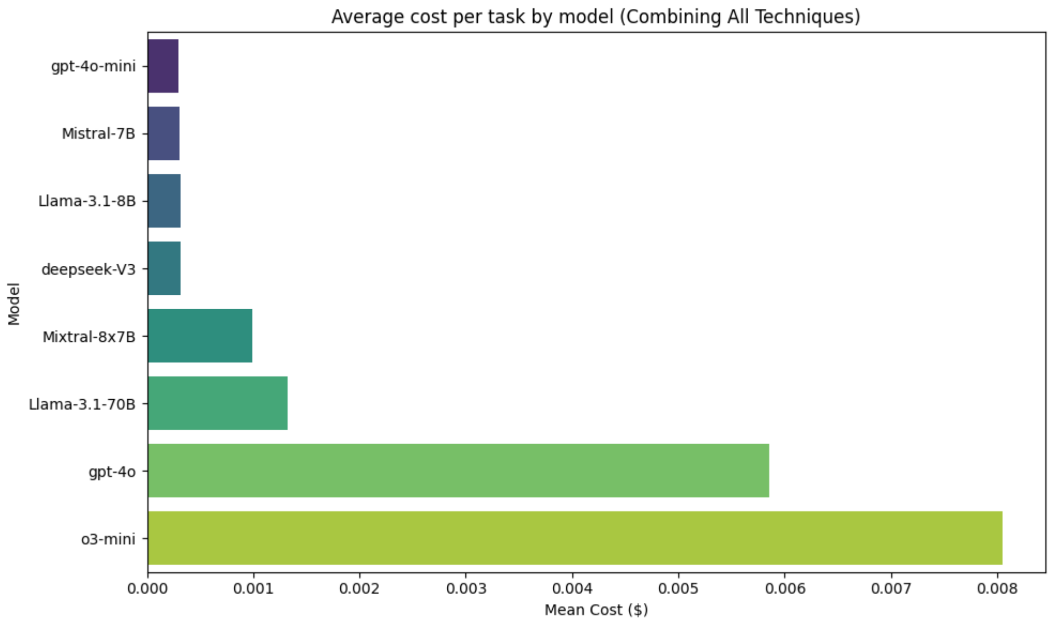



Figure 5.4: Average cost in dollars for task completion.

Finally, we plot the relationship of performance (based on similarity score) versus the cost of each model, which helped us choose the best modules for our further experiments. Figure 5.5 shows the results.

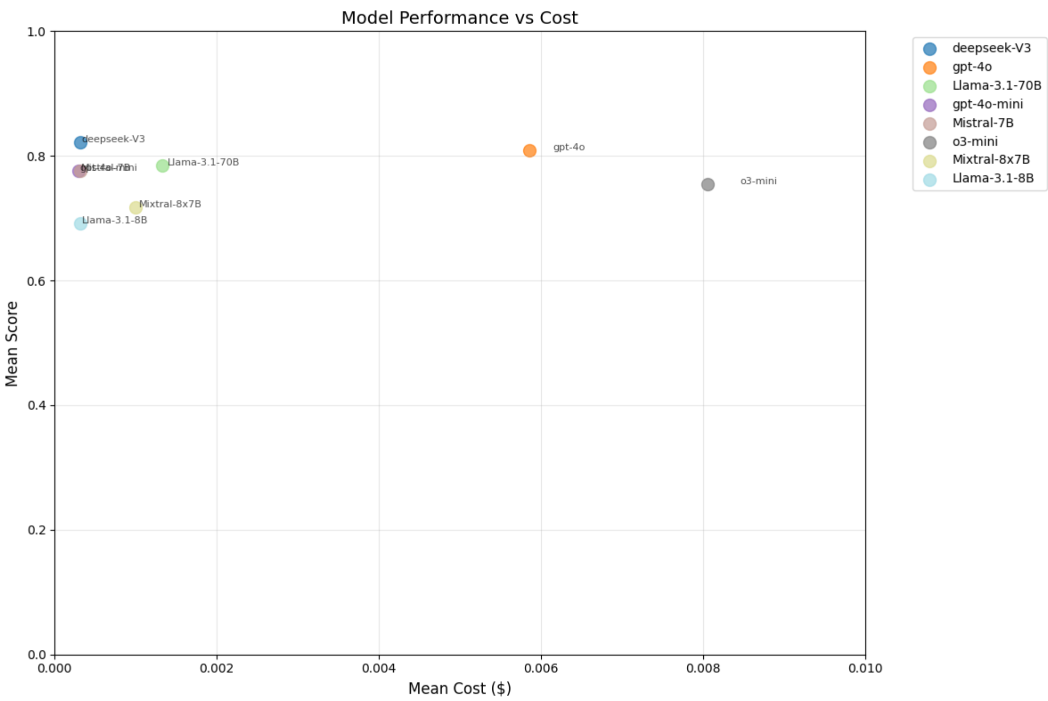



Figure 5.5: Cost (dollars) x Similarity Score per model.

In terms of cost efficiency, DeepSeek emerged as the most desirable option due to its near-zero cost per task. In contrast, OpenAI reasoning model o3-mini underperformed relative to their higher operational costs, making them less viable for large-scale deployment. Based on these findings, we will integrate Zero-Shot Chain of Thought (ZS-CoT) into our workflow, focusing on testing the models DeepSeek, GPT-4o, and Llama 3.1 70B.

## 5.4 Experiments with the Proposed Workflow

In our proposed method, we first use Zero-Shot Chain-of-Thought (CoT) prompting to generate an initial plan. We then experiment with varying the number of examples included in the prompt (which we call rank), and we also try different strategies for retrieving and representing these examples.

For the experiments involving the number of examples, we evaluated three configurations: *rank 1*, *rank 3*, and *rank 5*, which correspond to including 1, 3, and 5 examples, respectively. Regarding example retrieval, we explored three configurations:

– **Default:** We generate embeddings of the complete plans, including function names and their parameters, retrieve the most relevant examples, and include the full plans in the prompt.

– **SkeletonSearch:** We mask the parameters in the plans (keeping only the function names), generate embeddings of these masked plans, retrieve relevant examples based on this representation, and then include the full, unmasked versions of the examples in the prompt.

– **SkeletonPrompt:** Similar to *SkeletonSearch*, but the masked version of the plan is included in the prompt instead of the full plan.

These experiments aim to evaluate how the number and format of examples influence planning quality. We applied our workflow in the following scenario: single-task nd multi-task planning, where one agent performs one task. The following subsections describe each experiment in detail.

### 5.4.1 Planning for Single-Tasks

In the *One Agent, One Task* setting, the LLM receives a task request, general instructions, an environment map listing the available objects, and the agent's capabilities. We then prompt the LLM to generate an initial plan using Zero-Shot CoT, as illustrated in Figure 4.4.

Based on the task request and the initial plan, we retrieve relevant examples to include in the prompt. We analyze the effect of varying the number of examples (1, 3, or 5) and the format used, to understand whether full plans (with action names and the objects to interact with) or masked plans (only action names) contribute more effectively to performance.

We tested this workflow using the three best-performing LLMs from our previous evaluation (DeepSeek, GPT-4o, and LLaMA 3.1 70B), and we present the results in Figure 5.6.

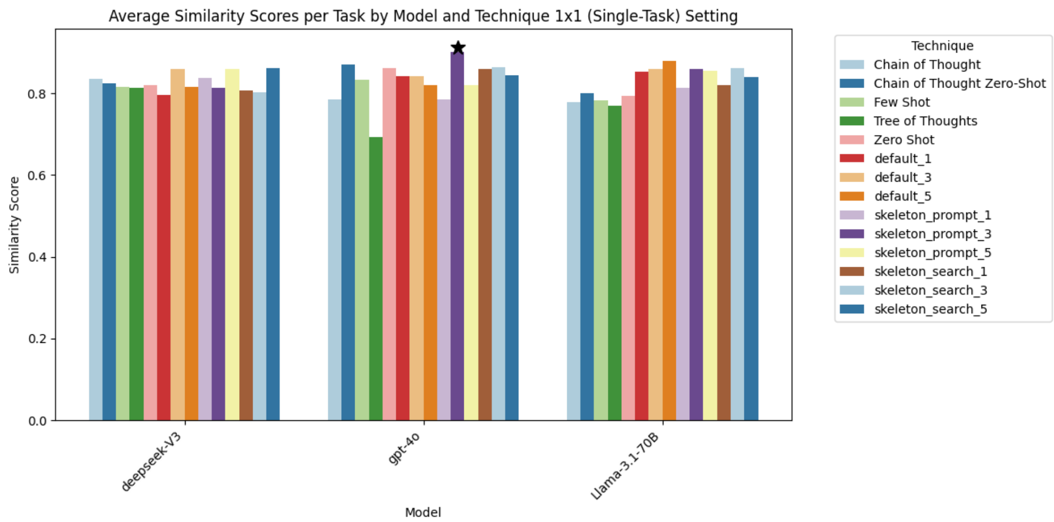



Figure 5.6: Average similarity score per task grouped by model and prompting technique for single-tasks (before plan refinements).

GPT-4o outperformed the other models, achieving the highest similarity score (90.20%) using the *SkeletonPrompt* technique with 3 examples. The best result using traditional prompting was 87.08% with Zero-Shot CoT, showing an improvement of over 3 percentage points in plan quality. In addition, half of the top 10 scores for the generated plans (see Table 5.3) belong to GPT-4o.

In terms of model comparison, LLaMA 3.1 70B performs competitively, especially in the *Default* configuration with 5 examples, achieving a similarity score of 87.88%. DeepSeek-V3, while slightly lower overall, still ranks in the top 5 when paired with the *SkeletonSearch* configuration using 5 examples.

Among the retrieval strategies, *SkeletonPrompt* with 3 examples and *Default* with 5 examples are the best configurations, suggesting that including more examples (3) or using complete plans (default setting) helps the LLM produce more accurate plans. The masked plan representation used for similarity search (both used with *SkeletonSearch* and *SkeletonPrompt*) also showed strong performance, appearing 7 times in the top 10 best scores, which suggests that structural similarity in plan format is an effective retrieval strategy. Regarding the number of examples, configurations with 3 examples appear 4

| Model | Technique | Score | Time | Tokens | Cost($) |
|---|---|---|---|---|---|
| gpt-4o | skeleton_prompt_3 | 90.20% | 10.89 | 2950.30 | 0.01 |
| Llama-3.1-70B | default_5 | 87.88% | 8.66 | 3250.13 | 0.00 |
| gpt-4o | Chain of Thought Zero-Shot | 87.08% | 5.12 | 1370.84 | 0.01 |
| gpt-4o | skeleton_search_3 | 86.34% | 9.02 | 3006.23 | 0.01 |
| deepseek-V3 | skeleton_search_5 | 86.25% | 43.63 | 3119.08 | 0.00 |
| Llama-3.1-70B | skeleton_search_3 | 86.18% | 9.45 | 3177.38 | 0.00 |
| gpt-4o | Zero Shot | 86.15% | 3.72 | 1316.54 | 0.00 |
| deepseek-V3 | skeleton_prompt_5 | 85.99% | 44.55 | 3058.49 | 0.00 |
| Llama-3.1-70B | skeleton_prompt_3 | 85.91% | 8.83 | 2975.83 | 0.00 |
| gpt-4o | skeleton_search_1 | 85.91% | 9.23 | 2926.05 | 0.01 |

Table 5.3: Top 10 configurations ranked by Similarity Score with their respective average time, token usage, and cost.

times in the top 10, while those with 5 examples appear 3 times, indicating that more examples can enhance plan quality.

In terms of cost and efficiency (Table 5.3), all top techniques stay within a low-cost range (less than $0.01), with moderate token usage. Traditional prompting techniques like *Chain of Thought Zero-Shot* and *Zero Shot* use fewer tokens and are faster, since they do not search for nor include any examples in the prompt, but they also tend to under perform in accuracy compared to our full workflow, which surpasses 90% in similarity score.

When it comes to time usage, it varies substantially depending on the technique and the model used. Inside the top 10, every technique that used DeepSeek-V3 surpassed the 40 seconds mark for plan generation, which can indicate an impediment for using this model in real-time applications despite its good performance and low cost. Both GPT-4o and LLaMA 3.1 70B deliver competitive performance within the 10-second range across all techniques.

The highest-performing configuration before plan refinements, *Skeleton-Prompt*, requires 10.9 seconds for plan generation. In contrast, faster alternatives like *Zero Shot* and *Chain of Thought Zero-Shot* take just 3.72 and 5.12 seconds, respectively, with a difference of minus 3 and 4 percentage points in plan similarity. Despite the higher accuracy, the best-performing techniques generally demand 2–3× more time and tokens. This trade-off reflects the cost of including multiple retrieved examples and longer prompts. In this context, Skeleton-based techniques mitigate redundancy, reducing token use despite long prompts. Thus, while adding more examples improves accuracy, it also increases latency and token usage. This is an important consideration for real-time applications, where performance may need to be balanced with response time and cost.

When it comes to plan refinements, Figure 5.7 and Table 5.4 compare similarity scores before and after plan refinement.

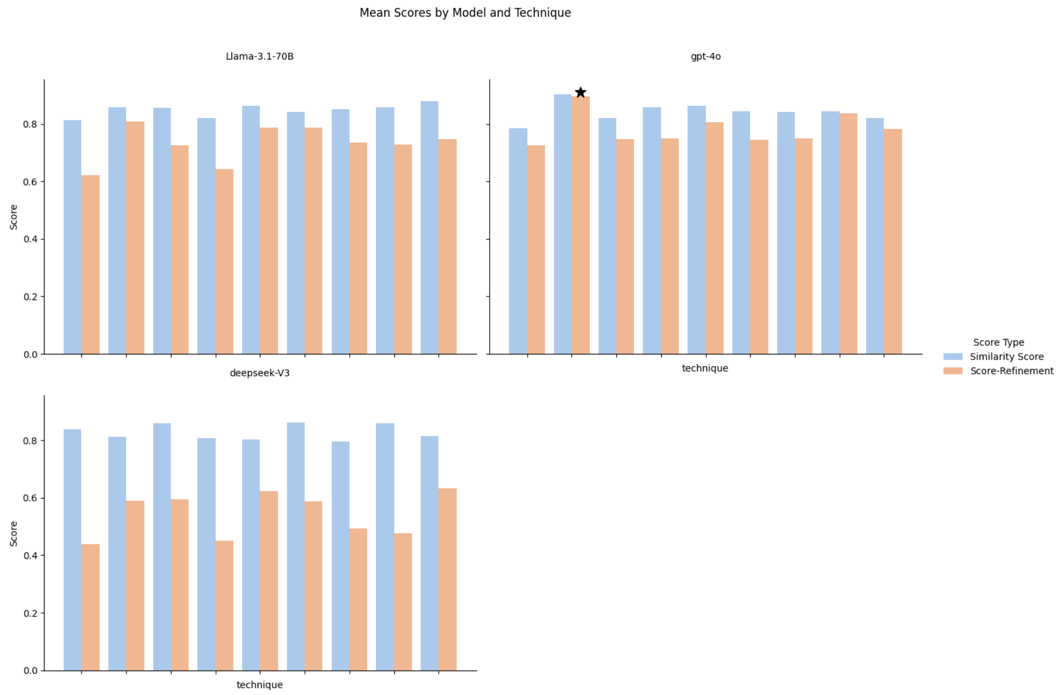



Figure 5.7: Average similarity score per task (before and after plan refinements) grouped by model. The star indicates the highest score amongst the refined plans.

We observe that for all configurations, refinements did not improve the quality of the plans, independent of the model and technique. Even the best score afeter refinement (marked with a star), is stil lower than the score previous to the refinement. For certain models, such as Deepseek-V3, the refinement decreased the quality of the plan in more than 30 percentage points. Table 5.4 also shows the impact of refinements on the total time, which nearly doubled the time to generate the plan. This indicates that for single tasks and simpler scenarios, plan refinement is not suitable, the initial plans are already good enough.

Overall, these results demonstrate that prompt quality, particularly in terms of example selection and representation, has a significant impact on plan quality, and that a combination of structured retrieval and refinement yields the best results, especially when paired with stronger models like GPT-4o. For single tasks, the use of plan refinements is not needed.

### 5.4.2 Planning for Multi-Tasks

In a similar manner, we tested our workflow with the three best-performing models (GPT-4o, Llama-3.1-70B, and DeepSeek-V3) in the *One*

| Model | Technique | Score | Score-R | Time | Time-R | Cost | Cost-R | Time-S |
|---|---|---|---|---|---|---|---|---|
| gpt-4o | skeleton_prompt_3 | 90.20% | 89.68% | 10.89 | 11.28 | 0.01 | 0.01 | 22.17 |
| gpt-4o | default_3 | 84.31% | 83.76% | 9.78 | 10.26 | 0.01 | 0.01 | 20.05 |
| Llama-3.1-70B | skeleton_prompt_3 | 85.91% | 80.92% | 8.84 | 4.84 | 0.00 | 0.00 | 13.68 |
| gpt-4o | skeleton_search_3 | 86.34% | 80.52% | 9.02 | 7.62 | 0.01 | 0.01 | 16.64 |
| Llama-3.1-70B | skeleton_search_5 | 84.06% | 78.71% | 5.27 | 4.35 | 0.00 | 0.00 | 9.62 |
| Llama-3.1-70B | skeleton_search_3 | 86.18% | 78.65% | 9.45 | 6.00 | 0.00 | 0.00 | 15.09 |
| gpt-4o | default_5 | 81.97% | 78.35% | 8.90 | 9.32 | 0.01 | 0.01 | 18.23 |
| gpt-4o | skeleton_search_1 | 85.91% | 74.93% | 9.23 | 7.60 | 0.01 | 0.01 | 16.83 |
| gpt-4o | default_1 | 84.12% | 74.91% | 10.42 | 9.34 | 0.01 | 0.01 | 15.80 |
| gpt-4o | skeleton_prompt_5 | 82.05% | 74.72% | 9.81 | 9.47 | 0.01 | 0.01 | 19.28 |

Table 5.4: Top 10 configurations with their respective score (Score), score after refinement (Score-R), average time and cost before and after refinement, as well as total time (Time-S).

*Agent, Two Tasks* (1x2) setting. The task pairs with varying levels of complexity are listed in Table 5.5:

| Task Level | Pair of Tasks |
|---|---|
| **Easy** | Drink Water(4), Water Plant(3); Drink Water(4), Play Games(5);<br>Sweep Floor(3), Watch TV(3); Watch TV(3), Water Plant(3); |
| **Medium** | Change Water (from fish tank)(15), Water Plant(3); |
| **Complex** | Make Coffee(17), Make Sandwich(19);<br>Heat Milk(17), Make Watermelon Juice(18);<br>Make Coffee(17), Make Watermelon Juice(18);<br>Make Coffee(17), Make Noodle(11);<br>Cook Meat(21), Make Noodle(11); Make Cereal(11), Make Sandwich(19);<br>Eat Watermelon(11), Make Watermelon Juice(18);<br>Heat Milk(17), Make Coffee(17);<br>Heat Milk(17), Make Cereal(11); Cook Meat(21), Make Sandwich(19);<br>Make Cereal(11), Make Coffee(17); |

Table 5.5: Types of tasks in the LEMMA dataset for the 1x2 *One Agent, Two Tasks* setting. The number in parentheses indicates the minimum number of steps required to complete each task.

The LLM receives a request to complete two tasks, general instructions for both tasks, and an environment map with all the objects needed for their completion, as well as the agent's capabilities. Similar to the 1x1 setting, we prompt the agent to generate an initial plan to complete both tasks. Based on this plan, we search for meaningful examples to include in the prompt, generate a final plan, and then refine this plan using the LLM again.

It is worth mentioning that the examples (either 1, 3, or 5) are selected based on the joint task request rather than on each task individually. The evaluation process also differs slightly for multi-task settings: each task is evaluated and scored separately; based on the scores for each task, we get the mean for both tasks. This makes the evaluation process take longer to complete.

Figure 5.8 shows the overall results before plan refinements, comparing the results of traditional prompting techniques and the results of the experiments with our workflow, while Table 5.6 shows the top 10 scores for plan generation.

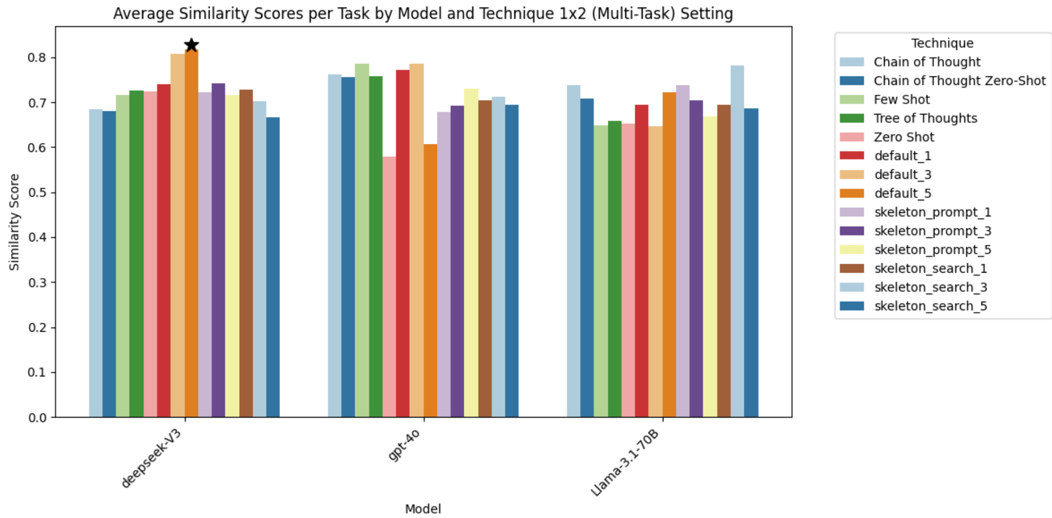



Figure 5.8: Average similarity score per task grouped by model and prompting technique for multi-tasks (before plan refinements).

Differently from the experiments with single-tasks, the best performing model for multi-tasks is Deepseek-V3, with the top-2 best results in the *Default* configuration with 5 and 3 examples, respectively, achieving 81.71% and 80.69% before plan refinements. The best result using traditional prompt techniques was 78.49% with GPT4-o using Few-Shot prompting, which shows an improvement of more than 3 percentage points with our workflow. We can see that for multitasks, using more examples (5 or 3) improved performance over a single example (Few-Shot), especially for the *Default* configurations. However, the gain from 5 to 3 examples was not that substantial.

Regarding the time for plan generation, it has significantly increased in comparison to the single tasks, achieving 1 minute and 6 seconds for plan generation before refinements with the highest similarity score. We notice that every technique that used Deepseek-V3 surpassed (or got close to) the minute mark, while GPT-4o remained around the 10-11 seconds, and Llama-3.1-70B, around 17 seconds. Although these times (the Deepseek ones) are acceptable for interactive or near real-time systems, they may be too slow for time-critical dynamic environments.

For the refinements part of our workflow, Figure 5.9 shows the similarity scores before and after refinement. We can see that for multi-tasks, the

| Model | Technique | Score | Time | Tokens | Cost($) |
|---|---|---|---|---|---|
| deepseek-V3 | default_5 | 81.71% | 66.26 | 3618.07 | 0.00 |
| deepseek-V3 | default_3 | 80.69% | 65.73 | 3533.38 | 0.00 |
| gpt-4o | default_3 | 78.49% | 10.13 | 3229.17 | 0.01 |
| gpt-4o | Few Shot | 78.47% | 5.31 | 1594.70 | 0.00 |
| Llama-3.1-70B | skeleton_search_3 | 78.23% | 17.15 | 3452.82 | 0.00 |
| gpt-4o | default_1 | 77.23% | 11.21 | 3174.52 | 0.01 |
| gpt-4o | Chain of Thought | 76.14% | 6.54 | 1705.32 | 0.00 |
| gpt-4o | Tree of Thoughts | 75.77% | 11.52 | 1677.33 | 0.00 |
| gpt-4o | Chain of Thought Zero-Shot | 75.56% | 6.35 | 1531.70 | 0.00 |
| deepseek-V3 | skeleton_prompt_3 | 74.28% | 57.70 | 3612.23 | 0.00 |

Table 5.6: Top 10 configurations for multi-tasks (1x2 setting) ranked by Similarity Score with their respective average time (in seconds), token usage, and cost.

refinement step resulted in similar or higher scores in most of the cases, achieving the best result with the *Default* setting with 5 examples using Deepseek-V3, 82.20%, an increase of five percentage points compared to the best result with traditional prompting techniques. We see that for the multi-task setting, the refinement part was much more beneficial, whici suggests that its use is more suited for complex scenarios.

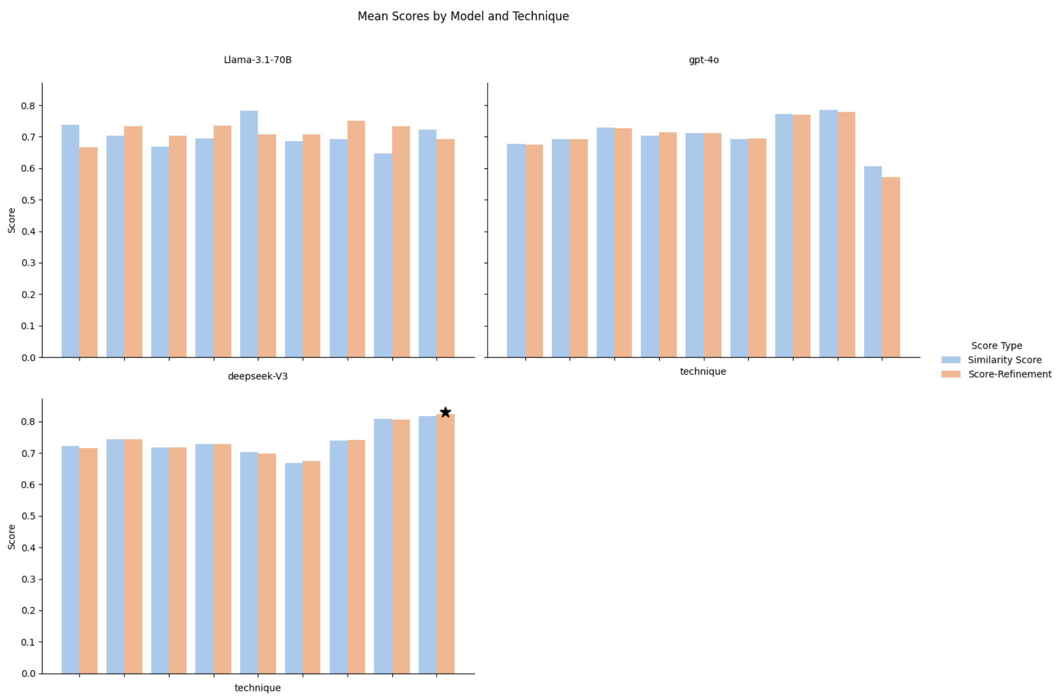



Figure 5.9: Average similarity score per task grouped by model for multi-tasks (before plan refinements). The star indicates the highest score amongst all plans.

Table 5.7 shows the top 10 results after refinements with more detail, also showing the total time and cost. In these experiments, the *Default* and *SkeletonSearch* configurations outperformed *SkeletonPrompt*, which suggests that including the full plan in the prompt improves plan generation for complex multi-task scenarios. Llama-3.1-70B was the model that presented more gain in the refined plans, with increases of 6 to 9 percentage points, while keeping the total time under 17 seconds. Deepseek-V3, however, reached the 1 minute and half mark with the best score for plan similarity, which hinders its use in critical real-time environment, despite its good results.

| Model | Technique | Score | Score-R | Time | Time-R | Cost | Cost-R | Time-S |
|---|---|---|---|---|---|---|---|---|
| deepseek-V3 | default_5 | 81.70% | 82.20% | 66.26 | 22.76 | 0.00 | 0.00 | 89.01 |
| deepseek-V3 | default_3 | 80.69% | 80.50% | 65.73 | 20.91 | 0.00 | 0.00 | 86.63 |
| gpt-4o | default_3 | 78.49% | 77.90% | 10.13 | 9.13 | 0.01 | 0.01 | 19.86 |
| gpt-4o | default_1 | 77.23% | 77.05% | 11.21 | 9.65 | 0.01 | 0.01 | 20.86 |
| Llama-3.1-70B | default_1 | 69.39% | 75.10% | 11.09 | 4.54 | 0.00 | 0.00 | 15.62 |
| deepseek-V3 | skeleton_prompt_3 | 74.28% | 74.37% | 57.70 | 18.93 | 0.00 | 0.00 | 76.63 |
| deepseek-V3 | default_1 | 73.96% | 74.10% | 64.42 | 19.37 | 0.00 | 0.00 | 83.79 |
| Llama-3.1-70B | skeleton_search_1 | 69.47% | 73.59% | 10.18 | 5.21 | 0.00 | 0.00 | 15.39 |
| Llama-3.1-70B | default_3 | 64.63% | 73.28% | 10.42 | 5.26 | 0.00 | 0.00 | 15.68 |
| Llama-3.1-70B | skeleton_prompt_3 | 70.31% | 73.28% | 10.97 | 5.78 | 0.00 | 0.00 | 16.75 |

Table 5.7: Top 10 configurations with their respective score (Score), refined score (Score-R), average and refined time and cost, as well as totals.

In general, we can see that the planning quality was lower in multi-task settings compared to single-task, which is expected due to the increased complexity in handling multiple tasks simultaneously. However, for the multi-task setting, the workflow significantly outperformed traditional prompting techniques, increasing the plan quality in 4 percentage points. Even in more complex multi-task scenarios, the workflow maintains superior performance over baseline prompting. This confirms that prompt structure and example retrieval strategy are key to maximizing planning performance. Default strategies using full plan examples are more beneficial in complex multi-task prompts. Plan refinements led to significant improvements for most techniques in the multi-task settings. This shows that complex scenarios such as multi-task plan generation benefit more from refinements than single ones.

To conclude, our proposed workflow demonstrated strong performance across both simple and complex task planning scenarios. By retrieving and incorporating meaningful examples into the prompt, such as full plans using the Default strategy, we significantly improved the quality of the generated plans compared to traditional prompting techniques. This effect was especially significant in multi-task settings, where the workflow achieved up to 4 percentage points of improvement and where plan refinements provided the most substantial gains. These results highlight the importance of contextually relevant examples, well-structured prompts, and refinement stages in generating better plans with LLMs.

# 6
# Conclusions

This thesis presented a workflow for planning tasks in ad hoc teamwork (AHT) scenarios using Large Language Models (LLMs). Given the dynamic nature of such environments, our goal was to equip ad hoc agents with mechanisms that allow them to adapt easily to different contexts and tasks. Since LLMs have shown remarkable generalization capabilities, we investigated their use in AHT scenarios involving human teammates performing simulated real-world tasks.

To address our first research question (*R.Q 1*), we proposed a novel LLM-based workflow that integrates task understanding, contextual awareness, in-context learning, and self-refinement. The workflow begins with Zero-Shot Chain-of-Thought prompting to generate an initial plan, which is then improved by incorporating contextually retrieved examples. We experimented with multiple retrieval and formatting strategies, including default and skeleton-based representations, and varied the number of examples to assess their impact on performance.

Our results showcase the potential of integrating LLMs into ad hoc agents to enhance planning in real-world human-robot collaboration. By leveraging the reasoning and generalization capabilities of LLMs, our workflow enables agents to identify collaboration opportunities and generate high-quality plans in dynamic environments without relying on predefined task libraries or rigid coordination strategies.

Regarding our second research question (*R.Q 2*), our experiments demonstrated that plan quality, cost, and latency can be effectively balanced. The best-performing configuration for single tasks (*skeletonPrompt* with 3 examples) achieved a similarity score of 90.20% without refinement, with a generation time of 11 seconds and a cost of $0.01. Compared to the best baseline technique (Zero-Shot CoT, 87.08%), our workflow improved plan quality by 3 percentage points while remaining suitable for real-time use. For multi-task scenarios, the best result was 82.20% after refinement, 4 percentage points higher than the best result with traditional prompting techniques. However, this setting may not meet real-time constraints, as it required 1.5 minutes to generate the final plan.

While the highest scores required more tokens and time, our results showed that prompt design and relevant examples had a greater impact on performance. Model selection was also critical: GPT-4o and Llama-3.1-70B kept plan generation under 18 seconds, while DeepSeek-V3 exceeded the one minute mark in complex multi-task scenarios. These findings validate the feasibility of using LLMs for planning in ad hoc settings, supporting both reliability and cost-effectiveness.

Overall, our results show that combining LLMs' reasoning abilities with contextual example retrieval and refinements for more complex scenarios significantly enhances task planning in dynamic environments. The proposed workflow enables agents to understand user instructions, adapt to new tasks, and generate executable plans without prior task-specific training. This contributes to the broader goal of building trustworthy and adaptive agents for mixed human-robot teams.

Despite these promising outcomes, this study has its limitations. Experiments were conducted in a simulated environment, and real-world deployment introduces additional challenges such as sensor noise, hardware constraints, and the unpredictability of human behavior, which requires further investigation. In addition, our experiments were limited to a single agent in both single and multi-task scenarios.

In this context, this work opens several directions for future research. First, we plan to extend the workflow to support collaborative scenarios, where two agents must coordinate multiple goals or work alongside humans on shared tasks. Second, we aim to explore more advanced prompting techniques, such as Graph-of-Thoughts or Tree-of-Thoughts, which may further enhance planning robustness. We also intend to investigate alternative approaches to Retrieval-Augmented Generation (RAG) to improve the quality of examples included in the prompts. Finally, we aim to deploy our workflow in a physical robot that receives input from multiple sensors to assess its real-world performance.

By continuing this line of research, we hope to contribute to the development of adaptable, intelligent, and practical agents capable of robust collaboration in real-world ad hoc teamwork scenarios.

# 7
# Bibliography

ALAMMAR, J.; GROOTENDORST, M. **Hands-on large language models: language understanding and generation**. [S.l.]: " O'Reilly Media, Inc.", 2024. Cited 2 times in pages 12 and 33.

BARRETT, S. et al. Making friends on the fly: Cooperating with new teammates. **Artificial Intelligence**, Elsevier, v. 242, p. 132–171, 2017. Cited 3 times in pages 18, 22, and 25.

BARRETT, S. et al. Making friends on the fly: Cooperating with new teammates. **Artificial Intelligence**, v. 242, p. 132–171, 2017. Cited in page 30.

BARRETT, S.; STONE, P. Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2015. v. 29, n. 1. Cited in page 18.

BELPAEME, T. et al. Social robots for education: A review. **Science robotics**, American Association for the Advancement of Science, v. 3, n. 21, p. eaat5954, 2018. Cited in page 18.

BORBONI, A. et al. The expanding role of artificial intelligence in collaborative robots for industrial applications: a systematic review of recent works. **Machines**, MDPI, v. 11, n. 1, p. 111, 2023. Cited in page 18.

BORGES, P. V. et al. Multimodal prompt engineering for multimedia applications using the gpt model. 2024. Cited 4 times in pages 31, 32, 33, and 36.

BROHAN, A. et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. **arXiv preprint arXiv:2307.15818**, 2023. Cited 2 times in pages 18 and 24.

BROWN, T. et al. Language models are few-shot learners. **Advances in neural information processing systems**, v. 33, p. 1877–1901, 2020. Cited 2 times in pages 32 and 33.

DEVLIN, J. et al. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. 2019. Cited in page 24.

DODAMPEGAMA, H.; SRIDHARAN, M. Generic-to-specific reasoning and learning for ad hoc teamwork. 2025. Cited in page 24.

FOSONG, E. et al. Learning complex teamwork tasks using a given sub-task decomposition. In: **Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems**. [S.l.: s.n.], 2024. p. 598–606. Cited in page 18.

GAO, D. et al. Text-to-sql empowered by large language models: a benchmark evaluation. arxiv. **arXiv preprint arXiv:2308.15363**, 2023. Cited 2 times in pages 43 and 45.

GIRAY, L. Prompt engineering with chatgpt: a guide for academic writers. **Annals of biomedical engineering**, Springer, v. 51, n. 12, p. 2629–2633, 2023. Cited in page 32.

GRUDA, D.; MCCLESKEY, J. How to avoid others and influence people: Attachment orientations predict leader prototypicality in ad hoc teams. **Acta Psychologica**, Elsevier, v. 230, p. 103742, 2022. Cited in page 29.

HUANG, S. et al. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. **arXiv preprint arXiv:2305.11176**, 2023. Cited 2 times in pages 19 and 26.

HUANG, X. et al. Understanding the planning of llm agents: A survey. **arXiv preprint arXiv:2402.02716**, 2024. Cited 2 times in pages 31 and 36.

HULBERT, D. **Using Tree-of-Thought Prompting to boost ChatGPT's reasoning**. Zenodo, 2023. https://github.com/dave1010/tree-of-thought-prompting. Disponível em: https://doi.org/10.5281/zenodo.10323452. Cited 3 times in pages 12, 35, and 36.

HUYEN, C. **AI Engineering: Building Applications with Foundation Models**. [S.l.]: O'Reilly, 2025. Cited 4 times in pages 12, 33, 37, and 38.

JIA, B. et al. Lemma: A multi-view dataset for le arning m ulti-agent m ulti-task a ctivities. In: SPRINGER. **European Conference on Computer Vision**. [S.l.], 2020. p. 767–786. Cited 3 times in pages 12, 41, and 49.

KOJIMA, T. et al. Large language models are zero-shot reasoners. **Advances in neural information processing systems**, v. 35, p. 22199–22213, 2022. Cited in page 34.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. **Advances in neural information processing systems**, v. 33, p. 9459–9474, 2020. Cited in page 43.

LIU, A. et al. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. **arXiv preprint arXiv:2303.13547**, 2023. Cited in page 33.

LIU, P. et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. **ACM Computing Surveys**, ACM New York, NY, v. 55, n. 9, p. 1–35, 2023. Cited in page 33.

LIU, X. et al. Leveraging large language model for heterogeneous ad hoc teamwork collaboration. **arXiv preprint arXiv:2406.12224**, 2024. Cited 2 times in pages 23 and 25.

MADAAN, A. et al. Self-refine: Iterative refinement with self-feedback. **Advances in Neural Information Processing Systems**, v. 36, p. 46534–46594, 2023. Cited 3 times in pages 12, 36, and 37.

MELO, F. S.; SARDINHA, A. Ad hoc teamwork by learning teammates' task. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 30, p. 175–219, 2016. Cited 2 times in pages 23 and 25.

MIN, S. Y. et al. Film: Following instructions in language with modular methods. **arXiv preprint arXiv:2110.07342**, 2021. Cited 2 times in pages 24 and 41.

MIRSKY, R. et al. A survey of ad hoc teamwork research. In: **European Conference on Multi-Agent Systems (EUMAS)**. [S.l.: s.n.], 2022. Cited 2 times in pages 18 and 29.

PRATTICÒ, F. G.; LAMBERTI, F. Mixed-reality robotic games: design guidelines for effective entertainment with consumer robots. **IEEE Consumer Electronics Magazine**, IEEE, v. 10, n. 1, p. 6–16, 2020. Cited in page 18.

RAHMAN, A. et al. A general learning framework for open ad hoc teamwork using graph-based policy learning. **Journal of Machine Learning Research**, v. 24, n. 298, p. 1–74, 2023. Cited in page 19.

RASCHKA, S. **Build a Large Language Model (From Scratch)**. [S.l.]: Simon and Schuster, 2024. Cited in page 30.

RAVULA, M. C. R. **Ad-hoc teamwork with behavior-switching agents**. Tese (Doutorado) — University of Texas, 2019. Cited in page 18.

RIBEIRO, J. G. et al. Helping people on the fly: Ad hoc teamwork for human-robot teams. In: SPRINGER. **Progress in Artificial Intelligence: 20th EPIA Conference on Artificial Intelligence, EPIA 2021, Virtual Event, September 7–9, 2021, Proceedings 20**. [S.l.], 2021. p. 635–647. Cited in page 23.

RIBEIRO, J. G. et al. **HOTSPOT: An ad hoc teamwork platform for mixed human-robot teams**. [S.l.]: Public Library of Science San Francisco, CA USA, 2024. e0305705 p. Cited 4 times in pages 19, 23, 25, and 30.

RIBEIRO, J. G. et al. Assisting unknown teammates in unknown tasks: Ad hoc teamwork under partial observability. **arXiv preprint arXiv:2201.03538**, 2022. Cited in page 18.

RIBEIRO, J. G. et al. Teamster: Model-based reinforcement learning for ad hoc teamwork. **Artificial Intelligence**, Elsevier, v. 324, p. 104013, 2023. Cited in page 18.

SHRIDHAR, M. et al. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In: **Proceedings of the International Conference on Learning Representations (ICLR)**. [s.n.], 2021. Disponível em: https://arxiv.org/abs/2010.03768. Cited 3 times in pages 12, 44, and 45.

SINGH, I.; TRAUM, D.; THOMASON, J. Twostep: Multi-agent task planning using classical planners and large language models. **arXiv preprint arXiv:2403.17246**, 2024. Cited 2 times in pages 26 and 28.

STONE, P. et al. Ad hoc autonomous agent teams: Collaboration without precoordination. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2010. v. 24, n. 1, p. 1504–1509. Cited 2 times in pages 18 and 29.

STONE, P. et al. Teaching and leading an ad hoc teammate: Collaboration without pre-coordination. **Artificial Intelligence**, Elsevier, v. 203, p. 35–65, 2013. Cited in page 29.

TASAKI, R. et al. Prototype design of medical round supporting robot "terapio". In: IEEE. **2015 IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.], 2015. p. 829–834. Cited in page 18.

TOUVRON, H. et al. Llama: Open and efficient foundation language models. **arXiv preprint arXiv:2302.13971**, 2023. Cited in page 31.

TRAINUM, K. et al. Robots in assisted living facilities: scoping review. **JMIR aging**, JMIR Publications Inc., Toronto, Canada, v. 6, n. 1, p. e42652, 2023. Cited in page 18.

VASWANI, A. et al. Attention is all you need. **Advances in neural information processing systems**, v. 30, 2017. Cited in page 30.

VEMPRALA, S. et al. Chatgpt for robotics: Design principles and model abilities. **Microsoft Auton. Syst. Robot. Res**, v. 2, p. 20, 2023. Cited in page 26.

WANG, Z. et al. Describe, explain, plan and select: interactive planning with large language models enables open-world multi-task agents. In: **Proceedings of the 37th International Conference on Neural Information Processing Systems**. [S.l.: s.n.], 2023. p. 34153–34189. Cited 2 times in pages 26 and 28.

WEI, J. et al. Chain-of-thought prompting elicits reasoning in large language models. **Advances in neural information processing systems**, v. 35, p. 24824–24837, 2022. Cited 2 times in pages 12 and 34.

XIONG, H. et al. Natural language based context modeling and reasoning with llms: A tutorial. **arXiv preprint arXiv:2309.15074**, 2023. Cited in page 25.

XU, M. et al. **Creative Robot Tool Use with Large Language Models**. 2023. Cited 2 times in pages 19 and 26.

YAO, S. et al. Tree of thoughts: Deliberate problem solving with large language models. **Advances in neural information processing systems**, v. 36, p. 11809–11822, 2023. Cited 2 times in pages 12 and 35.

ZHANG, C. et al. Proagent: building proactive cooperative agents with large language models. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2024. v. 38, n. 16, p. 17591–17599. Cited 2 times in pages 19 and 26.

ZHAO, P.; JIN, Z.; CHENG, N. An in-depth survey of large language model-based artificial intelligence agents. **arXiv preprint arXiv:2309.14365**, 2023. Cited in page 19.

ZHAO, W. X. et al. A survey of large language models. **arXiv preprint arXiv:2303.18223**, v. 1, n. 2, 2023. Cited in page 31.

ZHENG, L. et al. Judging llm-as-a-judge with mt-bench and chatbot arena. **Advances in Neural Information Processing Systems**, v. 36, p. 46595–46623, 2023. Cited 2 times in pages 37 and 39.

ZHENG, Z.; WANG, M.; JIA, B. T. Z. **LangSuit E: Controlling, Planning, and Interacting with Large Language Models in Embodied Text Environments**. GitHub, 2023. Disponível em: https://github.com/bigai-nlco/langsuite. Cited 4 times in pages 19, 24, 28, and 41.

# 8
# Appendix

This appendix presents supplementary material that supports the understanding of this research. It includes additional information, such as prompts and examples that complement the main content of the thesis.

## 8.1
## Prompts

In this section, we show additional prompts used in some of our experiments. We start with the prompt used in the Judge Model.

```
Hello, I need you to act as a judge and rate an action plan between 0 and 1 by its
similarity with a reference action plan. The closer to 1, the more similar the
plans are. When an action has the following format: get(spoon|fork, drawer, hand)
it means that either getting spoon or a fork is correct. Do not penalize the
user's plan on optional actions. Do not penalize the user's plan on actions that
are irrelevant to the task or for redundant actions.

        Try to base your rating on the percentage of actions in the reference plan
that the user's got right. The goal is to verify if the core sequence and intent
of the actions remain consistent with the reference plan. Take into account the
objects in the environment. Do not penalize the user for not being able to access
objects mentioned on the reference plan that are not in its environment.

        Be reasonable with equivalent actions. If on the reference plan the action
was 'put(purse,table,hand)', but the user did 'put(purse,sofa,hand)', do not
penalize the user's action as it was feasible and equivalent to the one on the
reference plan. Disregard actions from the reference plan that do not make sense
with the way the user decided to follow in executing the plan.

    Disregard slight variations in object placement that do not impact the
feasibility of the user's plan. Your final answer should be within .xml tags, for
example: <score>0.97</score>. Be concise with your answer and output only the
final score.
```

Figure 8.1: System Prompt used in our LLM Judge for generating similarity scores.

**When Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of capabilities and choose which objects that surround me I can use to solve those tasks. Those are my capabilities:**

1. **blend**: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format blend(target,tool). For example: **blend(coffee,spoon).**
2. **clean**: to clean targets. For example: clean hand. Input should be in the format clean(target). For example: **clean(hand).**
3. **close**: to close targets. For example: close drawer. Input should be in the format close(target). For example: **close(drawer).**
4. **cook:** to cook a target on tool or location. For example: cook a meat on a pan. Input should be in the format cook(target,tool). For example: **cook(meat,pan).**
5. **cut**: to cut targets with tools. For example: cut tomato with knife. Input should be in the format cut(target,tool). For example: **cut(tomato,knife).**
6. **drink**: to drink targets with tools. For example: drink milk with cup. Input should be in the format drink(target,tool). For example: **drink(milk,cup).**
7. **eat**: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format eat(target,tool). For example: **eat(sandwich,hand).**
8. **fill**: to fill targets with tools/something. For example: fill tank with water. Input should be in the format fill(target,tool). For example: **fill(tank,water).**
9. **get**: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input should be in the format get(target,location, tool). For example: **get(spoon|fork,drawer,hand).**
10. **open**: to open targets. For example: open closet. Input should be in the format open(target). For example: **open(closet).**
11. **play**: play targets with tools. For example: play game-console with controller. Input should be in the format play(target,tool). For example: **play(game-console(game),controller).**
12. **pour**: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in the format pour(target,location,tool). For example: **pour(milk,cup,hand)**
13. **put**: put targets to location with tools. For example: put meat to pan with fork. Input should be in the format put(target,location,tool). For example: **put(meat,pan,fork).**
14. **sweep:** to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the format sweep(target, tool). For example: **sweep(floor, vacuum).**
15. **switch:** to switch targets. For example: switch TV. Input should be in the format switch(target). For example: **switch(TV).**
16. **throw**: to throw targets into location. For example: throw wrapping into trash-can. Input should be in the format throw(target,location). For example: **throw(wrapping,trash-can).**
17. **turn-off:** to turn off targets with tools. For example: turn off TV with remote. Input should be in the format turn-off(target,tool). For example: **turn-off(TV,remote).**
18. **turn-on**: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the format turn-on(target,tool). For example: **turn-on(microwave,hand).**
19. **watch**: to watch targets. For example: watch TV. Input should be in the format watch(target). For example: **watch(TV).**
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example: **wash(cup).**

**Capabilities of the agent**

When asked to complete a task, the final output plan should be in xml tags, like the one shown below.
**Output**:
<plan><action>get(target,location,tool)</action><action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>

**Instructions for output formatting**

Your task is to **drink water**. Here are things to know before you start: the items you need for this task are mentioned in the environment map. Keep the area tidy, and return any used items to their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser. Additional Information: cup at/in/on table. Use the capabilities and objects informed.

**Task Request**

Figure 8.2: Zero-Shot prompt used in our experiments.

**When Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of capabilities and choose which objects that surround me I can use to solve those tasks. Those are my capabilities:**

1. **blend**: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format blend(target,tool). For example: **blend(coffee,spoon)**.
2. **clean**: to clean targets. For example: clean hand. Input should be in the format clean(target). For example: **clean(hand)**.
3. **close**: to close targets. For example: close drawer.  Input should be in the format close(target). For example: **close(drawer)**.
4. **cook:** to cook a target on tool or location. For example: cook a meat on a pan. Input should be in the format cook(target,tool). For example: **cook(meat,pan).**
5. **cut**: to cut targets with tools. For example: cut tomato with knife. Input should be in the format cut(target,tool). For example: **cut(tomato,knife)**.
6. **drink**: to drink targets with tools. For example: drink milk with cup. Input should be in the format drink(target,tool). For example: **drink(milk,cup)**.
7. **eat**: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format eat(target,tool). For example: **eat(sandwich,hand)**.
8. **fill**: to fill targets with tools/something. For example: fill tank with water. Input should be in the format fill(target,tool). For example: **fill(tank,water)**.
9. **get**: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input should be in the format get(target,location, tool). For example: **get(spoon|fork,drawer,hand)**.
10. **open**: to open targets. For example: open closet. Input should be in the format open(target). For example: **open(closet)**.
11. **play**: play targets with tools. For example: play game-console with controller. Input should be in the format play(target,tool). For example: **play(game-console(game),controller)**.
12. **pour**: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in the format pour(target,location,tool). For example: **pour(milk,cup,hand)**
13. **put**: put targets to location with tools. For example: put meat to pan with fork. Input should be in the format put(target,location,tool). For example: **put(meat,pan,fork)**.
14. **sweep:** to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the format sweep(target, tool). For example: **sweep(floor, vacuum)**.
15. **switch:** to switch targets. For example: switch TV. Input should be in the format switch(target). For example: **switch(TV)**.
16. **throw**: to throw targets into location. For example: throw wrapping into trash-can. Input should be in the format throw(target,location). For example: **throw(wrapping,trash-can)**.
17. **turn-off:** to turn off targets with tools. For example: turn off TV with remote. Input should be in the format turn-off(target,tool). For example: **turn-off(TV,remote)**.
18. **turn-on**: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the format turn-on(target,tool). For example: **turn-on(microwave,hand)**.
19. **watch**: to watch targets. For example: watch TV. Input should be in the format watch(target). For example: **watch(TV)**.
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example: **wash(cup)**.

**Capabilities of the agent**

When asked to complete a task, the final output plan should be in xml tags, like the one shown below.
**Output**:
<plan><action>get(target,location,tool)</action><action>fill(target,thing)</action><action>put(targets,location,tools)</action></plan>

**Instructions for output formatting**

For example, let's say you receive the following request:
**Input**:
You are a person and have one task to complete: put winebottle on shelf. Use the capabilities defined earlier. Use the objects and tools listed below.
Objects and tools: winebottle, countertop, shelf, cabinet, coffeemachine. Winebottle is over the countertop, coffeemachine is inside the cabinet.
The final output plan should be in xml tags, like the one shown below.
**Output**:
<plan><action>get(winebottle,countertop,hand)</action><action>put(winebottle,shelf,hand)</action></plan>

**Few-Shot Prompt**

Your task is to **drink water**. Here are things to know before you start: the items you need for this task are mentioned in the environment map.  Keep the area tidy, and return any used items to their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser. Additional Information: cup at/in/on table. Use the capabilities and objects informed.

**Task Request**

Figure 8.3: Few-Shot prompt with 1 example used in our experiments.

**When Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of capabilities and choose which objects that surround me I can use to solve those tasks. Those are my capabilities:**

1. **blend**: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format blend(target,tool). For example: **blend(coffee,spoon).**
2. **clean**: to clean targets. For example: clean hand. Input should be in the format clean(target). For example: **clean(hand).**
3. **close**: to close targets. For example: close drawer. Input should be in the format close(target). For example: **close(drawer).**
4. **cook:** to cook a target on tool or location. For example: cook a meat on a pan. Input should be in the format cook(target,tool). For example: **cook(meat,pan).**
5. **cut**: to cut targets with tools. For example: cut tomato with knife. Input should be in the format cut(target,tool). For example: **cut(tomato,knife).**
6. **drink**: to drink targets with tools. For example: drink milk with cup. Input should be in the format drink(target,tool). For example: **drink(milk,cup).**
7. **eat**: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format eat(target,tool). For example: **eat(sandwich,hand).**
8. **fill**: to fill targets with tools/something. For example: fill tank with water. Input should be in the format fill(target,tool). For example: **fill(tank,water).**
9. **get**: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input should be in the format get(target,location, tool). For example: **get(spoon|fork,drawer,hand).**
10. **open**: to open targets. For example: open closet. Input should be in the format open(target). For example: **open(closet).**
11. **play**: play targets with tools. For example: play game-console with controller. Input should be in the format play(target,tool). For example: **play(game-console(game),controller).**
12. **pour**: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in the format pour(target,location,tool). For example: **pour(milk,cup,hand)**
13. **put**: put targets to location with tools. For example: put meat to pan with fork. Input should be in the format put(target,location,tool). For example: **put(meat,pan,fork).**
14. **sweep:** to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the format sweep(target, tool). For example: **sweep(floor, vacuum).**
15. **switch:** to switch targets. For example: switch TV. Input should be in the format switch(target). For example: **switch(TV).**
16. **throw:** to throw targets into location. For example: throw wrapping into trash-can. Input should be in the format throw(target,location). For example: **throw(wrapping,trash-can).**
17. **turn-off:** to turn off targets with tools. For example: turn off TV with remote. Input should be in the format turn-off(target,tool). For example: **turn-off(TV,remote).**
18. **turn-on**: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the format turn-on(target,tool). For example: **turn-on(microwave,hand).**
19. **watch**: to watch targets. For example: watch TV. Input should be in the format watch(target). For example: **watch(TV).**
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example: **wash(cup).**

**Capabilities of the agent**

When asked to complete a task, the final output plan should be in xml tags, like the one shown below.
<plan><action>get(target,location,tool)</action><action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>

**Instructions for output formatting**

Your task is to **drink water**. Here are things to know before you start: the items you need for this task are mentioned in the environment map. Keep the area tidy, and return any used items to their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser. Additional Information: cup at/in/on table. Use the capabilities and objects informed. Think step by step.

**Task Request and Zero-Shot Chain of Thought Prompt**

Figure 8.4: Zero-Shot Chain of Thought prompt used in our experiments.

**When Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of capabilities and choose which objects that surround me I can use to solve those tasks. Those are my capabilities:**

1. **blend**: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format blend(target,tool). For example: **blend(coffee,spoon)**.
2. **clean**: to clean targets. For example: clean hand. Input should be in the format clean(target). For example: **clean(hand)**.
3. **close**: to close targets. For example: close drawer. Input should be in the format close(target). For example: **close(drawer)**.
4. **cook:** to cook a target on tool or location. For example: cook a meat on a pan. Input should be in the format cook(target,tool). For example: **cook(meat,pan)**.
5. **cut**: to cut targets with tools. For example: cut tomato with knife. Input should be in the format cut(target,tool). For example: **cut(tomato,knife)**.
6. **drink**: to drink targets with tools. For example: drink milk with cup. Input should be in the format drink(target,tool). For example: **drink(milk,cup)**.
7. **eat**: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format eat(target,tool). For example: **eat(sandwich,hand)**.
8. **fill**: to fill targets with tools/something. For example: fill tank with water. Input should be in the format fill(target,tool). For example: **fill(tank,water)**.
9. **get**: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input should be in the format get(target,location, tool). For example: **get(spoon|fork,drawer,hand)**.
10. **open**: to open targets. For example: open closet. Input should be in the format open(target). For example: **open(closet)**.
11. **play**: play targets with tools. For example: play game-console with controller. Input should be in the format play(target,tool). For example: **play(game-console(game),controller)**.
12. **pour**: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in the format pour(target,location,tool). For example: **pour(milk,cup,hand)**
13. **put**: put targets to location with tools. For example: put meat to pan with fork. Input should be in the format put(target,location,tool). For example: **put(meat,pan,fork)**.
14. **sweep:** to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the format sweep(target, tool). For example: **sweep(floor, vacuum)**.
15. **switch:** to switch targets. For example: switch TV. Input should be in the format switch(target). For example: **switch(TV)**.
16. **throw:** to throw targets into location. For example: throw wrapping into trash-can. Input should be in the format throw(target,location). For example: **throw(wrapping,trash-can)**.
17. **turn-off:** to turn off targets with tools. For example: turn off TV with remote. Input should be in the format turn-off(target,tool). For example: **turn-off(TV,remote)**.
18. **turn-on**: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the format turn-on(target,tool). For example: **turn-on(microwave,hand)**.
19. **watch**: to watch targets. For example: watch TV. Input should be in the format watch(target). For example: **watch(TV)**.
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example: **wash(cup)**.

**Capabilities of the agent**

When asked to complete a task, the final output plan should be in xml tags, like the one shown below.
**Output**:
<plan><action>get(target,location,tool)</action><action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>

**Instructions for output formatting**

For example, let's say you receive the following request:
**Input**:
You are a person and have one task to complete: put winebottle on shelf. Use the capabilities defined earlier. Use the objects and tools listed below.
Objects and tools: winebottle, countertop, shelf, cabinet, coffeemachine. Winebottle is over the countertop, coffeemachine is inside the cabinet.
To complete this task, I want you to think step by step. The final output plan should be in xml tags, like the one shown below.

**Output**:
1 - First, I have to get the winebottle.
2 - After that, I need to go to the shelf.
3 - Finally, I need to place the winebottle on the shelf.
<plan><action>get(winebottle,countertop,hand)</action><action>put(winebottle,shelf,hand)</action></plan>

**One-Shot Chain of Thought Prompt**

Your task is to **drink water**. Here are things to know before you start: the items you need for this task are mentioned in the environment map. Keep the area tidy, and return any used items to their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser. Additional Information: cup at/in/on table. Use the capabilities and objects informed. Think step by step.

**Task Request**

Figure 8.5: Chain of Thought prompt with 1 example used in our experiments.

**When Hello, I want you to help me plan how to tackle tasks. Help me complete tasks with actions from my pool of capabilities and choose which objects that surround me I can use to solve those tasks. Those are my capabilities:**

1. **blend**: to blend targets with tools. For example: blend coffee with spoon. Input should be in the format blend(target,tool). For example: **blend(coffee,spoon)**.
2. **clean**: to clean targets. For example: clean hand. Input should be in the format clean(target). For example: **clean(hand)**.
3. **close**: to close targets. For example: close drawer. Input should be in the format close(target). For example: **close(drawer)**.
4. **cook:** to cook a target on tool or location. For example: cook a meat on a pan. Input should be in the format cook(target,tool). For example: **cook(meat,pan)**.
5. **cut**: to cut targets with tools. For example: cut tomato with knife. Input should be in the format cut(target,tool). For example: **cut(tomato,knife)**.
6. **drink**: to drink targets with tools. For example: drink milk with cup. Input should be in the format drink(target,tool). For example: **drink(milk,cup)**.
7. **eat**: to eat targets with tools. For example: eat sandwich with hand. Input should be in the format eat(target,tool). For example: **eat(sandwich,hand)**.
8. **fill**: to fill targets with tools/something. For example: fill tank with water. Input should be in the format fill(target,tool). For example: **fill(tank,water)**.
9. **get**: to get targets from location with tools. For example: get spoon, fork from drawer using hand. Input should be in the format get(target,location, tool). For example: **get(spoon|fork,drawer,hand)**.
10. **open**: to open targets. For example: open closet. Input should be in the format open(target). For example: **open(closet)**.
11. **play**: play targets with tools. For example: play game-console with controller. Input should be in the format play(target,tool). For example: **play(game-console(game),controller)**.
12. **pour**: pour targets into location with tools. For example: pour milk into cup with hand. Input should be in the format pour(target,location,tool). For example: **pour(milk,cup,hand)**
13. **put**: put targets to location with tools. For example: put meat to pan with fork. Input should be in the format put(target,location,tool). For example: **put(meat,pan,fork)**.
14. **sweep:** to sweep targets or an area with tools. For example: sweep floor with vacuum. Input should be in the format sweep(target, tool). For example: **sweep(floor, vacuum)**.
15. **switch:** to switch targets. For example: switch TV. Input should be in the format switch(target). For example: **switch(TV)**.
16. **throw:** to throw targets into location. For example: throw wrapping into trash-can. Input should be in the format throw(target,location). For example: **throw(wrapping,trash-can)**.
17. **turn-off:** to turn off targets with tools. For example: turn off TV with remote. Input should be in the format turn-off(target,tool). For example: **turn-off(TV,remote)**.
18. **turn-on**: to turn on targets with tools. For example: turn on microwave with hand. Input should be in the format turn-on(target,tool). For example: **turn-on(microwave,hand)**.
19. **watch**: to watch targets. For example: watch TV. Input should be in the format watch(target). For example: **watch(TV)**.
20. wash: to wash targets. For example: wash cup. Input should be in the format wash(target). For example: **wash(cup)**.

**Capabilities of the agent**

When asked to complete a task, the final output plan should be in xml tags, like the one shown below.
**Output**:
<plan><action>get(target,location,tool)</action><action>fill(target,thing)</action>
<action>put(targets,location,tools)</action></plan>

**Instructions for output formatting**

For example, let's say you receive the following request:
**Input**:
You are a person and have one task to complete: put winebottle on shelf. Use the capabilities defined earlier. Use the objects and tools listed below.
Objects and tools: winebottle, countertop, shelf, cabinet, coffeemachine. Winebottle is over the countertop, coffeemachine is inside the cabinet.
To complete this task, I want you to think step by step. The final output plan should be in xml tags, like the one shown below.

**Output**:
1 - First, I have to get the winebottle.
2 - After that, I need to go to the shelf.
3 - Finally, I need to place the winebottle on the shelf.
<plan><action>get(winebottle,countertop,hand)</action><action>put(winebottle,shelf,hand)</action></plan>

**One-Shot Chain of Thought Prompt**

Your task is to **drink water**. Here are things to know before you start: the items you need for this task are mentioned in the environment map. Keep the area tidy, and return any used items to their original place after completing the task.

Objects in the environment: cup, table, water, water-dispenser. Additional Information: cup at/in/on table. Use the capabilities and objects informed. Think step by step.

**Task Request**

Figure 8.6: Tree of Thoughts prompt used in our experiments.