UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

Final Dissertation

Exploring the Use of LLMs for Agent Planning: Strengths and Weaknesses

Supervisor
Paolo Giorgini

Student
Davide Modolo
229297

Academic year 2023/2024

Contents

Abstract

1 Introduction

This thesis explores the capabilities of Large Language Models (LLMs) in the context of a logistics problem. Artificial intelligence has made significant strides in generative systems, particularly with the advent of LLMs, which are capable of producing coherent and contextually relevant text based on input prompts. However, their ability to autonomously plan and achieve goals without additional external structures remains a topic of investigation. The primary aim of this work is to assess whether LLMs can be effectively utilized as agents in dynamic environments without leveraging predefined frameworks or knowledge bases.

To achieve this, we first conducted an analysis of existing methodologies in the literature. Traditional approaches such as PDDL and Reinforcement Learning provide structured and systematic ways to tackle planning problems. PDDL offers explainability and efficiency in constrained environments but lacks adaptability, making it impractical for real-time applications. On the other hand, Reinforcement Learning is highly adaptable and effective in changing environments but suffers from issues such as convergence to local optima and lack of explainability.

Recent research has also explored planning capabilities of LLMs, with several studies investigating their reasoning abilities as well as planning skills.

The specific problem addressed in this thesis involves evaluating the generative ability of "raw" LLMs, devoid of external structures, to solve logistics problems in dynamic settings. [TODO MAKE THIS LONGER]

We also analyzed some ways to evaluate the LLM uncertainty and used the log-probability based one to evaluate the uncertainty of an LLM in choosing the action. The approach required to generate a token that referred to a possible action and working on a bias for the logits to then compute the probability of correct.

[EXPECTED RESULTS]

The thesis is structured as follows:

- Chapter 2 provides an overview of the background and related work.
- Chapter 3 describes the experimental setting and the environment used.
- Chapter 4 details the development of the agent.
- Chapter 5 explains the data collection process.
- Chapter 6 presents the results and discusses the findings.
- Chapter 7 outlines potential future works.
- Chapter 8 concludes the thesis.

2 Background

In this thesis, we will analyze in detail the behavior of a Large Language Model (LLM) as an agent within a controlled environment performing a logistics task.

Before presenting all the work carried out in detail, this chapter aims to provide a comprehensive explanation of all the theoretical foundations necessary to understand the steps presented in the following chapters. Starting from a brief introduction of Artificial Intelligence (AI) just to define the boundaries in which we are working, we will move to the core concepts. In particular, we want to highlight what an LLM is and how it works, with a special focus on the Attention mechanism and how the uncertainty of an LLM can be calculated. This will serve as a basis for correctly interpreting the results analyzed in Section 6.

There will also be a broader discussion on agents in a strict sense and *LLM agents* to better show the difference between our implementation and what is currently being discussed in the literature.

To better define the context of this thesis, we will also examine the main alternative approaches to solving a logistics problem currently studied in the literature.

2.1 Artificial Intelligence

Artificial Intelligence is a very broad field, that can be resumed as systems designed to perform tasks that traditionally require intelligence, such as *Natural Language Understanding*, visual perception, decision-making, and problem-solving.

In recent years, AI has rapidly evolved, driven by advances in deep learning[1], increased computational power, and the easy availability of massive datasets (language models are even trained on the entirety of the internet). Early AI systems, including expert systems and early machine learning models, relied on manually crafted rules or statistical techniques. However, with the rise of neural networks, particularly deep learning models, AI has shifted toward self-learning systems capable of extracting complex patterns from raw data.

One of the key breakthroughs in this evolution was the development of deep neural networks (DNNs), particularly Convolutional Neural Networks (CNNs) for image processing, introduced by Krizhevsky et al. [14] and Recurrent Neural Networks (RNNs) for sequential data, including language modeling, introduced by Hochreiter et al. [10].

Despite their success, RNNs struggled with long-term dependencies due to vanishing gradients[2], leading to the development of the *Transformer architecture* (from Vaswani et al., Attention Is All You Need [27]), which eliminated recurrence in favor of self-attention mechanisms, significantly improving efficiency and scalability in natural language processing. This shift enabled the emergence of large-scale AI models, particularly in NLP, where two main categories can be defined: discriminative models and generative models.

Discriminative models Discriminative models are a class of machine learning models that aim to directly model the decision boundary between different classes in a dataset. Unlike generative models, which learn the underlying distribution of the data, discriminative models focus on learning the conditional probability of a target class given the input features. Classical models like Support Vector Machines[3] and Conditional Random Fields[4] have been widely used for text classification and sequence labeling tasks such as Named Entity Recognition (Lafferty et al. [15]). More recently, deep learning-based models like BERT (Devlin et al. [6]) have been invented, that leverage contextualized

[1]`https://en.wikipedia.org/wiki/Deep_learning`
[2]`https://en.wikipedia.org/wiki/Vanishing_gradient_problem`
[3]`https://en.wikipedia.org/wiki/Support_vector_machine`
[4]`https://en.wikipedia.org/wiki/Conditional_random_field`

word representations to improve performance on tasks like sentiment analysis, intent detection, and slot filling.

Generative models Generative models learn the underlying data distribution to create new samples that resemble the original data. This category includes several architectures that have pushed the boundaries of AI-generated content. Variational Autoencoders (Kingma and Welling [13]) introduced a probabilistic approach to generating structured data, while Generative Adversarial Networks (Goodfellow et al. [8]) refined the concept by using two competing neural networks, a generator and a discriminator, to iteratively improve synthetic data generation. More recently, diffusion models (Ho et al. [9]) have surpassed GANs in generating high-quality images by modeling data transformations through iterative denoising processes. In the domain of text generation, autoregressive models like GPT (Radford et al. [20]) demonstrated the power of large-scale, unsupervised pretraining. These Large Language Models predict the next token (that can be seen as a building block of a word, approximately a syllable) in a sequence based on vast amounts of textual data, learning contextual nuances and producing human-like responses.

2.2 Large Language Models

Large Language Models (LLMs) are a class of deep learning models that leverage the transformer architecture to generate coherent and contextually relevant text. These models have revolutionized natural language processing by achieving state-of-the-art performance on a wide range of tasks, including language modeling, translation, summarization, and question-answering.

The transformer architecture, introduced by Vaswani et al. in the paper 'Attention Is All You Need' [27], is the foundation of LLMs. It consists of an encoder-decoder structure, where the encoder processes the input sequence and generates a sequence of hidden states, while the decoder generates the output sequence based on the encoder's hidden states. The key innovation in transformers is the self-attention mechanism, which allows the model to weight the importance of different input tokens when generating the output. This mechanism enables transformers to capture long-range dependencies and contextual information more effectively than RNNs.

In this thesis, we will focus on the models built by OpenAI, we will analyze them more in depth in Section 3.3.

2.2.1 Attention Mechanism

The attention mechanism is a fundamental component of the transformer architecture (Figure 2.1), enabling the model to focus on specific parts of the input sequence when generating the output. The attention mechanism computes a weighted sum of the input tokens, where the weights are learned during training based on the relevance of each token to the current context.

The self-attention mechanism works in this way:

1. create 3 vectors from embeddings (Query, Key, Value) multiplying by 3 matrices learned during the training process;

2. calculate a score that determines how much focus goes to different parts of the input sentence as it encodes a word;

3. divide the score for more stable gradients and apply softmax;

4. multiply each value vector by the softmax score to keep the value of the word it focuses on, and sink other irrelevant words;

5. sum the weighted value vectors: this produces the output of the self-attention layer at this position.

The self-attention operation computes the relevance of each token in the input with respect to the query token using the scaled dot-product attention:

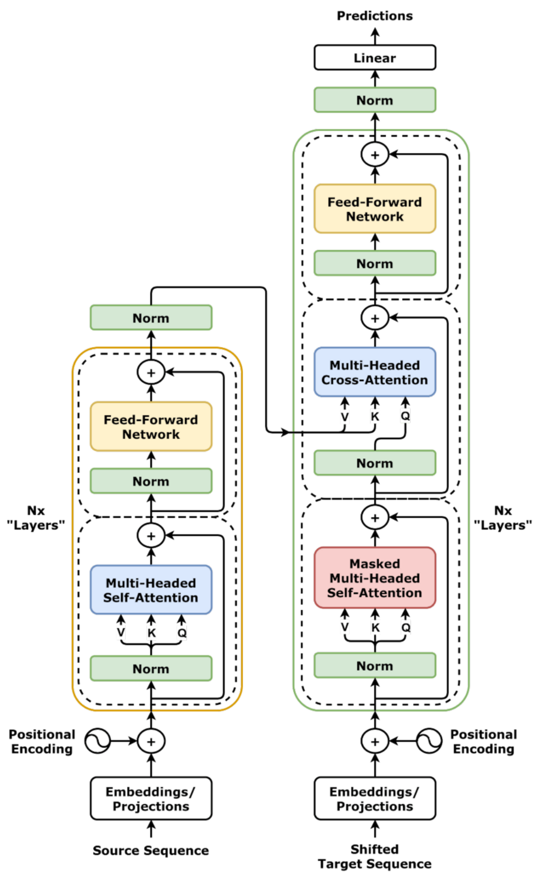


Figure 2.1: Transformer Architecture
Source: Vaswani et al., Attention Is All You Need [27]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimensionality of the key vectors, ensuring that the dot products do not grow too large as input size increases. The softmax function normalizes the scores into attention weights, which determine how much influence each token should have on the final representation.

Multi-head attention extends this mechanism by computing multiple sets of Q, K, V matrices in parallel, allowing the model to capture different aspects of contextual relationships:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each attention head independently applies scaled dot-product attention, and the outputs are concatenated and linearly projected using W^O (Weight matrix). This improves the model's ability to encode complex dependencies and contextual meaning.

The attention mechanism allows the model to focus on different parts of the input sequence based on the current context, enabling it to capture long-range dependencies and improve performance on tasks like text generation.

2.2.2 LLMs' Uncertainty

Despite their impressive capabilities, LLMs are inherently probabilistic and can generate responses that are syntactically correct yet factually inaccurate. Understanding and quantifying this uncertainty is crucial for evaluating the reliability of generated text, especially in high-stakes applications such as medical diagnosis, legal advice, or automated decision-making.

For example, if an LLM generates an answer to a yes/no question with probabilities:

$$P(\text{Yes}) = 0.51, P(\text{No}) = 0.49$$

then the model is nearly uncertain, and this information should be communicated rather than presenting “Yes” as a definitive response.

A key consequence of uncertainty is the phenomenon of *hallucination*, where the model generates confident but factually incorrect or fabricated information [12]. Hallucinations arise when:

- the model lacks knowledge about a specific query but still generates an answer;
- the training data contains conflicting or misleading patterns;
- the model overgeneralizes from limited training examples.

Mitigating hallucinations involves uncertainty-aware generation techniques, the most common one is *Retrieval-Augmented Generation* (RAG) [16], which enhances the prompt with additional context from a knowledge base to improve the model's factual accuracy.

The literature studied different approaches to quantify uncertainty in LLMs, and this thesis will use one of the most common methods to quantify the probability of correctness in the generated choice by the agent.

2.2.2.1 Expressing Uncertainty

A study titled ‘Can LLMs Express Their Uncertainty? An Empirical Evaluation of Confidence Elicitation in LLMs’ [30] investigates methods for eliciting confidence from LLMs without accessing their internal parameters or fine-tuning. The researchers propose a framework comprising three components:

- Prompting Strategies: Techniques to elicit verbalized confidence from the model;
- Sampling Methods: Generating multiple responses to assess variability;
- Aggregation Techniques: Computing consistency across responses to determine confidence levels.

The study evaluates these methods on tasks such as confidence calibration and failure prediction across various datasets and LLMs, including GPT-4 and LLaMA 2 Chat.

Key findings indicate that LLMs often exhibit overconfidence when verbalizing their certainty, possibly mirroring human confidence expression patterns. Additionally, as model capabilities increase, both calibration and failure prediction performance improve, though they remain suboptimal. They show that implementing strategies like human-inspired prompts and assessing consistency among multiple responses can mitigate overconfidence. Notably, while methods requiring internal model access perform better, the performance gap is narrow.

2.2.2.2 Stable Explanations as Confidence Measures

In the pursuit of reliable uncertainty quantification in Large Language Models, the paper ‘Cycles of Thought: Measuring LLM Confidence through Stable Explanations ’ [1] introduced a novel framework that assesses model confidence through the stability of generated explanations.

Their approach posits that the consistency of explanations accompanying an answer can serve as a proxy for the model's certainty. Instead of assigning a single probability to an answer, the method generates multiple explanations for the same question and treats each explanation-answer pair as a distinct classifier. A posterior distribution is then computed over these classifiers, allowing for a principled estimation of confidence based on explanation stability. If the model's explanations remain stable across different reasoning paths, it suggests high confidence in the answer. Conversely, significant variation in explanations signals uncertainty. Empirical evaluations across multiple datasets demonstrated that this framework enhances confidence calibration and failure prediction, outperforming traditional baselines.

However, there are some potential drawbacks. The method requires generating multiple explanations, which increases computational cost and latency. Additionally, it can be sensitive to prompt variations, and may misinterpret repetitive patterns as high confidence.

2.2.2.3 Tokens' log-probability

The paper 'Robots That Ask For Help: Uncertainty Alignment for Large Language Model Planners' [24] introduces the KnowNo framework, which is the one we took inspiration from, to quantify the uncertainty of the agent in this thesis.

The KnowNo framework leverages Conformal Prediction (CP)[5], a statistical method that provides formal guarantees on the reliability of predictions to assess uncertainty.

In the paper, they ask the LLM to generate a set of four actions for a given prompt (since the `logit_bias` parameter in the OpenAI API was limited to five tokens at that time, more on this in Section 3.3.2), and then they append a "no-op" action to the set. This will not be the case of this thesis, since the actions will always be the same, but we will use the same math behind the uncertainty calculation.

Then, they ask the model for the action to select, adding the bias to the tokens representing each action. Here they use the log-probabilities of the tokens (referring to the actions) to compute the uncertainty.

KnowNo computes uncertainty evaluating the "validity" of each option: CP calculates a confidence interval based on previous data, and from this, a set of valid actions is generated (based on their scaled log-probability). This set can include one or more actions, and the size of this set is indicative of the level of uncertainty:

- **Singleton**: If CP narrows down the options to just one action, this indicates low uncertainty, and the robot can proceed confidently with the task. The model is highly certain that this action is the most appropriate next step.
- **Multiple Options**: When CP leaves multiple possible actions in the valid set, this may indicate high uncertainty. In such cases, KnowNo triggers the robot to request human assistance. This allows the robot to seek clarification when it is unsure, thereby avoiding errors that might arise from acting on uncertain predictions.

A simplified version of the KnowNo flow can be seen in Figure 2.2. Technically speaking, the computation of the uncertainty can be summarize in 5 steps:

1. give each action a single-token label (eg. `A)`, `B)`, `C)`, `D)`, `E)`);
2. use the `logit_bias` parameter in the API to force the model to only answer using these labels;
3. get the log-probabilities of the tokens and scale them: this results in a "confidence" value for each token;
4. filter the resulting set of option with a threshold computed with CP;
5. either the result will be a singleton (no uncertainty) or a set of options.

They also say that the framework has the advantage of being model-agnostic, as it can be applied to LLMs out-of-the-box without requiring any fine-tuning, thanks to the "caution" that is given if the resulting filtered set of options is not a singleton.

2.3 Agents

As widely explained in the book 'An Introduction to Multiagent Systems' [29], we can summarize the definition of an agent as an autonomous entity that perceives its environment through sensors and acts upon it through effectors, making decisions based on its perceptions and objectives in order to achieve specific goals.

This definition highlights several key aspects of agents:

- Autonomy: Agents operate without direct human intervention, controlling their own actions.

[5] `https://en.wikipedia.org/wiki/Conformal_prediction`

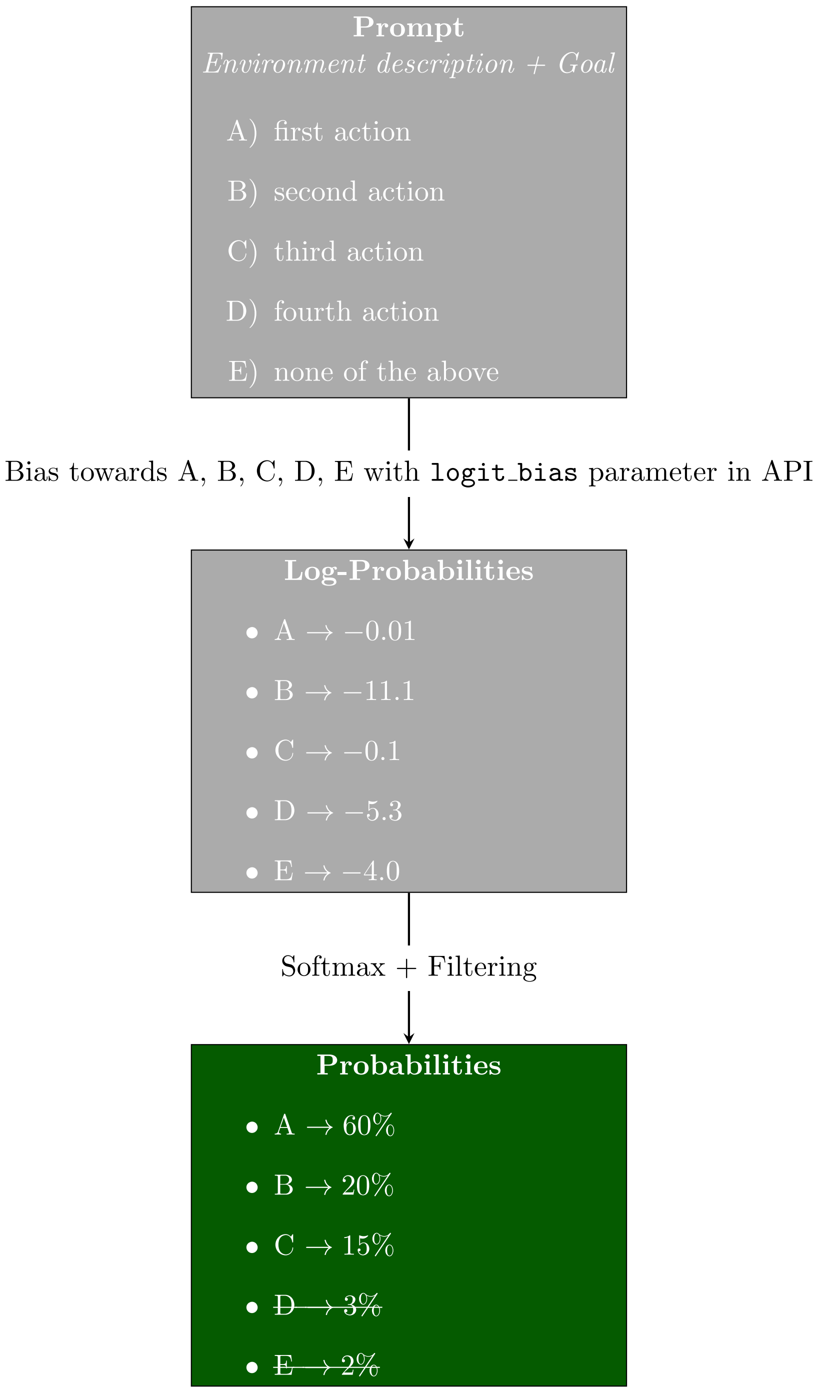


Figure 2.2: KnowNo Uncertainty Computation

- Perception and Action: They interact with the environment via sensors (perception) and actuators (action execution).
- Decision-making: Agents select actions based on their internal model, goals, and the state of the environment.
- Non-determinism and Adaptability: Since environments are generally non-deterministic, agents must be prepared for uncertainty and potential failures in action execution.
- Preconditions and Constraints: Actions are subject to certain conditions that must be met for successful execution.

Thus, an agent's fundamental challenge is deciding which actions to perform in order to best satisfy its objectives, given the constraints and uncertainties of its environment.

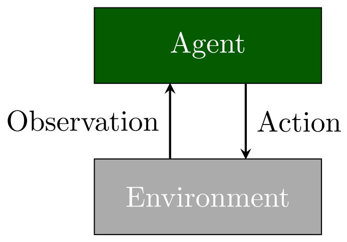


Figure 2.3: Agent Design Scheme
Source: redesign of a scheme in [29]

As shown in Figure 2.3, an agent is some entity that perceives the environment and reacts to it. The setting can be anything from a simple thermostat to a complex system like a self-driving car. The idea is that the agent is able to react to a change in the environment and take actions to achieve its goals.

We will analyze in detail the prompts and the choices in Section 5.2, but to give an some anticipation to align our agent with the definition above, we can map some of its concept to what this thesis will analyze:

- Autonomy: the agent will choose its action based on the prompt built using the environment information only;
- Perception and Action: what the server sends about the current state of the environment can be seen as the perception of the agent, while the action it can take will be given in the prompt in a specific way.
- Decision-making: the decision-making process will be the generation of the text by the LLM, weighted by the uncertainty.
- Non-determinism and Adaptability: to emulate the non-determinism of the environment, the state received by the server will be used "raw" in the prompt, without any hard processing or parsing.
- Preconditions and Constraints: being in a "limited" map with a fixed number of walkable cells, is itself a constraint the agent must consider.

2.3.1 BDI Architecture

The Belief-Desire-Intention (BDI) architecture is a widely adopted framework in artificial intelligence for modeling rational agents. It was formally developed by Rao and Georgeff in 1995 [23] and has been implemented in several architectures, including PRS (1987), dMARS (1998), JAM (1999), Jack (2001), and JADEX (2005). BDI provides a structured approach to practical reasoning, allowing agents to function effectively in dynamic and unpredictable environments.

2.3.1.1 Core Components of BDI

BDI agents operate based on three key components:

- Belief: Represents the agent's knowledge about the world, including past events and observations;
- Desire (Goals): Defines the agent's objectives or preferred end states;
- Intention: Represents the commitments of an agent toward achieving specific goals through selected plans.

BDI has been extensively used in fields like robotics, automated planning, and multi-agent systems.

2.4 State of the Art

A logistic problem is a fundamental challenge in the field of Artificial Intelligence (AI), since depending on the complexity of the specific problem, it can contain tasks such as route optimization, supply chain management, and delivery scheduling. These problems arise in various domains, including transportation, e-commerce, and manufacturing, where efficient resource allocation and decision-making are critical. Given the complexity of modern logistics, AI has emerged as a powerful tool for finding optimal or near-optimal solutions.

Traditional research techniques, such as linear programming and heuristics, have been widely employed. However, with the increasing availability of data and computational power, machine learning (ML) and deep learning methods have become more prevalent. These methods can predict demand, optimize routes dynamically, and enhance decision-making under uncertainty based on the data. Additionally, reinforcement learning (RL) has gained attention for its ability to learn optimal strategies through trial and error, particularly in dynamic and unpredictable environments.

In the recent years with the explosion of Large Language Models (LLMs), many researchers started to apply them to different fields, including planning and logistics.

2.4.1 PDDL Based Solutions

Planning Domain Definition Language (PDDL) is a human-readable format for problems in automated planning that gives a description of the possible states of the world, a description of the set of possible actions, a specific initial state of the world, and a specific set of desired goals.
Source: Wikipedia[6]

The fundamental distinction between a PDDL-based solution and any Machine Learning/Deep Learning approach lies in the very nature of how problems are defined and solved.

In a PDDL-based system, the problem must be explicitly encoded using a formal, structured language that describes the initial state, the goal state, and the set of available actions. This formal encoding serves as a blueprint for the planner, which then performs the computationally intensive task of exploring a vast search space. The planner systematically generates and evaluates possible action sequences, using algorithms to determine an optimal path from the initial state to the goal state. This process is highly deterministic, with each action being considered in the context of its direct impact on reaching the goal.

While effective in structured, static environments with well-defined parameters, this approach is inherently time-consuming and computationally demanding. The planner must traverse a potentially enormous state space, guided by heuristics to prune less relevant possibilities, but still constrained by the rigid formalism of PDDL. Because of this, it can struggle with real-time decision-making, particularly in situations where the environment is dynamic, uncertain, or rapidly changing.

[6]`https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language`

PDDL Code

```
(define (domain bit-toggle)
  (:requirements :strips :negative-preconditions)
  (:predicates
    (bit ?b)                          ; predicate meaning
                                      ; bit ?b is set (true)
  )

  (:action setbit
    :parameters (?b)
    :precondition (not (bit ?b))      ; can only set a bit if
                                      ; it is not already set
    :effect (bit ?b)                  ; setting the bit to true
  )

  (:action unsetbit
    :parameters (?b)
    :precondition (bit ?b)            ; can only unset a bit if
                                      ; it is currently set
    :effect (not (bit ?b))            ; setting the bit to false
  )
)
```

Listing 2.1: Domain file example for a bit toggle problem

PDDL Code

```
(define (problem bit-toggle-full-problem)
  (:domain bit-toggle-full)
  (:objects
     b1 b2 b3
  )
  (:init)                             ; Initially all bits are unset (
   false)

  (:goal                              ; It can be any combination of T/F
     (and (bit b1) (bit b2) (not(bit b3)))
  )
)
```

Listing 2.2: Problem file example for a bit toggle problem

With the increasing number of variables (actions or predicates), the number of arcs and nodes grows exponentially. A little example that makes this problem easy to visualize is the Domain where we can have N possible bits, that can be turned to `true` or `false` (Domain file in Listing 2.1) and the Problem where everything start at `false` and we want a specific final combination (Problem file in Listing 2.2).

As we can see in the plot Figure 2.5, the number of arcs (example of graphs for 2, 3 and 4 bits in Figure 2.4) grows exponentially with the number of bits, as well as the number of states obviously. This shows how even a simple problem with a simple solution can become time-intensive and not

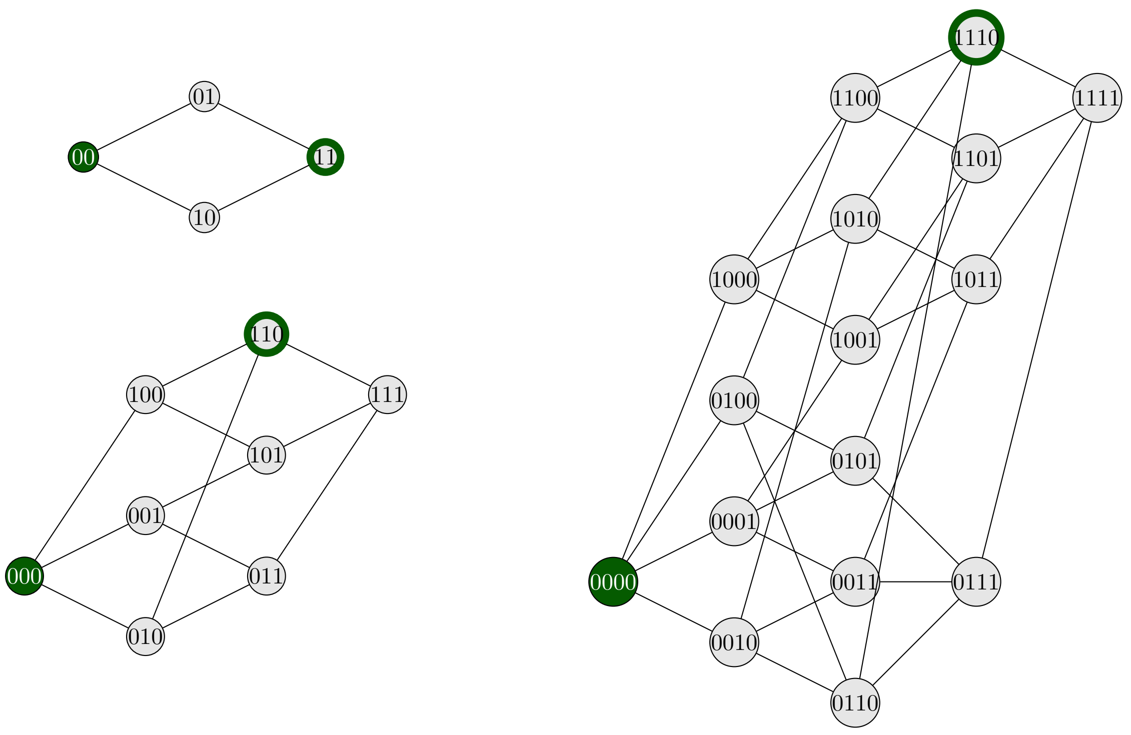
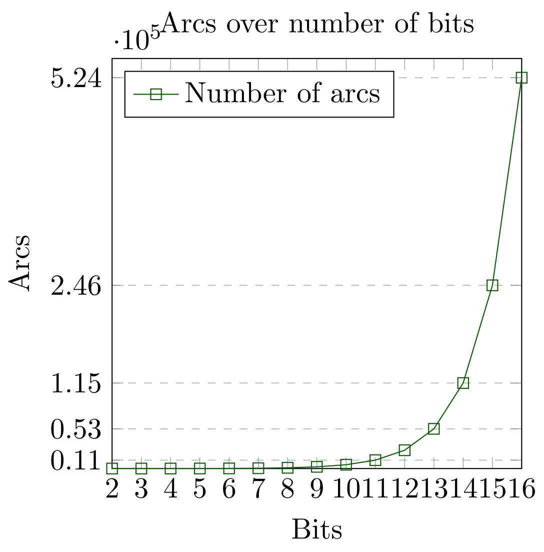
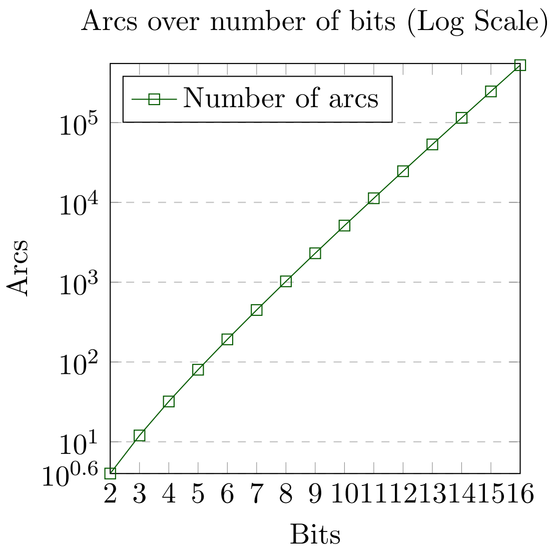


Figure 2.4: Graphs for bit-toggle problem with 2, 3, and 4 bits



(a) label 1



(b) label 1

Figure 2.5: Arcs per Bit

suitable for real-time applications.

PDDL Code

```
           ; Found Plan (output)
(setbit b2)
(setbit b1)
```

Listing 2.3: Plan for the bit toggle problem (`110`), solved by LAMA-first planner

However, a PDDL approach is more explainable, since all the information is provided by the user and the output result is a sequence of actions (example at Listing 2.3). This makes it easier to understand and debug the solution, as each step is explicitly defined. Of course, there might be different paths to reach the goal, and the planner might choose one based on heuristics or optimization criteria. This transparency in the decision-making process is one of the key advantages of using PDDL for planning problems.

Literature An example of a problem related to the one presented in this thesis, solved using PDDL, can be found in the paper 'An AI Planning Approach to Emergency Material Scheduling Using Numerical PDDL' by Yang et al. [31].

In their work, they utilize PDDL 2.1 that allows to model the scheduling problem, incorporating factors such energy consumption constraints. Their approach employs the Metric-FF planner to generate optimized scheduling plans that minimize total scheduling time and transportation energy usage. However, while this demonstrates the applicability of AI planning to emergency logistics, their model simplifies the real-world scenario by assuming predefined transport routes, limited vehicle types, and abstract representations of congestion effects. This highlights a broader limitation of PDDL in capturing the full complexity of dynamic and uncertain environments often encountered in emergency response situations.

2.4.2 Reinforcement Learning Solutions

Reinforcement Learning is a branch of machine learning focused on making decisions to maximize cumulative rewards in a given situation. Unlike supervised learning, which relies on a training dataset with predefined answers, RL involves learning through experience. In RL, an agent learns to achieve a goal in an uncertain, potentially complex environment by performing actions and receiving feedback through rewards or penalties.

Source: GeegksforGeeks [7]

Reinforcement Learning is a learning setting, where the learner is an Agent that can perform a set of actions depending on its state in a set of states and the environment.

It works by defining:

- **Environment**: the world in which the agent operates
- **Agent**: the decision-maker that interacts with the environment
- **Actions**: the possible moves the agent can make
- **Rewards**: the feedback the agent receives for its actions
- **Policy**: the strategy the agent uses to select Actions

In performing action `a` in state `s`, the learner receive an immediate reward `r(s,a)`. In some states, some actions could be not possible or valid.

The task is to learn a policy (a full specification of what action to take at each state) allowing the agent to choose for each state the action maximizing the overall reward, including future moves.

To deal with this delayed reward problem, the agent has to trade-off exploitation and exploration:

- **Exploitation**: the agent chooses the action that it knows will give some reward
- **Exploration**: the agent tries alternative actions that could end in bigger rewards

When considering a logistics problem, reinforcement learning naturally comes to mind. This is because defining a reward function is relatively straightforward: it could be measured in terms of packages delivered per minute, per step, or a similar metric. Additionally, the entire process can be simulated in a virtual environment, allowing multiple parallel simulations to accelerate the agent's learning process. As illustrated in Figure 2.6, the structure of the Reinforcement Learning framework

[7] `https://www.geeksforgeeks.org/what-is-reinforcement-learning/`

closely resembles the agent-based model depicted in Figure 2.3. In both cases, the agent interacts with its environment, receives feedback in the form of rewards, and continuously refines its policy to optimize future performance.

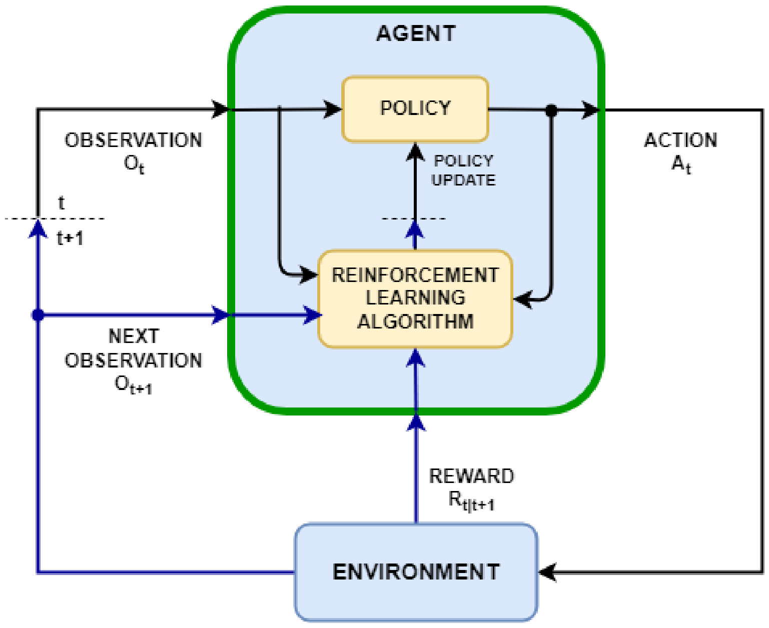


Figure 2.6: RL Agent Scheme
Source: Mathworks[8]

However, RL has its own set of challenges. The most common one is the convergence to a local minimum in the reward function. This means that the agent might become stuck in suboptimal strategy that is not the best one. Moreover, RL is not explainable, meaning that we can't understand why the agent took a specific action in a specific situation.

Another issue with RL is the cost of training. Since the agent learns through trial and error, it needs to perform a large number of actions to explore the environment and learn the best strategy. This can be computationally expensive and time-consuming, especially for complex problems with many variables and states. Moreover, once the agent is trained, its adaptability to new environments or situations is limited, as it is optimized for a specific reward function and environment configuration.

Literature An example of a problem similar to the one presented in this thesis, solved using Reinforcement Learning, can be found in the paper 'DeliverAI: a distributed path-sharing network based solution for the last mile food delivery problem' by Ashman et al. [17].

They aimed at solving the last-mile delivery problem by developing a distributed path-sharing network based on Reinforcement Learning. Their approach uses a multi-agent system to optimize delivery routes and schedules, considering factors such as traffic congestion, delivery time windows, and vehicle capacity.

However, their model simplifies the real-world scenario by assuming fixed delivery locations and known traffic patterns, which may not accurately reflect the dynamic and uncertain nature of real-world logistics environments. Moreover, their approach requires extensive training and tuning to achieve optimal performance.

2.4.3 Planning with LLM

LLMs are trained on vast amounts of textual data and have demonstrated remarkable performance across a wide range of language tasks, from translation and summarization to reasoning and problem-solving. This success has naturally led researchers to explore whether these models can be repurposed for more complex, multi-step decision-making problems that require planning.

[8]`https://it.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html`

The key idea is that the same abilities that allow LLMs to understand and generate language can be harnessed to decompose a planning task into intermediate steps, reason about the consequences of actions, and even generate entire action sequences with minimal or no task-specific training.

2.4.3.1 Chain-of-Thought Reasoning

One of the most influential ideas for using LLMs in planning is Chain-Of-Thought (CoT) prompting. Instead of asking the model to jump directly from a problem statement to a final answer, CoT prompting encourages the model to "think aloud" by generating intermediate reasoning steps. This decomposition can help in planning problems where the solution involves multiple, logically connected steps.

This was first discovered by Wei et al. [28], who demonstrated that prompting the LLM to 'answer step by step' led to improved performance on mathematical problems compared to requesting only the final answer. They also showed that this step-by-step approach could be applied to other fields, ultimately giving rise to Chain-of-Thought reasoning models.

"Reasoning" Models Reasoning-focused LLMs are trained to generate multiple Chain-of-Thought steps, exploring different solution paths before selecting the most optimal one, often using Reinforcement Learning [5] techniques such as RLHF (Reinforcement Learning from Human Feedback) or self-consistency methods during the training.

This approach enhances both accuracy and explainability, as the model articulates its reasoning process while still operating as a generative AI system. Expanding on this concept, reasoning models can integrate external tools, memory, and API calls, forming what is commonly referred to as an LLM Agent, capable of autonomous decision-making and real-world interaction.

Most recent and famous reasoning models released to the public have been developed by many companies, both big and small, such as:

- **OpenAI**: o1[9], o1-mini[10] and o3-mini[11] are reasoning models designed to enhance logical problem-solving capabilities. o1 is specialized in complex problems across various domains, offering robust reasoning skills. Building upon this foundation, o3-mini provides a more cost-effective and faster alternative;

- **DeepSeek**: DeepSeek-R1[12], is a notable AI model from a startup[13]. Released in early 2025, DeepSeek-R1 is recognized for its powerful reasoning and coding skills, achieved at a fraction of the development cost compared to other leading models. Its open-source nature and efficiency have made it a significant player in the AI landscape.

2.4.3.2 Zero-Shot and Few-Shot Planning

In zero-shot planning, LLMs generate action sequences by utilizing their extensive pretraining on text and code, effectively inferring plausible step-by-step solutions to given tasks. Few-shot planning further enhances this by providing LLMs with a small set of demonstrations, enabling them to generalize patterns and improve their action sequencing capabilities.

However, while LLMs can produce reasonable plans, their direct applicability to embodied environments remains challenging. Huang et al. [11] highlight the limitations of naive LLM planning, noting that LLMs struggle with real-world constraints, action feasibility, and long-horizon dependencies. Their work demonstrates that these shortcomings can be mitigated by leveraging the world knowledge embedded within LLMs and applying structured guidance, such as constraints on action generation and feedback-based refinements.

Similarly, Silver et al. [26] extend this inquiry to classical AI planning domains by evaluating few-shot prompting of LLMs on problems expressed in the Planning Domain Definition Language (PDDL). Their findings reveal mixed results: while LLMs can generate syntactically correct PDDL

[9]`https://openai.com/o1/`

[10]`https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/`

[11]`https://openai.com/index/openai-o3-mini/`

[12]`https://github.com/deepseek-ai/DeepSeek-R1`

[13]`https://www.deepseek.com/`

plans in certain domains, they often fail due to a lack of explicit access to transition models and logical constraints inherent to planning problems. Nonetheless, their study also introduces a hybrid approach where LLMs are used to initialize heuristic-based search planners, demonstrating that even imperfect LLM-generated plans can improve the efficiency of traditional AI planning methods.

These findings collectively suggest that while LLMs alone are not yet fully capable of robust autonomous planning, their ability to extract and apply commonsense knowledge makes them valuable tools for augmenting structured planning frameworks. By integrating LLM-generated outputs with classical search-based methods, researchers have shown improvements in planning efficiency and problem-solving robustness, highlighting a promising direction for future research at the intersection of language models and automated planning.

Literature In the paper 'Exploring and Benchmarking Planning Capabilities of Large Language Models' by Bohnet et al. [2], the authors systematically analyze the planning capabilities of LLMs through a novel benchmarking suite that includes both classical planning tasks (expressed in PDDL) and natural language-based planning problems. Their work highlights the limitations of LLMs in planning, particularly their tendency to generate suboptimal or incorrect plans despite their strong language understanding capabilities. To address these shortcomings, they explore various methods to improve LLM-based planning (including many-shot in-context learning, fine-tuning with optimal plans, and the use of chain-of-thought reasoning techniques such as Monte Carlo Tree Search (MCTS) and Tree-of-Thought (ToT)). The results indicate that, while LLMs struggle with planning in zero-shot and few-shot settings, their performance significantly improves when provided with structured demonstrations and reasoning strategies. Moreover, fine-tuning on high-quality plan data leads to near-perfect accuracy in some cases, even with relatively small models. However, challenges remain in out-of-distribution generalization, where models fail to generalize effectively to novel scenarios without additional training. Their analysis also identifies key failure modes in LLM planning, such as constraint violations, failure to reach goal states, and incorrect action sequences, emphasizing the need for better training data curation and reasoning frameworks.

Literature In 'Generalized Planning in PDDL Domains with Pretrained Large Language Models' by Silver et al. [25], the authors investigate whether LLMs, specifically GPT-4, can serve as generalized planners, not just solving a single planning task, but synthesizing programs that generate plans for an entire domain. They introduce a pipeline where GPT-4 is prompted to summarize the domain, propose a general strategy, and then implement it in Python. Additionally, they incorporate automated debugging, where GPT-4 iteratively refines its generated programs based on validation feedback. Their evaluation on seven PDDL domains demonstrates that GPT-4 can often generate efficient domain-specific planning programs that generalize well from only a few training examples. The study also finds that automated debugging significantly improves performance, while the effectiveness of Chain-of-Thought summarization is domain-dependent. Notably, GPT-4 outperforms previous generalized planning approaches in some cases, particularly when leveraging semantic cues from domain descriptions. However, limitations remain, especially in handling domains requiring deeper structural reasoning or non-trivial search processes.

3 Experiment Setting

In this chapter, we provide a comprehensive and in-depth description of the experimental framework designed to evaluate the performance of our LLM-driven agent.

We begin by formally defining the problem, ensuring that our study is framed within a well-structured and precise context. We also outline the specific aspects of the problem that our research aims to investigate, clarifying our objectives and highlighting the choices we made in our work.

Following this, we offer a thorough explanation of the environment used to simulate the delivery platform; this section provides a detailed overview of the web-based system that serves as the operational space for our agent. We describe the structure of the platform, its key features, and how it functions as a testbed for evaluating autonomous agents.

Finally, we discuss the selection of various Large Language Models (LLMs) used in our experiments, including both the models that were actively tested and those that were considered but ultimately not included in our evaluations.

3.1 Problem Definition

As widely explained in Section 2.4.3, the recent advancements in Large Language Models (LLMs) have demonstrated their impressive capabilities across a wide range of tasks. Their ability to process and reason about complex problems opened new avenues for research, particularly in fields such as planning and logistics. Given the power and versatility of these models, we are motivated to further explore their potential in tackling planning and logistic challenges, evaluating their ability to comprehend and solve such problems autonomously.

In this work, our primary focus is on assessing the inherent strengths and weaknesses of LLMs when used in their raw form, without integrating any additional planning frameworks, heuristic search algorithms, or explicit reasoning mechanisms on top of them. Unlike conventional approaches that rely on dedicated pathfinding algorithms, rule-based systems, or carefully structured reinforcement learning paradigms, our objective is to investigate how well an LLM can independently interpret and navigate a logistic scenario using its generative abilities alone.

One of the key aspects we wish to emphasize is that our approach remains purely generative. In other words, rather than embedding domain-specific logic or fine-tuned strategies within the model, we allow the LLM to operate autonomously, generating its own understanding of the environment and devising its own strategies for completing the given tasks.

3.1.1 Our Task

Specifically, our problem formulation is centered around asking the LLM to provide only the *next step* that moves the agent closer to the goal, rather than generating an entire solution at once. This step-by-step approach enables the model to iteratively refine its path based on new observations using the conversation history as "action-result" feedback. Furthermore, we assess the reliability of each generated step by computing the uncertainty of the model's response using the methodology detailed in Section 2.2.2.3.

By taking this approach, we aim to answer these questions about the problem-solving skills of LLMs in logistics problems:

- To what extent can an agent, powered solely by an LLM, solve a logistic problem when placed in an unexpected and unfamiliar environment?

- What are the intrinsic limitations and strengths of this approach compared to traditional rule-based or algorithmic solutions?

To simulate an unexpected and dynamic environment, we designed our experiments around a web-based platform that interacts with the agent through API calls. The platform provides a structured yet unpredictable setting in which the agent must operate. A critical design choice we made in our methodology was to avoid parsing the JSON response containing the map structure. Instead, the agent receives the raw map data (that is added to the prompt) and is expected to interpret it entirely on its own. This decision was made to ensure that the LLM must independently derive the necessary spatial and logistical information without relying on pre-processed or structured inputs.

Additionally, this design choice introduces a layer of robustness: if the API undergoes modifications, such as changes in the response format, the addition of new parameters, or variations in data structure, the agent should still be capable of functioning. This property aligns with our objective of evaluating the adaptability of LLM-driven agents in dynamically changing environments, where real-world conditions may not always remain constant.

Our experimental setup and results will be presented in detail in Chapter 6. However, to summarize our primary evaluation criteria, we focus on testing the following goals of the LLM-based agent:

- **Parcel Pickup:** We evaluate whether the agent is capable of successfully identifying the correct location of a parcel on the map and navigating to that specific tile to pick it up. This task requires the agent to correctly interpret spatial relationships and make movement decisions accordingly;
- **Parcel Delivery:** The second evaluation criterion involves determining whether the agent can correctly identify and reach the intended delivery location based on the information available in the raw map data. Since no explicit delivery coordinates are pre-processed for the agent, it must infer this information on its own.

Through these experiments, we aim to provide valuable insights into the problem-solving capacity of LLMs in a logistic setting, evaluating their adaptability, reasoning limitations, and potential advantages in real-world scenarios.

3.2 Environment - Deliveroo.js

Deliveroo.js it's an Educational Game, developed by Marco Robol for the course on Autonomous Software Agents (ASA) by Prof. Paolo Giorgini, using the Treejs[1] framework.

The code for the server is open and can be accessed on GitHub [2] as well as some example of agents (with different level of complexity)[3].

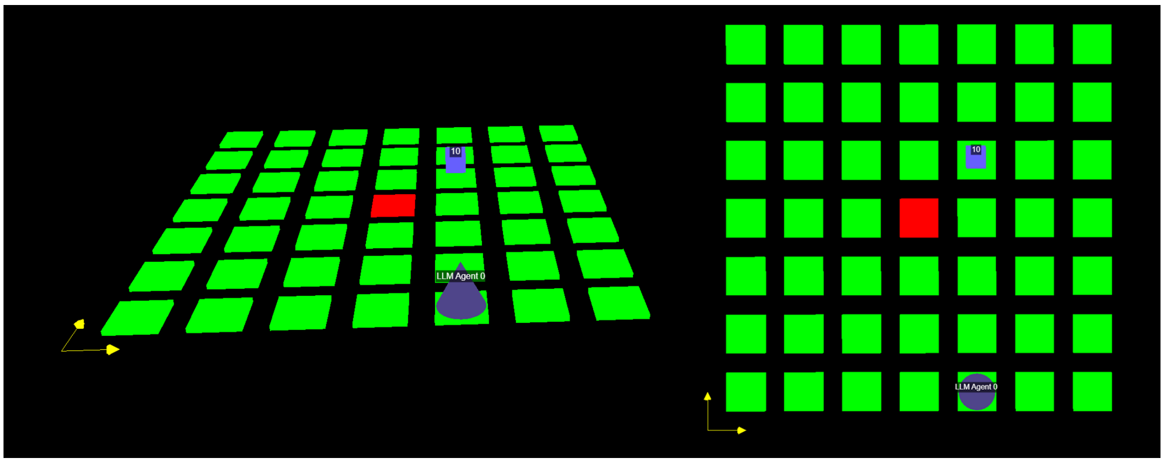


Figure 3.1: Two views of the same 7x7 Map in Deliveroo.js with a single central delivery zone.

The game can be played even by humans, by interacting in the browser; technically speaking, it is a web-based platform consisting of three main components connected to each other via sockets

[1]https://threejs.org/
[2]https://github.com/unitn-ASA/Deliveroo.js
[3]https://github.com/unitn-ASA/DeliverooAgent.js

(implemented with Socket.io[4]):

- **Game server**: it contains the entire logic of the game and it includes the implementation of client connection handler, parcel spawning, current environment status and so on;
- **Agent client**: it is the custom component that we developed to interact with the game server. It is a JavaScript file that connects to the server, manages all the logic of the agent (in our case, the LLM agent) and sends the actions to the server;
- **3D web app**: it is the visual representation of the game. It is a web page that connects to the server and receives the status of the game to render it in a 3D environment. It is not necessary for the agent to work, but it is useful to understand what is happening in the game.

As we can see from the Figure 3.1, the game is a grid of $N \times M$ tiles where the agent can move. Right now, the $(0, 0)$ cell is in the bottom left corner. The map is defined in a JS file (example in Listing 3.1) where the number inside a cell represent the type of the cell.

JavaScript Code

```javascript
// 7x7 goal center deliver
module.exports = [
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 2, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1],
];
```

Listing 3.1: Example of a 7x7 map with a single central delivery zone

There are three possible types of cells:

- **green** (`1`): the agent can move on it. They can contain multiple parcels but only one agent at a time;
- **red** (`2`): the agent can move on it and deliver any number of parcel it has;
- **black** (`0`): the agent can't move on it and they can't contain any parcel (we will not use them in our tests).

The functioning is very straight forward:

- **Agents**: there can be any number of agents that can cooperate or compete. Each agent has a score that is increased by delivering parcels. They are represented as cones with their name on it on the map ('LLM Agent' in Figure 3.1 is ours).
- **Parcels**: they are represented as small cubes with a number on it. The number is the reward the agent will get by delivering it. They spawn in random cells and they can be picked up by the agent. If they are not delivered in a certain amount of time, they may disappear.

[4] `https://socket.io/`

3.2.1 Server Configuration and Event Handling

JavaScript Code

```javascript
module.exports = {
  MAP_FILE: "map_file",

  PARCELS_GENERATION_INTERVAL: "5s",
  PARCELS_MAX: "1",

  MOVEMENT_STEPS: 1,
  MOVEMENT_DURATION: 50,
  AGENTS_OBSERVATION_DISTANCE: 100,
  PARCELS_OBSERVATION_DISTANCE: 100,
  AGENT_TIMEOUT: 100,

  PARCEL_REWARD_AVG: 10,
  PARCEL_REWARD_VARIANCE: "0",
  PARCEL_DECAYING_INTERVAL: "infinite",

  RANDOMLY_MOVING_AGENTS: 0,
  RANDOM_AGENT_SPEED: "2s",

  CLOCK: 50,
};
```

Listing 3.2: Example of a configuration file for the server

The behavior of parcels in the system is defined through the server configuration file. This file specifies key parameters that control parcel generation, reward values, and decay over time. One such configuration is shown in the example in Listing 3.2.

Based on the server settings, a maximum number of parcels can be active simultaneously. Each parcel is spawned at a fixed interval, with a random reward value determined by a specified average and variance. Additionally, the configuration dictates whether the reward remains constant or decreases over time.

In the example, parcels are generated every 5 seconds, but only one can exist at a time. Each parcel starts with a reward value of exactly 10. Furthermore, since `PARCEL_DECADING_INTERVAL` is set to `"infinite"`, the reward does not decrease over time and the parcel will not disappear (until delivered). This setup ensures a stable environment for testing the agent's performance.

The agent can interact with the environment using the following actions:

- **up, down, left, right**: move in the specified direction, if the cell is empty and green or red;
- **pickup**: the agent can pickup a parcel in the cell it is in;
- **deliver**: the agent can drop a parcel in the cell it is in: if it is a delivery zone the parcel will disappear and the reward will be added to the player's score, otherwise it will just remain on the cell.

The server is responsible for transmitting events to the agent, ensuring that it receives all relevant updates in real-time. Specifically, the following events were utilized in our tests:

- `onMap` (width, height, tiles): it sends the width and the height of the map, along with all the tiles in it. Tiles are currently sent as a dictionary {`x: INT, y: INT, delivery: BOOL,` `spawner: BOOL`} where `delivery` is true if the tile is a delivery zone and `spawner` is true if a parcel can spawn on it;

- `onYou` (id, name, x, y, score): it sends the id, the name, the x and y coordinates and the score of the agent connected that the code is piloting;
- `onParcelsSensing` async (perceived_parcels): it is an async function that sends the parcels that the agent can see at any time. The parcels are sent as a dictionary {`x: INT, y: INT, reward: INT`} where `x` and `y` are the coordinates of the parcel and `reward` is the reward the agent will get by delivering it.

3.3 Large Language Models Selection

As mentioned in Section 2.2, Large Language Models are powerful tools that have revolutionized the field of natural language processing. The way they generate text based on input prompts has opened up new possibilities for research and applications in various domains.

One of the core aspects of LLMs is their autoregressive nature, meaning they generate text one token at a time (given the current implementation, but alternative solution are currently being studied, for example by Meta AI[5] in the paper 'Better & Faster Large Language Models via Multi-token Prediction' by Gloeckle et al.[7]), predicting the next most likely token based on the context provided. This capability is what allows LLMs to generate coherent and contextually relevant responses. The way these systems operate can be broken down into a (simplified) step-by-step process:

- The prompt (the request) is tokenized, which means it is divided into smaller units called tokens. These tokens are predefined character combinations that serve as the building blocks for processing text. An example of this tokenization process is illustrated in Figure 3.2. Once tokenized, the prompt is passed to the model for processing;
- The model then generates the "next token" based on a probability distribution computed using attention mechanisms, to determine the most contextually appropriate next token;
- The newly generated token is appended to the existing sequence, and the process is repeated iteratively. This continues until a predefined stopping criterion is met, either reaching a maximum token limit or encountering a special termination token.

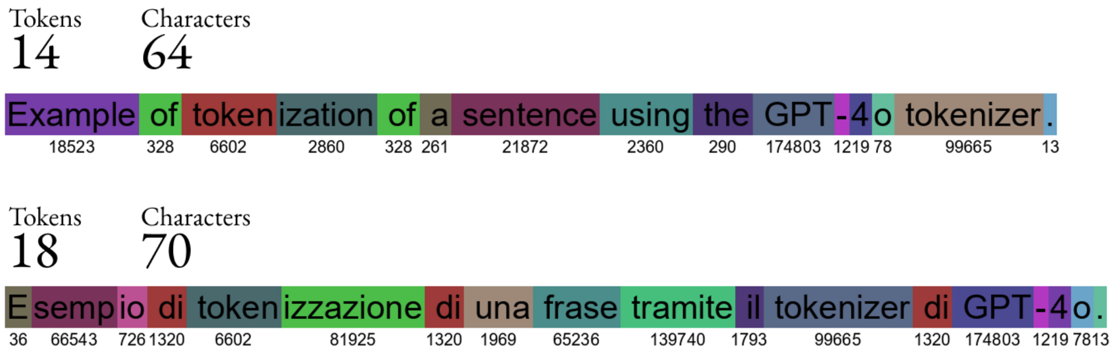

Tokens 14
Characters 64
Example | of | token | ization | of | a | sentence | using | the | GPT | - | 4 | o | tokenizer | .
18523 328 6602 2860 328 261 21872 2360 290 174803 1219 78 99665 13
Tokens 18
Characters 70
E | semp | io | di | token | izzazione | di | una | frase | tramite | il | tokenizer | di | GPT | - | 4 | o | .
36 66543 726 1320 6602 81925 1320 1969 65236 139740 1793 99665 1320 174803 1219 7813

Figure 3.2: Example of tokenization of a sentence using the GPT-4o tokenizer.
Source: Data from OpenAI Platform[6]

One key element of this process is how the next token is selected. The winning token is picked from a probability distribution obtained through the softmax function. However, if selection were purely deterministic, the model would always generate the same output given the same prompt, making it

[5]https://ai.meta.com/
[6]https://platform.openai.com/tokenizer

rigid and predictable. To introduce variability and prevent repetitive patterns, a controlled amount of randomness is introduced using the `temperature` parameter.

The temperature parameter plays a crucial role in regulating randomness in text generation: this mechanism explains why the same prompt can yield different outputs when used multiple times: because the token selection is influenced by this controlled randomness.

Beyond temperature, another factor that influences token selection is the logit bias[7]. Logit bias allows direct intervention in the probability of specific tokens being chosen during text generation. Instead of relying solely on the model's learned probabilities, users can manually adjust the likelihood of certain tokens appearing by modifying the logits (the unnormalized probabilities) before applying the softmax function.

The logit bias mechanism operates as follows:

- Positive bias values increase the probability of a specific token being selected, making it more likely to appear in the generated text.
- Negative bias values decrease the probability of a token, potentially eliminating it from consideration altogether.

This approach gives users more control over text generation, allowing them to guide the model toward preferred outputs while avoiding undesired words or phrases. A simplified implementation of how logit bias works can be seen in the Python code snippet in Listing 3.3, where we want to increase the likelihood of the word "fox" and decrease the likelihood of the word "quick".

Python Code

```python
[...]

# Get the logits (raw predictions)
outputs = model(**inputs)
logits = outputs.logits

logit_bias = {
  # Decrease likelihood for the word "quick"
  tokenizer.encode("quick")[0]: -2.0,
  # Increase likelihood for the word "fox"
  tokenizer.encode("fox")[0]: 2.0,
  # Almost never generate the word "slow"
  tokenizer.encode("slow")[0]: -100.0,
  # Almost always generate the word "brown"
  tokenizer.encode("brown")[0]: 100.0,
}

# Apply logit bias: modify logits of specific tokens
for token_id, bias in logit_bias.items():
  logits[token_id] += bias

# Convert logits by applying softmax
probs = torch.nn.functional.softmax(logits, dim=-1)

[...]
```

Listing 3.3: Example of what a basic implementation of logit bias could look like

[7] https://www.vellum.ai/llm-parameters/logit-bias

This technique is particularly useful in scenarios where the generated text must adhere to specific constraints. For example, logit bias can be used for content moderation, ensuring that the model avoids generating harmful, offensive, or inappropriate content. By assigning a strong negative bias to certain tokens used to build specific words or sentences, users can effectively steer the model away from producing undesirable responses.

Not only, there are some cases where an LLM has been expanded with custom and private information, via fine-tuning or Retrieval Augmented Generation (as we cited in Section 2.2.2). In this context, we may want to force the model to not generate some specific content that may be useful for the overall response but should not be shared.

On the other hand, logit bias can be leveraged to override built-in model safeguards. In some cases, AI models have safety mechanisms that prevent them from answering certain types of questions, responding with phrases like "I'm sorry, but I can't provide that information." By applying a negative logit bias to the tokens that generate the words that build this response, users can force the model to produce an alternative reply, whether it be a reworded refusal or even an answer to the original question.

Overall, logit bias is a powerful tool for modelling model behavior, allowing developers to enforce preferred linguistic patterns, avoid specific terminology, and customize AI responses according to their needs. When combined with temperature adjustments and other generation techniques, it provides a robust framework for controlling LLM output and ensuring its alignment with desired objectives.

3.3.1 Open Source Models

The term "Large" in Large Language Models (LLMs) refers to their extensive number of parameters and the vast datasets used during training. Training these models is a resource-intensive process, both in terms of computational power and time. However, some organizations choose to release their models as open source, allowing the community to access and utilize them freely. This approach is also beneficial the broader AI community, as it fosters innovation: if a new open source model is released and it's "the most powerful", it sets a new baseline for companies. This compels them to find further innovations, either by reducing costs or developing even more powerful models with new features.

In the context of LLMs, the term "open source" differs from its traditional usage in software development[8]. Specifically, there is a nuanced distinction that categorizes publicly available models into two primary types:

- **Open Source**: The creators provide full access to the model's source code, architecture, training data, and pre-trained weights. This level of transparency allows users to understand, modify, and enhance the model comprehensively.
- **Open Weights**: The creators make the model's pre-trained weights publicly available but may withhold other critical components, such as the training data or detailed methodologies used during training. This approach enables users to employ the model for specific tasks but limits their ability to fully comprehend or modify its underlying structure.

In the following subsections, we will present some open models we tested in the early stages of our research. The results analyzed in Chapter 6 do not consider these models due to certain limitations, which we will discuss shortly. Nonetheless, they are worth mentioning as they provided a valuable starting point during the initial phases of our project.

3.3.1.1 Challenges with Open Source Models

One significant challenge we encountered with open source models was the implementation of the logit bias mechanism. While the example in Listing 3.3 may suggest simplicity, the reality is more complex. Implementing this mechanism requires:

- Reconstructing the model's architecture accurately.
- Loading the pre-trained weights appropriately.

[8] `https://promptengineering.org/llm-open-source-vs-open-weights-vs-restricted-weights/`

- Modifying the model's intricate structure to extract the raw values and change them to incorporate the logit bias.

These steps demand substantial time, and in addition, running those models locally requires large computational resources; all of this for something not essential to the core objectives of our project (since closed models provide this feature out of the box).

Moreover, during initial tests, we observed difficulties in constraining the Open Source models to produce specific tokens from a list without altering the logit bias. For instance, when prompted to "Answer with just a single letter between U, D, L, R." (the explanation for this prompt will be given in Section 5.3) the model often responded with full sentences like "Sure, the answer is U!". Truncating such responses to a single token would result in outputs like "Sure" which is not the desired outcome. This is a problem that we didn't have with the closed source models, that we will discuss in the next sub-section 3.3.2.

So, open source models have been tested only in the first instance of the project, to test the logic of the agent without wasting credit with OpenAI API.

They have been run through Ollama[9], a tool for running and managing large language models locally. It simplifies downloading, running, and interacting with models without relying on cloud services.

3.3.1.2 LLaMa 3.2

LLaMa 3.2, developed by Meta AI[10], is a multimodal large language model designed to process both textual and visual data, marking Meta's first open-source AI model with such capabilities. The model is available in two configurations: an 11-billion-parameter version and a more robust 90-billion-parameter variant. There are also a 1-billion-parameter and 3-billion-parameter text-only versions of the models. These models are optimized for deployment on mobile (yet powerful) hardware platforms.

The model tested in our experiments was the text-only 3-billion-parameter version.

Source: Meta icons created by Freepik - Flaticon

3.3.1.3 Gemma 2

Gemma 2, introduced by Google[11], is an open suite of language models available in parameter sizes of 2 billion, 9 billion, and 27 billion. The 27-billion-parameter model has demonstrated exceptional performance, surpassing larger models in real-world conversational benchmarks. This suite is built upon the same research and technology that underpins Google's Gemini models, emphasizing both performance and accessibility.

The model tested in our experiments was the 9-billion-parameter version.

Source: logowik

3.3.1.4 DeepSeek-V3

DeepSeek-V3, developed by DeepSeek [12], is a Mixture-of-Experts (MoE) language model comprising a total of 671 billion parameters, with 37 billion activated per token during inference. It employs Multi-head Latent Attention (MLA) and the DeepSeekMoE architecture to achieve efficient inference and cost-effective training. The model was pre-trained on 14.8 trillion diverse and high-quality tokens, followed by supervised fine-tuning and reinforcement learning stages to fully harness its capabilities. Despite its extensive scale, DeepSeek-V3's training process is notably efficient and it has demonstrated

Source: DeepSeek GitHub

[9]https://ollama.com/
[10]https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/
[11]https://developers.googleblog.com/en/gemma-explained-new-in-gemma-2/
[12]https://github.com/deepseek-ai/DeepSeek-V3

remarkable stability throughout training. Comprehensive evaluations reveal that DeepSeek-V3 outperforms other open-source models and achieves performance comparable to leading closed-source models.

Unfortunately, we didn't get to test it because it released after the end of the project, but it is worth mentioning for future research thanks to its impressive performance.

3.3.2 Closed Source Models - OpenAI

Before delving into the specifics of this section, we want to emphasize that OpenAI is not the only company offering closed-source AI models. Several other providers exist, such as Anthropic[13], which develops the Claude family of models, but also DeepSeek[14], that developed the open source DeepSeek-V3 provides API access to its models as well.

While OpenAI remains the most widely recognized and utilized provider, particularly in research and industry applications, it is important to acknowledge that many of the benefits and drawbacks of closed-source models apply broadly across all such services. Nevertheless, since our work specifically relies on OpenAI's models, our discussion will primarily focus on them while keeping in mind that similar considerations extend to other closed-source alternatives.

A key distinction between open-source and closed-source models is the transparency regarding their architecture. In many closed-source models, details such as the exact number of parameters are not publicly disclosed. For instance, while OpenAI's GPT-3[3] is known to have a maximum of 175 billion parameters[15], the parameter counts for subsequent models like GPT-4[18] have not been officially confirmed. Estimates suggest that GPT-4 may have around 1.76 trillion parameters, but this remains speculative[16].

OpenAI was established as a research organization dedicated to advancing artificial intelligence. They initially released models such as GPT-2[21] and Whisper[19] (a speech recognition model) to the public at no cost. GPT-2, for example, was made available with various model sizes, the largest containing 1.5 billion parameters[17].

Subsequently, OpenAI developed GPT-3 and DALL·E[22], introducing them through a commercial platform and API services. The landscape of AI applications shifted significantly with the release of ChatGPT, a conversational AI model that garnered widespread attention for its advanced language understanding and generation capabilities.

Source: Vecteezy

OpenAI offers access to their models via subscription plans and a pay-as-you-go API, with pricing varying based on the specific model utilized. The API provides granular control over model behavior through parameters such as `logit_bias`, that, as the name suggests, allows users to adjust the likelihood of specific tokens appearing in the generated output by assigning bias values ranging from -100 to 100.

However, a limitation of closed-source models is the lack of transparency regarding updates or changes. Users may be unaware of modifications that could affect model behavior. For instance, there have been instances where the behavior of the logit_bias parameter changed without prior (or "at run-time") notice. We have identified three primary behaviors that the logit_bias parameter can have depending on the version of the API, which appears to be independent from any update of the models themselves:

1. **No change**: The API does not apply the logit_bias at all, resulting in outputs identical to those generated without using the parameter;

2. **Exclusive**: The logit_bias is applied strictly. Setting a token's bias to 100 forces the model to produce that token; if multiple tokens are set to 100, the model will choose among them.

[13]https://www.anthropic.com/
[14]https://api-docs.deepseek.com/
[15]https://en.wikipedia.org/wiki/GPT-3
[16]https://en.wikipedia.org/wiki/GPT-4
[17]https://en.wikipedia.org/wiki/GPT-2

Conversely, setting a token's bias to -100 prevents that token from appearing in the output;

3. **Soft**: The logit_bias is applied moderately. Assigning a token a bias of 100 significantly increases its likelihood of being produced, but other tokens may still be selected. Similarly, setting a token's bias to -100 greatly decreases its likelihood, but it may still appear.

Our tests were conducted during a period when the API exhibited the third behavior (Soft). However, due to frequent API updates, we cannot guarantee the reproducibility of these results. To more mitigate variability, we set the temperature parameter to zero during our tests, aiming for deterministic outputs. To limit the impact of unwanted tokens appearing in the output or vice versa, we decided to ignore any token that was not in the list of expected tokens and we assigned to the missing tokens the 20th log-probability minus the 19th one.

Another crucial feature offered by OpenAI is the `logprobs` parameter. When specified, this parameter returns the log probabilities of the top tokens that the model may generate as the "next token".

This information is essential for computing the uncertainty of the model's predictions, as detailed in Section 2.2.2.3. It's important to note that also the behavior of the `logprobs` parameter can change over time. For instance, when the paper 'Robots That Ask For Help: Uncertainty Alignment for Large Language Model Planners' was released, the `logprobs` parameter returned a maximum of 5 tokens. Since then, this limit has been increased to a maximum of 20 tokens. In the future, it may be adjusted again, so it's important to keep this in mind when working with the API.

Pricing per 1 million tokens for all the OpenAI models used can be seen in Table 3.1; the functioning of caching is explained in Section 5.3.

Model	**Input**	**Cached Input**	**Output**
gpt-4o-mini	\$0.15	\$0.075	\$0.60
gpt-4o	\$2.50	\$1.25	\$10.00
gpt-3.5-turbo	\$0.50	/	\$1.50

Table 3.1: Pricing Table per 1Mil tokens for the tested models.
Source: OpenAI [18]

3.3.2.1 GPT-4o-mini

GPT-4o-mini, developed by OpenAI[19], is a lightweight variant of the GPT-4o architecture, optimized for efficiency while maintaining strong performance across a range of tasks. It is designed to deliver fast inference and lower computational costs, making it suitable for deployment in real-time applications and on-device AI systems. While OpenAI has not publicly disclosed the parameter count, it is positioned as a more efficient alternative to larger models in the GPT-4 family.

The main model used for the tests in our experiments was indeed GPT-4o-mini, selected for its balance of performance and pricing.

3.3.2.2 GPT-4o

GPT-4o is OpenAI's flagship multimodal model, capable of processing and generating text, images, and audio in real time. It represents a significant leap in efficiency and latency, outperforming previous iterations such as GPT-4-turbo while operating at a lower computational cost. Unlike its predecessors, GPT-4o is natively trained across multiple modalities rather than combining separate models for different inputs. Though OpenAI has not released detailed architectural specifications, benchmark results indicate substantial improvements in reasoning, multilingual proficiency, and response speed.

It has been used less than GPT-4o-mini because of the higher cost and the fact that, since from the first tests, the results were almost identical to the mini version.

[18]`https://platform.openai.com/pricing`
[19]`https://openai.com/research/gpt-4o`

AzureAPI GPT-4o was accessed via the Azure OpenAI API, provided by the University of Trento, offering a secure and private interface for interacting with the model. To facilitate communication between the agent's JavaScript code and the API, a lightweight Python server was developed. The server's code is available on GitHub[20].

3.3.2.3 GPT-3.5-turbo

GPT-3.5-turbo is a high-performance variant of OpenAI's GPT-3.5 model. It provides strong general-purpose capabilities while being more accessible for applications requiring large-scale deployment. Though it does not match GPT-4-level reasoning abilities, it remains competitive in many NLP benchmarks and is widely used for production AI services.

Unfortunately, this model has been the weakest in our tests; we will discuss this in depth in the Section 6.5.

[20] `https://github.com/davidemodolo/master_thesis_project/blob/main/azureAPI/server.py`

4 Agent Development

In this chapter we present the iterative development of the agent. We describe the successive phases of its evolution, from the initial prototype to the final implementation. During this process, several challenges and unexpected issues emerged. For each phase, we detail the encountered problems and the approaches adopted to overcome them, explaining the reasoning behind the design choices. The majority of the choices taken in the crafting of the various prompt will be discussed in Chapter 5, while reporting here all the discarded ones.

4.1 Development information

The go-to programming language for AI development is Python. However, we chose to use JavaScript for this project for several reasons. First, both the server and the example agents were already implemented in JavaScript; so a JavaScript interface to communicate with the server was already available. This allowed us to focus only on the agent's logic, without having to worry about the server's implementation. Second, thanks to the availability of the project of the Autonomous Software Agents course, our initial plan was to use an existing JavaScript-based benchmark from the course. Although we ultimately decided against using this benchmark, as explained later, JavaScript remained our language of choice.

Additionally, since there is no more a dedicated JavaScript library for the Azure OpenAI API, instead of manually recreating the necessary API calls we opted for a more efficient approach by setting up a lightweight Python server to act as a middleman. This solution, discussed in Section 3.3.2.2, allowed us to integrate Azure's services seamlessly while maintaining JavaScript for the core development.

4.2 First Approach

In this initial phase of the development and testing process, a trial-and-error methodology was adopted to iteratively refine the system's behavior and try to optimize performance. Unfortunately, this led to moving away from the definition of the problem and, in combination with the poor performance of the agent, it was decided to start over with a new approach. Nonetheless, this first attempt was crucial in understanding the challenges and limitations of the problem, so it is important to describe it.

Text

```
[ROLE]

[MAP]

[LEGEND]

[ACTIONS]

[PARCELS ALREADY PICKED]

[RULES]

[QUESTION]
```

Listing 4.1: Scheme of a prompt used in the first approach, full in Appendix A.1

The approach began by parsing crucial information from the server, which served as the foundation for understanding how the LLM would interact with it. The main point of discuss in the parsing topic was the map, that was represented as a multi-line string like the one in Listing 4.2. The first prompt sent to the LLM was crafted by concatenating the map with all the other information needed to describe the state of the environment, as shown in Listing 4.1.

Text
```
1 1 1 1 1
1 1 P 1 1
1 1 1 1 1
2 A 1 1 1
1 1 1 1 1

LEGEND:
1: Walkable cell
2: Delivery point
A: Agent
P: Parcel
```
Listing 4.2: Parsed Map Result with partial legend

This implementation started as a full raw approach, letting the LLM also decide the goal to pursue. As various challenges and inefficiencies were identified during extensive testing, we progressively implemented a total of seven "helping" parameters.

4.2.1 Helping Parameters

These parameters were introduced with the objective of addressing specific issues observed during experimentation, and each of them played a significant role in shaping the overall functionality of the agent:

- `ANTI_LOOP`: This parameter was introduced to eliminate a common inefficiency in agent movement, wherein the agent would repeatedly traverse the same path in a circular loop, failing to make meaningful progress toward its goal. By setting this parameter to `true`, if the last four action were `["U", "R", "D", "L"]` (either clockwise or counterclockwise) the agent was forced to take an action that prevented the loop for happening. This optimization helped the agent make more intelligent movement decisions, thereby removing the possibility of being stuck in repetitive cycles;

- `HELP_THE_BOT`: The primary purpose of this parameter was to assist the agent in handling parcels more effectively. When activated by setting it to `true`, the agent was programmed to automatically take a parcel if the parcel was located directly below its current position. Additionally, if the agent was positioned at a delivery point, this parameter ensured that the agent would immediately proceed with shipping the parcel without requiring additional decision-making steps. This was implemented to reduce the number of calls to the LLM, since, even in this version of the agent, it was able to always pick up a parcel and deliver it (if in the correct tile);

- `SELECT_ONLY_ACTION`: This parameter was designed to simplify the agent's decision-making process in cases where the list of available actions contained only a single option. When set to `true`, the agent would automatically select and execute the sole available action without hesitation or delay. This was made by a big filtering phase that returned the legal actions:
  - remove the opposite of the last action;
  - remove all the action that was not possible (like going left while in the first column or going up while in the first row);
  - remove the delivery action if the agent wasn't carrying a parcel and in a delivery point;

  - remove the pick action if the agent was in a cell with no parcel.

  This, in combination with the `HELP_THE_BOT` parameter, reduced the number of unnecessary calls to the LLM, thereby enhancing the agent's efficiency, but also giving the agent too little decision power;

- `USE_HISTORY`: This parameter is the only one that was kept for all the future iterations (more on this in Section 6.2). The role of this parameter was to decide whether each call to the LLM should contain only the current state of the environment of the entire message history. If set to `true`, the LLM would have access to the full conversation history, allowing it to make more informed decisions based on past interactions and events. This feature was particularly useful and powerful, but also with a big downside related to the LLM context length, that will be discussed in Chapter 8;

- `REDUCED_MAP`: This parameter was introduced to optimize the space the map occupied in the prompt by limiting the environment described as a slice of the full map and then scaling all the coordinates (of the agent and the parcels) treating the reduced map as the total map. The reduction in size was determined based on the maximum value between `PARCELS_OBSERVATION_DISTANCE` and `AGENTS_OBSERVATION_DISTANCE` (from Server Configuration File, see Section 3.2.1), ensuring that the agent only received the most relevant spatial data necessary for making informed decisions. Essentially, this optimization allowed the attention of the LLM to not be too sparse, but bringing some extra problems, for example by removing any delivery zone from the map since it was too far away while the current goal was to deliver a parcel;

- `HELP_FIND_DELIVERY`: This parameter was specifically designed to assist the agent in locating delivery points more effectively. By setting it to `true`, the system ensured that the closest delivery point (using Manhattan Distance) was always included in the agent's prompt (not as coordinates but as directions, eg. "right and up"), even if that particular delivery point was not within the agent's immediate field of view. In fact, this parameter was implemented to remedy the problem described in the `REDUCED_MAP` point. This feature provided the agent with valuable directional guidance, allowing it to make better routing decisions and reducing the risk of wandering aimlessly in search of a delivery location (or worse, by looping again and again), but also reduced our ability to track the LLM ability in finding the delivery point by itself (more on this in Section 5.2);

- `HELP_SIMULATE_NEXT_ACTIONS`: The goal of this parameter was to enhance the agent's decision-making process by simulating and displaying the expected outcomes of each possible action. When activated by setting it to `true`, the prompt provided to the LLM included a detailed breakdown of how each available action would alter the surrounding environment, by computing for each action the resulting map and attaching all of them to the final prompt. In theory, this additional information could help the agent anticipate and select the most favorable course of action. However, experimental results indicated that enabling this feature led to suboptimal performance, resulting in poor decision-making and inefficiencies, probably due to the size of the prompt that was too big for the LLM to handle.

This design resulted in an implementation that was bringing the project in the wrong direction, because the whole "no framework on top" idea was breaking down even if this didn't have anything to do with planning in a strict sense. Without those helps, the agent was not able to perform well in any environment, and the decision was made to start over with a new approach.

Overall, while this initial approach did not yield the desired performance, it was an essential step for the next iterations of the agent's development.

The code for this first implementation can be found in the `archive/raw_llm_agent.js` file inside the project repository [4].

4.2.2 Takeaways

Through this initial approach, we gained valuable insights into the challenges of designing an effective agent. The key lessons learned from this phase can be summarized as follows.

Why this approach has been discarded? Several key issues made this method not suitable for the project's objectives:

- Unclear prompt style: the way information was structured in the prompt was not optimal. This became particularly evident in the uncertainty computation, where the agent frequently exhibited high uncertainty in its actions;
- Over-reliance on helping parameter: providing excessive hints and structured input to the agent hindered its ability to explore the environment effectively. While guidance could be helpful, too much assistance made the results too distant from what the LLM could achieve by itself.

Key Takeaways from this phase

- Performance: when extensive guidance was provided, the results were still acceptable but inferior to those obtained using PDDL-based solutions. Initially, we considered implementing a PDDL version of the agent to serve as a benchmark. However, as discussed in Section 2.4.1, this comparison would have been unfair due to fundamental differences in approach and assumptions;
- Unintended biases in LLM behavior: Although not directly related to the core functionality of the agent, an interesting observation emerged regarding how the LLM interpreted the presence of other agents. The server allowed the spawning of "enemy" agents capable of blocking paths. While this feature was not used in the main experiments, we discovered that simply including information about these agents in the prompt led the LLM to assume that agents near parcels were actively trying to steal them. In reality, these agents were merely obstacles with no intent to compete for parcels. This behavior highlights inherent biases in the LLM's training data, where similar context might have been associated with adversarial interactions.

4.3 Second Approach

In our exploration of different strategies for designing an LLM-driven agent, this second approach was the weakest one. The fundamental idea was to adopt a "full raw" approach, where the model received all available unprocessed data from the server without any pre-processing or filtering. The hypothesis was that by exposing the LLM to as much raw information as possible, it might be able to infer meaningful patterns and determine the best course of action on its own, giving us the ability to evaluate the LLM's planning capabilities without any external structure.

Text

```
[ROLE]

Raw 'onMap' response: [JSON]

Raw 'onYou' response: [JSON]

Raw 'onParcelsSensing' response: [JSON]

[ACTIONS]

[QUESTION]
```

Listing 4.3: Scheme of a prompt used in the second approach, full in Appendix A.2

To achieve this, the agent's prompt was constructed as a simple collection of "name of the call - JSON result" pairs. Unlike other approaches that structured data into a more human-readable or semantically meaningful format, this method provided the LLM with a direct dump of server responses. The only additional elements in the prompt were the list of available actions and a loosely defined query requesting the next step (necessary to compute the uncertainty).

At first glance, this approach seemed promising in that it completely avoided manual interpretation of server responses, reducing the need for custom logic or intermediate representations. If successful, it could have allowed for a highly adaptable agent that functioned independently of predefined schemas or rigid data structures.

However, in practice, this approach performed poorly. While it occasionally worked in very small maps, it became unreliable and inefficient as soon as the environment grew even slightly more complex. The agent frequently took suboptimal paths, exhibited excessive backtracking, and often failed to reach its goal efficiently.

Additionally, for the agent to work correctly in such small maps, the parameter `USE_HISTORY` from the first implementation was set to `true`, allowing the LLM to leverage the 'action-result' history.

The lack of structured guidance made it difficult for the model to consistently produce useful responses, ultimately leading us to discard this approach in favor of more refined strategies.

4.3.1 Takeaways

Even if this approach was the weakest one and has been discarded very soon, it provided us with valuable insights into the limitations of the LLM in processing raw data. The key lessons learned from this phase can be summarized as follows.

Why this approach has been discarded?

- Arbitrary Naming of API Calls: The names of server calls were not standardized and could change depending on the server's development. For instance, a call named "onMap" might return a list of map tiles, but there was no inherent guarantee of this behavior. Moreover, without explicit context, even we—humans—found it difficult to interpret certain responses;

- Lack of Context and Poorly Structured Input: The raw JSON data lacked structured context, making it challenging for the LLM to infer the correct course of action. Additionally, the query format itself was suboptimal. For example, if the prompt simply asked, "Go there, give me the next step to go there," the model might struggle to determine that the immediate goal was not just reaching (x, y) but selecting the best intermediate step that moved the agent closer to that final destination.

Key Takeaways from this phase

- LLM Inference Capabilities: Surprisingly, the LLM was still able to extract some meaningful information from the unstructured data. While the results were inconsistent, there were instances where the model successfully inferred useful actions despite the messy prompt;

- Significant Impact of Prompt Engineering: Small modifications to the prompt led to drastic changes in the agent's behavior. This highlighted the critical role of prompt engineering in optimizing LLM performance, a topic we will explore in detail in Section 5.3.

4.4 Final Agent

This represents the final iteration of our approach, incorporating substantial improvements in both prompt generation and the overall structure of the agent's implementation.

Initially, prompts were dynamically constructed at runtime using a series of if/else conditions, which made them difficult to manage, debug, and scale. That approach lacked flexibility, as any modification required changes to the core logic of the agent, increasing complexity and reducing maintainability.

4.4.1 Prompt Management System

In the revised approach, we transitioned to a structured prompt management system where predefined templates are stored in a dedicated folder (`prompts` folder in the GitHub repository [4]). These templates use variable placeholders that are replaced dynamically via regex-based substitutions. This change provided multiple advantages: it improved readability, ensured consistency across different prompts, and made modifications significantly easier. Instead of altering the logic of the agent itself, changes could now be made directly at the prompt level, allowing for rapid experimentation and iteration. Additionally, this structured approach facilitated more systematic testing, as different versions of the prompts could be evaluated with minimal effort. Overall, this refinement not only enhanced the reliability of the agent but also contributed to a more efficient development workflow.

4.4.2 Agent Refactoring

Beyond improving prompt handling, a major structural refactoring was undertaken to optimize the agent's implementation. The original codebase was relatively monolithic, with large, complex functions handling multiple aspects of decision-making. This made debugging and extending functionality cumbersome. In the final version, the implementation was restructured into a more modular design, with a greater number of smaller, well-defined functions handling specific tasks. This decomposition significantly reduced code redundancy, improved readability, and made the agent easier to modify. One of the key benefits of this modular approach was its impact on the data acquisition process. Since the agent's core logic was now more flexible, adapting it for different data collection tasks required minimal effort. Simple function modifications or prompt adjustments were often sufficient to tailor the agent's behavior to new requirements. This ability to rapidly reconfigure the agent streamlined experimentation and allowed us to gather diverse datasets efficiently, ultimately improving the quality and scope of our evaluations.

4.4.3 RAG Experiments

We also explored the potential of Retrieval-Augmented Generation (RAG) as a way to enhance the agent's decision-making process. The initial idea was to categorize the parcels *a posteriori* based on predefined classes and provide the LLM with priority information for each category (example in Listing 4.4). This would have allowed the model to make more informed decisions by leveraging structured context about the importance of different types of parcels. However, while this approach showed promise, it was ultimately considered too far from the core objectives of this thesis and was therefore not pursued further.

JavaScript Code

```javascript
if (PARCEL_CATEGORIZATION) {
  for (let parcel of rawOnParcelsSensing) {
    const parcelIdNumber = parseInt(parcel.id.substring(1));
    parcel.food = parcelIdNumber % 2 === 0 ? "banana" : "pineapple";
  }
}
```

Listing 4.4: Discarded implementation of an example of *a posteriori* parcel categorization

Another experimental RAG-based approach involved providing the agent with direct information about past actions, such as: "The last time you were in position `(x, y)`, you attempted to move up, but the path was blocked". While this could be a useful strategy for a "blind" agent (one without direct state awareness) its application in our case would have led to a problematic behavior. The agent would work by just performing random actions, gathering information about the blocked/suboptimal paths and then relying entirely on the RAG-generated feedback to navigate. This effectively bypassed the agent's core decision-making process, turning it into a reactive system rather than a proactive one. Given our focus on autonomous decision-making, this approach was deemed unsuitable.

Nonetheless, these experiments highlighted the potential of RAG for different types of autonomous agents and may be worth exploring in future research.

4.4.4 Stateless and Stateful Agents

The final agent played a crucial role in generating the heatmaps discussed in Section 5.4, primarily leveraging GPT-4o-mini. Task categorization into `pickup` and `delivery` was handled by computing the number of parcels currently in possession of the agent, rather than inferred dynamically from the environment state. While an automated approach could have been explored, manually assigning these tasks ensured clarity and allowed for more controlled testing conditions.

Both stateless and stateful configurations of the agent were developed and tested. The stateless version made decisions based solely on the current state, while the stateful version incorporated historical context to refine its choices over time thanks to the `action-result` feedback in the conversation history.

4.4.5 Uncertainty Handling

To handle uncertainty in decision-making, we experimented with three different approaches:

- **Raw probability selection**: The agent directly selected the action with the highest probability, without any additional modifications. This approach was straightforward, but led to suboptimal (or totally wrong) paths when the highest-probability action was incorrect;

- **Weighted selection**: Instead of always choosing the most probable action, the agent sampled actions based on their probabilities. This method was particularly effective in the stateful configuration: if an incorrect action initially had the highest probability, randomness allowed the agent to eventually correct itself over multiple iterations on the same spot;

- **Stopping mechanism**: This approach follows literally the process explained in the paper 'Robots That Ask For Help: Uncertainty Alignment for Large Language Model Planners' (explained in Section 2.2.2.3): by filtering the actions after the computation of the scaled log-probability of each action, if the resulting set of possible actions was not a singleton the agent would stop waiting for the user input. This method was implemented for completeness in the development, but not tested automatically since it would have required human intervention.

Among these methods, weighted selection proved to be the most effective for data acquisition and testing, as it leveraged randomness to improve path accuracy. This experiment demonstrated that even when an LLM-based agent lacks perfect reasoning abilities, incorporating controlled stochasticity can help guide it toward better long-term performance.

4.4.6 Takeaways

We can summarize the biggest improvements in the final agent as "prompt management" and "agent refactoring". The key lessons learned from this phase can be summarized as follows.
The change in the prompt management provided multiple advantages:

- Improved readability: The separation of logic from text made prompts easier to understand and modify;

- Consistency and maintainability: Storing prompts as templates reduced duplication and made debugging simpler;

- Faster experimentation: Changes could be made at the prompt level without modifying the agent's core logic;

- Better testing and evaluation: Different prompt versions could be systematically compared with minimal effort.

The refactoring of the agent structure also brought several advantages:

- More modular design: Smaller, well-defined functions improved readability and maintainability;

- Reduced code redundancy: Reusable components simplified implementation and debugging;

- Greater flexibility: The agent could be easily adapted for new tasks, making data acquisition faster and more efficient.

4.5 Extra: Closest Cell to the Goal

As part of the iterative process of refining our agent, we also tried to systematically isolate and address specific challenges by progressively reducing the complexity of the final problem. This incremental approach not only helped us pinpoint potential weaknesses in the agent's decision-making process but also provided deeper insights into the LLM's underlying behaviors and limitations.

To achieve this, we conducted a series of controlled experiments, including:

- **Testing on smaller, simplified maps**: By reducing environmental complexity, we could more easily observe the agent's decision-making patterns and identify whether failures were due to reasoning errors or external factors;
- **Using custom, "disposable" prompts**: We introduced minor variations in prompt structures to assess how sensitive the LLM was to different formulations of the problem. This helped us determine whether misinterpretations were caused by the model itself or by the way the information was presented.
- **Decomposing the final goal into smaller, manageable sub-goals**: Breaking down the problem into intermediate objectives allowed us to test whether the agent could handle incremental progress rather than needing to solve the entire task at once. This approach is further detailed in the following paragraphs.

One key issue we wanted to investigate was why the agent often failed to select the optimal action leading toward the goal. Was the model inherently incapable of making the correct choice, or was the issue rooted in the way the prompt was structured? To better understand this, we designed an experiment that introduced a two-step decision-making process (visible in Figure 4.1):

- **Identifying the best neighboring tile**: Before making a move, the agent was first asked to determine the most optimal adjacent tile to step toward, effectively transforming the problem into a local optimization task.
- **Selecting the best action to reach that tile**: Once the target tile was identified, the agent was then tasked with choosing the appropriate action to move in that direction.

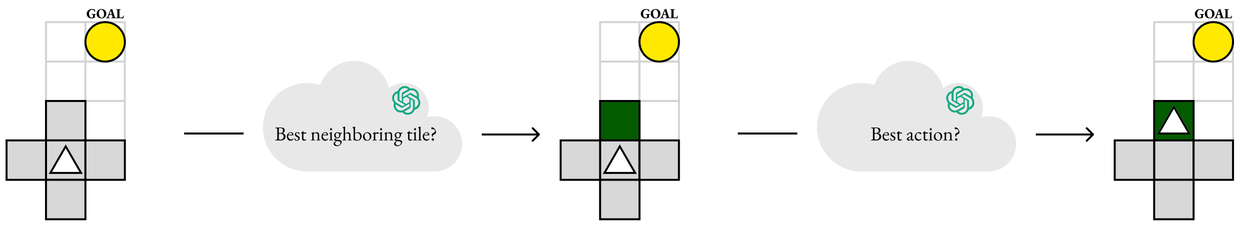


Figure 4.1: Two-steps decision making example

This method effectively reframed the problem from a complex, global pathfinding challenge into a series of simpler, localized decisions. We hypothesized that this decomposition would improve performance by reducing the cognitive load on the LLM and allowing it to focus on smaller, well-defined tasks.

However, despite these modifications, the results did not align with our expectations. The primary issue was that if the agent misidentified the best neighboring tile by choosing a position that actually increased the distance to the goal, then even a perfectly chosen direction would still lead to suboptimal behavior.

Another significant factor was the role of uncertainty computation in the LLM response. When relying on the raw output of the LLM ('Raw probability selection' in Section 4.4.5), results were often inconsistent or outright incorrect. To address this, we experimented also with the uncertainty-weighted random action selection approach ('Weighted selection' in Section 4.4.5), where decisions were influenced by the model's confidence levels. Unfortunately, this introduced additional challenges, as interpreting the results became even more complex.

Ultimately, this experiment underscored the limitations of LLM-based decision-making in logistics scenarios where local optimizations must align with global objectives. The agent's inability to consistently select the correct neighboring tile demonstrated how small errors at the decision-making level could compound, leading to inefficient or even counterproductive actions.

However, as we will explore in Chapter 6, newer LLM models exhibit notable improvements in our task, suggesting that continued advancements in the technology may help overcome these challenges and enhance overall performance in complex logistics tasks.

5 Data Collection

5.1 Visualize the Attention

5.2 Prompts

Explain we needed just a single token.

Cite multichoice benchmarking.

No effect of action because cheating $\rightarrow$Cite emerging behaviors.

5.3 Prompt Creation Choices

Also token usage

5.4 Heatmap Generation

6 Results Discussion

6.1 Stateless

6.2 Stateful

6.3 Stateless and Stateful Combined results

6.4 Closest Cell to the Goal Problems

6.5 Models Comparison

7 Future Works

8 Conclusions

Bibliography

[1] Evan Becker and Stefano Soatto. Cycles of thought: Measuring llm confidence through stable explanations, 2024.

[2] Bernd Bohnet, Azade Nova, Aaron T Parisi, Kevin Swersky, Katayoon Goshvadi, Hanjun Dai, Dale Schuurmans, Noah Fiedel, and Hanie Sedghi. Exploring and benchmarking the planning capabilities of large language models, 2024.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] Davide Modolo. GitHub Project Repository. `https://github.com/davidemodolo/master_thesis_project/`. 12/03/2025.

[5] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[7] Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better and faster large language models via multi-token prediction, 2024.

[8] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997.

[11] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022.

[12] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, March 2023.

[13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[15] John Lafferty, Andrew Mccallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. pages 282–289, 01 2001.

[16] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[17] Ashman Mehra, Snehanshu Saha, Vaskar Raychoudhury, and Archana Mathur. Deliverai: Reinforcement learning based distributed path-sharing network for food deliveries, 2024.

[18] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel

Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[19] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

[20] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[21] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[22] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

[23] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *1st International Conference on Multi Agent Systems (ICMAS 1995)*, pages 312–319, San Francisco, CA, USA, 12-14 June 1995. The MIT Press.

[24] Allen Z. Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, Zhenjia Xu, Dorsa Sadigh, Andy Zeng, and Anirudha Majumdar. Robots that ask for help: Uncertainty alignment for large language model planners, 2023.

[25] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models, 2023.

[26] Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDL planning with pretrained large language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

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

[29] Michael Wooldridge. *An Introduction to Multiagent Systems.* John Wiley & Sons, 1 edition, 2002. Chapter 2.

[30] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms, 2024.

[31] Liping Yang and Ruishi Liang. An ai planning approach to emergency material scheduling using numerical pddl. In *Proceedings of the 2022 International Conference on Artificial Intelligence, Internet and Digital Economy (ICAID 2022)*, pages 47–54. Atlantis Press, 2022.

Appendix A Attachment - Prompts

Prompt

```
You are a delivery agent in a web-based game and I want to test your ability.
    You are in a grid world (represented with a matrix) with some obstacles and
    some parcels to deliver.
Parcels are generated at random on random free spots.
The value of the parcels lowers as the time passes, so you should deliver them
    as soon as possible.
Your view of the world is limited to a certain distance, so you can only see
    the parcels and the delivery points that are close to you.
MAP:
1 1 1 1 1
1 1 P 1 1
1 1 1 1 1
2 A 1 1 1
1 1 1 1 1

LEGEND:
- A: you (the Agent) are in this position;
- 1: you can move in this position;
- 2: you can deliver a parcel in this position (and also move there);
- P: a parcel is in this position;
- X: you are in the same position of a parcel;
- Q: you are in the delivery/shipping point;

ACTIONS you can do:
- U: move up
- D: move down
- L: move left
- R: move right
- T: take a parcel
- S: ship a parcel

You have 1 parcels to deliver.
Important rules:
- If you have 0 parcels, you must look for the closest parcel to pick up.
- If you are going to deliver >0 parcels and on the way you find 1 parcel, you
    should go and pick it up before shipping.
- If you have at least 1 parcel, your goal should be to deliver it/them to the
    closest delivery point. The more parcels you have, the more important it is
    to deliver them as soon as possible.
- If you can't see any delivery point, just move around to explore the map
    until one enters your field of view, then go and deliver the parcels.
- If there is no parcel in the map, just move around to explore the map until
    one parcel spawns, then go and get it.

You want to maximize your score by delivering the most possible number of
    parcels. You can pickup multiple parcels and deliver them in the same
    delivery point.
Don't explain the reasoning and don't add any comment, just provide the action.
Try to not go back and forth, it's a waste of time, so use the conversation
    history to your advantage.
Example: if you want to go down, just answer 'D'.
The closest delivery point is right and up from you.
What is your next action?
```

Listing A.1: Example of a prompt used in the first agent implementation, see Section 4.2

Prompt

```
You are a delivery agent in a web-based game I am going to give you the raw
    information I receive from the server and the possible actions. You have to
    take (pickup) the parcel and ship (deliver) it in a delivery tile.
Don't loop using the same moves.
If the information does not change, it means you are choosing the wrong actions
    .
Raw 'onMap' response: {"width":2,"height":5,"tiles":[{"x":0,"y":0,"delivery":
    false},{"x":0,"y":1,"delivery":true},{"x":1,"y":0,"delivery":false},{"x":1,"
    y":1,"delivery":false},{"x":2,"y":0,"delivery":false},{"x":2,"y":1,"delivery
    ":false},{"x":3,"y":0,"delivery":false},{"x":3,"y":1,"delivery":false},{"x
    ":4,"y":0,"delivery":false},{"x":4,"y":1,"delivery":false}]}

Raw 'onYou' response: {"id":"75d4e78ed8e","name":"raw_llm_agent","x":3,"y":1,"
    score":0}

Raw 'onParcelsSensing' response: [{"id":"p0","x":3,"y":0,"carriedBy":null,"
    reward":10}]

ACTIONS you can do:
U): move up
D): move down
L): move left
R): move right
T): take the parcel that is in your tile
S): ship a parcel (you must be in a delivery=true tile)
Don't explain the reasoning and don't add any comment, just provide the action.
What is your next action?
```

Listing A.2: Example of a prompt used in the second agent implementation, see Section 4.3