FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Planning with Large Language Models

Akanksha Chawla, 03766791

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Planning with Large Language Models

Author:	Akanksha Chawla, 03766791
Supervisor:	Prof. Matthias Grabmair
Advisor:	Prof. Matthias Grabmair
Submission Date:	Submission date

I confirm that this master's thesis in informatik is my own work and I have documented all sources and material used.

Munich, Submission date Akanksha Chawla, 03766791

Acknowledgments

I would like to express my deepest gratitude to everyone who has supported me throughout my journey to this point in my life.

First and foremost my, I am greatly indebted to my Guru - my guide, Matthias Grabmair. His invaluable guidance, encouragement and unwavering commitment to cultivating a meaningful research experience for his students has bolstered my confidence in my own abilities time and again. He has helped me in more ways than one: through insightful feedback, intellectual freedom, and by always making time for my questions, no matter how trivial. His mentorship has left a lasting impact on how I think and learn. Thank you, for fuelling my aspiration to grow as a student for life.
Under his mentorship, I had the opportunity to learn from and work alongside capable colleagues at the TUM Legal Tech Department. A supportive network of curious minds and collaborative spirits.
A Master's thesis is a project that truly brings the phrase "it takes a village" to life. Fortunately, my village has been enriched with nurturing, generous souls.

Long before I could dream, two young engineers were dreaming for me. I start with expressing my gratitude to the two people in my life who sowed the seed of learning in me, way before I even gained consciousness. My dear parents, who repeatedly fought against their own limitations to make my life's possibilities limitless. Without your momentous efforts and constant sacrifices, I would not be writing this document. Thank you, for always believing

Abstract

Contents

1 Introduction

The advent of Large Language Models (LLMs) revolutionizing Natural Language tasks and disrupting almost every field that can be automated is the general preface to most papers in this area of research. Since the introduction of the Transformer architecture in the seminal 2017 paper titled "Attention Is All You Need" (Vaswani, Shazeer, Parmar, et al. 2017), the domain of Artificial Intelligence (AI) has witnessed path-breaking research leveraging this architectural advancement. A year following this was Open AI's first Generative Pre-Trained Transformer (GPT) that marked the birth of LLM as we now know them. Though these models were originally developed to perform word sequence completion tasks, through numerous iterations and experimentations researchers have inched closer to harnessing the true potential hidden within these neural network black boxes. The plethora of AI-led assistants — such as Khanmigo, Notion AI, Cursor, and DeepScribe — is a testament to this progress, showcasing how Transformer-based architectures have revolutionised human-computer interaction.

But the question we try to explore in this research is: how far can we extend the capabilities of these models? Even more specifically, what are the limits of their reasoning, planning, and generalisation abilities? As more research dedicated to developing the next cornerstone in the journey towards channelling the cognitive potential of machines emerges, it becomes increasingly important to examine their planning abilities. At present - no matter how skilled - LLMs still largely act as assistants in human workflows. For LLMs to become truly self-reliant agents, the ability to perform intuitive and context-aware planning is indispensable. This interest has led to an outburst of research prodding and probing their behaviour almost as if they are already human-like organism. To this end, planning, as a form of reasoning, has long been a focus within the AI community. At its core, it's about figuring out a sequence of actions: a policy, that can take: an agent from where it is now to where it needs to be. Traditionally, planning has been approached as an inference problem over world and reward models. In this paper, we explore how well large language models can reason about actions and change, specifically in the context of Blocksworld planning tasks.

On this note, we have seen a recent evolution of LLMs into Language Reasoning Models (LRMs). LRM build upon the capabilities of LLMs by essentially making models pause

to reason before responding. While LLMs are trained to simply predict the next token of generation, LRMs are expected to build reasoning traces leading up to the final answer, thus making thought-through decisions rather than rushing to the conclusion. It's a notion where the model learns throughout its journey, not just at the end as was traditionally done. Within this context of reasoning capabilities lies OpenAI's o1 series of models, which were the first to demonstrate inference-time scaling simply by increasing the length of their Chain of Thought (CoT) reasoning process. It is suspected that OpenAI used these longer CoT traces to Supervised Fine-Tuning (SFT) their existing models, aligning responses more closely with human preferences. This accomplishment has achieved significantly impressive results across reasoning tasks such as mathematics, coding and scientific reasoning. Following this first step on the untouched land, there was a surge of research exploring different ways to get models to reason—ranging from simply adding a 'WAIT' token to encourage more deliberate responses, to using external verifiers as hierarchical reward signals to guide reasoning. Among this group, emerged a novel model training algorithm Group Relative Policy Optimization (GRPO) that made use of the fact that models can understand better than they can generate. The model born out of this training algorithm was: DeepSeek. DeepSeek-R1 matched, if not outperformed, its predecessors on various benchmarks. Unlike larger labs such as OpenAI, the DeepSeek team achieved these results using significantly fewer compute resources, while still leveraging the Reinforcement Learning (RL) paradigm.

Our aim in this paper is to experiment with these methods championed by the industry leaders and emerging researchers on a smaller model by Google - Google/gemma-3-12b-it. We intend to do this on a dataset well-known in the planning with AI community: Blocksworld. We have extended this dataset to fit our purposes. Blocksworld is inherently structured and lends itself easily to reasoning tasks. The planning tasks in this dataset involves moving blocks or stacks of blocks across towers to reach a specific desired goal state. While trivial for humans, the challenge for models lies in the multi-step planning required to get there. As model needs to reason step-by-step through these problems. The crux of our experiment is to see which one of the three: prompting, SFT, or RL, best helps optimize the model's reasoning abilities. Furthermore, whether such reasoning can scale even without massive model sizes or proprietary data long form CoT trails. We test if the model can learn to think in terms of intermediary sub-goals, rather than hallucinate or shortcut its way to the final goal state. Each training phase is designed to not only improve accuracy, but to shape the manner in which the model arrives at its answer. Thus, reflecting the larger shift from language prediction to deliberative reasoning.

2 Literature Survey

This chapter presents a survey of existing work related to reasoning with large language models and its evaluation in well-structured domains. To maintain a clear flow the literature studied has been categorised based on the main techniques used. Each section gives an overview of selected papers, highlights key ideas, and explains how they connect to this research project.

2.1 Overview of Reasoning with Language Models

In recent years, natural language processing (NLP) has seen an immense degree of evolution, and its success has to be credited to the invention of self-attention and Transformer; both of which have neural network-based architectures. This advent has led to exploring different techniques to make models work on downstream tasks of NLP including but not limited to: commonsense reasoning (J. Yang, Jin, Tang, et al. 2023), multiple-choice question answering (Robinson, Rytting, and Wingate 2023) and solving math world problems (Cobbe, Kosaraju, Bavarian, et al. 2021), without having to train these models from scratch.

A distinctive aspect that sets, the last example in this list, apart from the rest is our main area of interest for this project. Solving tasks like grade-school math problems requires one to reason to reach the desired solution. In recent literature, researchers are avidly exploring how to get large (and sometimes small) language models to reason. Research from only couple years ago (Wei, X. Wang, Schuurmans, et al. 2022) introduce Chain-of-Thought prompting, where models are guided to reason step-by-step before giving an answer. This shows strong gains on the above mentioned, math and additionally logic tasks. This acts as a benchmark for testing arithmetic reasoning abilities of models. This is fuelled further with proposal of methods (B. Zhang, Y. Shao, Deng, et al. 2022) to automatically select better reasoning paths from multiple samples. This idea is extended with self-consistency (X. Wang, Wei, Schuurmans, et al. 2023), where multiple reasoning paths are sampled and the final answer is chosen by majority vote.

Following the initial breakthroughs, many more such research work has followed. There have been significant developments in the intersection of LLMs and planning. In this

intersection, LLMs have taken up an array of roles (Kambhampati, Valmeekam, Marquez, and Guan 2023). Roles ranging from generating plans (W. Huang, Abbeel, Pathak, and Mordatch 2022; Valmeekam, Marquez, Sreedharan, and Kambhampati 2023) and heuristics (Valmeekam, Marquez, Sreedharan, and Kambhampati 2023; Ahn, Brohan, Brown, et al. 2022) to eliciting planning knowledge (Guan, Valmeekam, Sreedharan, and Kambhampati 2023). Along with efforts to use human feedback from users and/or the environment (W. Huang, Xia, Xiao, et al. 2022; Raman, Cohen, Rosen, et al. 2022; Yao, Zhao, Yu, et al. 2023) to improve performance. Research has also explored using LLMs as scoring models (Ahn, Brohan, Brown, et al. 2022).

Even so, most research in this domain of planning with LLMs still tackle datasets that focus on rather shallow reasoning. They fail to provide much insight into the true planning abilities of language models. The most prominent and widely used datasets for evaluating recent advancements in reasoning with language models include: BIG-BENCH (Srivastava, Rastogi, Rao, et al. 2022), GSM8K (Cobbe, Kosaraju, Bavarian, et al. 2021), SVAMP (Patel, Marasovic, Agrawal, et al. 2021), CommonsenseQA (J. Yang, Jin, Tang, et al. 2023) and StrategyQA (Geva, Khashabi, Khot, et al. 2021). As the performance on these benchmarks is overall good, the methodologies applied to achieve them are touted in the industry. The most recent such example in this category, are the DeepSeek models (DeepSeek-AI, D. Guo, D. Yang, et al. 2025).

As eluded to the previous chapter, our aim in this research is to experiment with these methods championed by the industry and then evaluate on a dataset well-established in the domain of **classical planning**. In our domain sub-goals cannot be simply ignored and the planning is more tangible. This helps us measure whether these reasoning techniques actually translate to structured, symbolic problem settings that unlike other tasks, cannot be hacked by rote-learning. To this end, the next few sections will elaborate on the methods we adopt and existing work that motivates them, setting the foundation for our research.

2.2 Attempts at Eliciting Reasoning

Inference Only Methods

A growing body of research explores how to elicit reasoning behaviour from language models. The first line of work focuses on inference only methods that do not involve any training. These methods use prompting techniques, most notably CoT, to guide models to produce intermediate reasoning steps.

For example the aforementioned, Wei et al. (Wei, X. Wang, Schuurmans, et al. 2022)

show that adding intermediate reasoning steps in the prompt significantly improves performance on math and logic benchmarks. Another team of researchers (Kojima, Gu, Reid, et al. 2022) extend this further by demonstrating that even simple COT incantations like "Let's think step by step" help models reason better without the need of any supervised data. Other work such as Wang and Zhou (B. Wang and D. Zhou 2022) study conditions under which models can independently generate reasoning steps even without explicit CoT cues.

However, as Ye and Durrett (Ye and Durrett 2022) show, the quality and reliability of these reasoning trace textual explanations remain inconsistent. This research shows that model as large as GPT-3 falter in generating sound reasoning traces. While these methods may seem intriguing the actual impact of these methods on performance for a well-structured classical planning problem like our dataset, still remains to be evaluated. Furthermore, as hinted at by fellow studies (Wei, X. Wang, Schuurmans, et al. 2022; B. Wang and D. Zhou 2022) COT does not positively impact performance for models smaller than 100B parameters. Unsurprisingly, while models smaller in size are capable of producing fluent COT, their coherence with respect to the task remains rather questionable.

In our project, we use an instruction-fine-tuned 12B parameter model due to hardware limitations. This places us well below the 100B+ scale where CoT methods have been shown to be most effective (Wei, X. Wang, Schuurmans, et al. 2022; B. Wang and D. Zhou 2022). In fact, several studies suggest that CoT has limited or even negative impact on smaller models. This makes it important to critically evaluate whether inference-only methods actually help in our setup.

Supervised Approaches for Reasoning

Next in line are supervised approaches, where models are fine-tuned on datasets that contain explicit example reasoning traces. Unlike inference-only methods, these approaches try to teach the base model how to reason by directly exposing it to numerous examples of step-by-step thinking during training. Starting with study that offers practical insights (Pareja, Smith, Kumar, and Lee 2024) on how small instruction-tuned models can benefit from reasoning-aligned supervision. Followed by research that train verifiers on GSM8K (Cobbe, Kosaraju, Bavarian, et al. 2021), to improve both answer accuracy and reasoning quality. Other works (M. Li, R. Zhang, X. Chen, and Gupta 2023) contradict this and state that models often rely more on structural patterns in demonstrations than on the content itself. This raises questions about what is actually being learned by the model. A question we will try to answer in this research.

Another crucial factor within this approach is its heavy reliance on long, high-quality reasoning traces. This data needs to be of a sufficient accuracy and well-formed to help the model learn effective reasoning. (Yeo, Tong, Niu, et al. 2025; L. Yuan, W. Li, H. Chen, et al. 2024; DeepSeek-AI, D. Guo, D. Yang, et al. 2025) Studies show that models can suffer from low-quality reasoning traces, as this may introduce noise and reduce overall performance, especially if the model size itself is relatively small (<3B) (R. Luo, J. Li, C. Huang, and W. Lu 2025). Fortunately, our model size is well over the suggested limit so it is less likely to be affected with this result. Prior work suggests that fine-tuning should outperform prompting alone (Stiennon, Ouyang, J. Wu, et al. 2020; Perez, Ribeiro, Kiela, et al. 2021; Ouyang, J. Wu, Jiang, et al. 2022), to what extent does this holds for our use-case is however, yet to be determined.

Reasoning with Rewards

A rather recent approach to shape model reasoning is using Reinforcement Learning (RL). In general, such methods provide reward signals based on correctness, explanation quality, or process alignment. The DeepSeek-R1-Zero work (DeepSeek-AI, D. Guo, D. Yang, et al. 2025) is a prominent example where pure RL without supervised fine tuning (SFT) is used to incentivise reflective, multi-step reasoning. However, this is far from the first attempt to harness the power of RL to teach models how to reason effectively. Earlier work introduced RLHF (Reinforcement Learning from Human Feedback) (Ouyang, J. Wu, Jiang, et al. 2022), which used preference-based rewards to improve alignment and response quality. A technique the o1-openAI team is also suspected to be using in the making of their recent successful strawberry model, which has demonstrated strong performance on complex reasoning tasks (Valmeekam, Marquez, Sreedharan, and Kambhampati 2023; Qin, X. Li, H. Zou, et al. 2024). Additionally, more recent studies have shifted toward more reasoning specific rewards. For example, a study proposes RLEF: Reinforcement Learning through Execution Feedback (Gehring, K. Zheng, Copet, et al. 2025). This method uses execution feedback at inference time, to guide LLMs toward correct solutions.

A step further into examining how LLMs think, some methods are designed to map out the model's entire "thought process" and use reward signals from external verifier to guide the desired reasoning path. This paradigm generally, leverages the Monte Carlo Tree Search (MCTS) to its benefit (Qin, X. Li, H. Zou, et al. 2024). A paper built on this notion that uses process-level reward functions to enable models to learn structured reasoning steps without needing step-by-step supervision. (L. Yuan, W. Li, H. Chen, et al. 2024). Another study that applies this also introduces an enhanced iterative preference learning framework. They break down instance level rewards into fine grained step-wise signals via MCTS rollouts and apply Direct Preference Optimization (DPO) to update

the model (Y. Xie, Goyal, W. Zheng, et al. 2024).

Another upcoming way to make use of external verifiers is through Reinforcement Learning with Verified Rewards (RLVR) (Su, Yu, Song, et al. 2025; Yue, Z. Chen, R. Lu, et al. 2025; Y. Wang, Q. Yang, Zeng, et al. 2025). In this setup, there is no separate trained reward model. The signal is directly gathered from an existing deterministic heuristic. This is an approach we also adopt and will discuss in more detail later. It works well in our case because we are working with classical planning problems, where established libraries can be used to evaluate plans reliably. A notable concern with RLVR, as pointed out by existing work, is that while it improves sampling efficiency, it does not significantly expand underlying reasoning abilities of the base model (Yue, Z. Chen, R. Lu, et al. 2025). Whether we reach the coveted 'AHA' moment as claimed by the DeepSeek team with this approach, still remains an open question.

2.3 Evaluating Reasoning in Language Reasoning Models (LRM)

In previous sections, we briefly touched upon the question of LLMs genuinely reason, or merely pattern-match against their training data. This section delves deeper into this question and explores results seen using different methodologies already proposed in the existing literature to probe reasoning abilities in both LLMs and their more new and improved version: Language Reasoning Model (LRM).

2.3.1 LLMs: Can they plan?

Central topics of discussion in this paper is to test whether LLMs are capable of planning. As seen earlier, while LLMs have demonstrated strong performance on a range of tasks, there is still scepticism on their planning abilities. Some even deem the models to simply be universal approximate retrieval n-gram models (ref:can llms plan?). This argument claims that LLMs are not solving tasks with guarantee and rather retrieving approximate guesses for solution. To tackle this scepticism, studies have attempted to put critique into the training loop. The role of generating this critique, was also assigned to an LLM. In such scenarios where LLMs are encouraged to reason through self-critiquing (Valmeekam, Marquez, and Kambhampati 2023; Weng, H. Guo, H. Liu, et al. 2022) while one group claims that the performance improves, another experienced a stark opposite. This divergence highlights that the effectiveness of self-critiquing, and more broadly, the planning capabilities of LLMs, can vary significantly depending on the task formulation, model architecture, training objective, and evaluation method. Therefore, there have been continued efforts made to improve the reasoning abilities of LLMs and consequently

evaluate these improvements (C. Zheng, Z. Liu, E. Xie, et al. 2024; T. Liu, W. Xu, W. Huang, et al. 2025; L. Zhang, B. Wang, Qiu, et al. 2025). The relevant methods have been illustrated in detail in the prior section. Our aim through this study is to evaluate the performance of these paradigms for our problem statement.

2.3.2 Rethinking Current Reasoning Benchmarks

The main challenge in evaluating reasoning lies in the ambiguity of the term itself. In traditional contexts, reasoning involves: inference, abstraction and generalisation. However, when applied to LLMs, reasoning is often assessed through final task accuracy without necessarily even verifying whether the model followed a coherent reasoning process. As a result, models that rely on memorised patterns or dataset artefacts may appear to "reason" effectively, even when no actual inferential process is involved. As demonstrated by a literature tackling such evaluation tasks (Du, He, N. Zou, et al. 2023; C. Xie, Y. Huang, C. Zhang, et al. 2025; Asai and Fukunaga 2018).

This evaluation strategy is often accompanied with benchmark categories that further promote this naive interpretation of the task. Examples of which were mentioned in the previous section 2.1. As a consequence there was an emerging need for more reasoning oriented benchmarks. Datasets designed that have been designed to test different aspects of reasoning. Multi-hop question answering tasks like HotpotQA (Z. Yang, Qi, S. Zhang, et al. 2018) and MuSiQue (Trivedi, Rajani, Xiong, et al. 2022) require models to combine information from multiple sources. While tasks like CLUTRR (Sinha, Sodhani, Rajeswar, et al. 2019) or SCAN are designed to test systematic generalisation and symbolic reasoning. More recently, environments such as BabyAI (Chevalier-Boisvert, Bahdanau, Lahlou, et al. 2019) and ALFWorld (Shridhar, X. Yuan, Côté, et al. 2020) have been introduced to assess reasoning in interactive, sequential contexts such as planning and control. Our domain of choice falls very close to the last example in this list. As mentioned earlier in the introduction section, we use Blocksworld: a classical planning domain dataset. To evaluate model performance in this domain we enact step by step plan generated by the model on a simulated environment. Thereby, somewhat circumventing the reliance on solely the final output. We build on the Blocksworld dataset introduced by Asai (Asai and Fukunaga 2018) , which was intended to bridge symbolic planning and real-world imagery. We then extend it for plan generation tasks by making it more exhaustive. We do this by: increasing the variety of initial and goal states and adding longer and more complex action sequences. This makes the dataset more challenging and better suited to evaluate multi-step reasoning in LLMs.

2.3.3 Blocksworld as a Reasoning Benchmark

A fellow, recent study uses Blocksworld domain to evaluate reasoning abilities of LRMs (Valmeekam, Marquez, Olmo, et al. 2023). In this study, researchers evaluate an array of models ranging from different versions of Claude, OpenAI GPT-4, LLaMA and finally Gemini. They benchmark these models across different experimental suites and importantly, consider factors such as efficiency, cost, and reliability. Practical metrics that are often overlooked in most academic research. This group's findings show that albeit after a consuming a larger number of inference tokens (Open AI's o1), LRM is able to exceedingly surpass it's former LLMs in performance on the simplest task in their portfolio. While they have argued in the past that LLMs generate outputs via approximate retrievals (Kambhampati 2024), the authors claim that o1 (an LRM) can supplement a base LLM with the missing skills required for improved reasoning, without relying on an external verifier. However, if one were to look at other practical metrics mentioned earlier: efficiency, cost, and reliability, LRM's standing becomes rather murky. This model in its form factor at the time, levied an ambiguous costing system wrt inference token surcharges (the main suspected source of it's better performance). Additionally, this model was not always reliable in its performance with a bit of obfuscation within the same task and tasks other than that of plan generation within the same domain of Blocksworld. Leading to the authors' recommendation of using an LLM-modulo approach: an external domain-specific solver along with an LLM for translation between task language and natural language. This approach is touted as 100% accurate, reliable and cost-effective.

Aforementioned paper further holds a notion that has since become a popular opinion with regards to o1, especially after the recent success of DeepSeek models. The team believes that o1 models combine an underlying LLM into an RL-trained system. RL steers the creation, curation and final selection private Chain-of-Though reasoning traces. Thus deeming it fundamentally different in nature from LLMs. Whether, this paradigm is re-producible for our domain within limited constraints is what we explore in our research. According to a study RL does not elicit any novel generations leading to an 'AHA' moment as claimed by DeepSeek paper (**rlincentivize**). This claims that RL primarily serves to refine and reinforce the behaviours already present in the base model. Thereby streamlining the path chosen by the existing base model This suggests that the perceived gains from RL may stem more from sampling efficiency and response stability than from any genuine growth in reasoning ability. In light of these insights, our work aims to assess whether RLVR, when applied within a well-structured and symbolic domain like Blocksworld, can offer genuine improvements in reasoning or merely polish what the base model already knows.

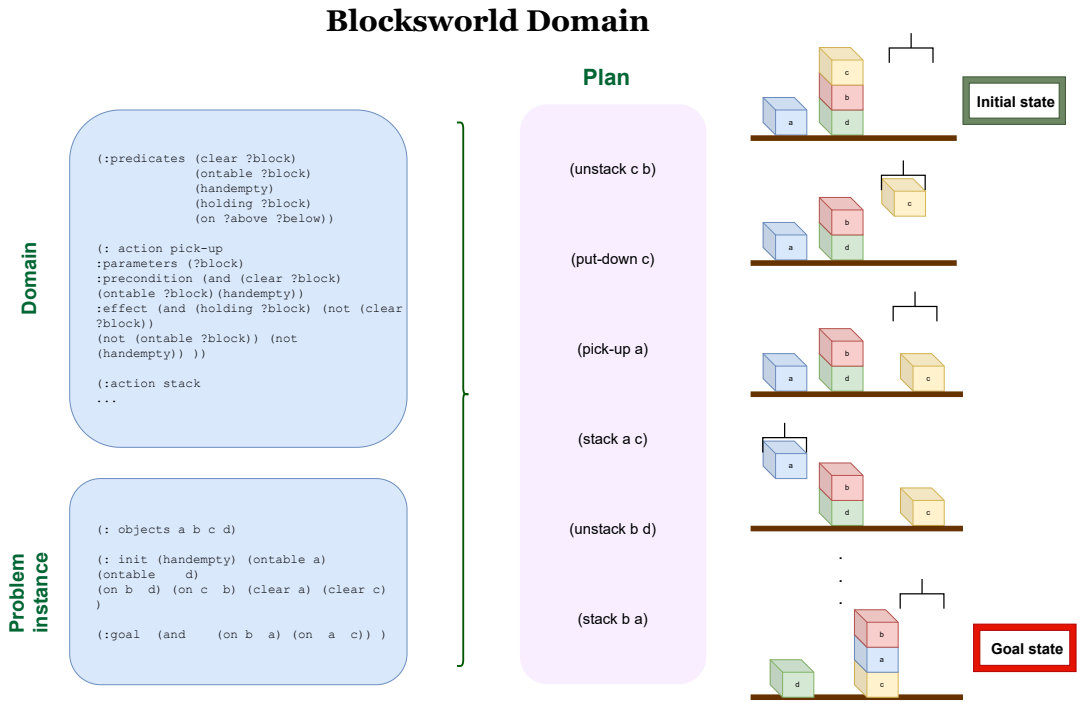

Figure 1: PDDL representation of the Blocksworld domain. We start with describing the domain language itself in the top-left section of the image. Followed-up with an example problem instance in the box below the domain language description. After this is given a sample plan to reach the desired goal state. Enlisted actions in this plan are acted upon in the environment on the right. The environment state representation has been annotated with initial and goal state.

2.4 Limitations and Research Gaps

Throughout this chapter we have tried to hint at potential blind spots in the existing methods of approaches to reasoning with LLMs. In this final section of the literature survey, we bring these observations together to present a consolidated view of the current gaps and thereby emphasising our motivation behind the need for this project.

In previous sections we dissected the approaches we intend to take during the course of our research. Out of the methodologies proposed, SFT requires large amounts of reasoning specific data to make any noticeable impact on performance. This dependency on data availability highlights the importance of gathering robust data. As eluded to earlier, there have been studies that showed LLMs can tend to pattern match and heavily rely more on structural patterns than on the content of the input data itself. An analysis first brought to print in 2022 (M. Li, R. Zhang, X. Chen, and Gupta 2023) and then revisited as late as the beginning of this year, 2025 (D. Li, Cao, Griggs, et al. 2025). The overall conclusion retained. Even after we can ensure to circumvent this issue through mystifying or obfuscating the data structure, it remains crucial to gather enough of such data. The fact that AI training tasks require large amounts of data is certainly not unheard of. The novel problem here is that, for purposes of evoking human-like reasoning from models, one also needs to find a way to produce high-quality text to train the base model on (Ding, Y. Chen, B. Xu, et al. 2023; Y. Wu, Sun, H. Yuan, et al. 2024; L. Luo, Y. Liu, R. Liu, et al. 2024; Cui, L. Yuan, Ding, et al. 2024). This is far from trivial. The current strategies of SFT requires large amounts of high-quality, structured data to train models to generate coherent and interpretable reasoning steps. Crafting such data, especially in symbolic or multi-step domains is costly, time consuming, and domain specific. As pointed out in an earlier section, study done to asses the quality of reasoning traces produced by LLMs continue to show reasoning traces that are often superficial or inconsistent. Whether this idea has changed after the emergence of LRMs like o1 and DeepSeek, is to be evaluated.

A crucial limitation from an implementation perspective, which is often overlooked in theoretical discussions, is the need for significant compute and memory resources, especially for reinforcement learning (RL). RL algorithms are particularly demanding because they require large amounts of memory and powerful hardware to run efficiently.

As academic researchers, we have limited access to such resources. This is one of the main reasons why we use Parameter-Efficient Fine-Tuning (PEFT) methods in our research (Hu, Shen, Wallis, et al. 2023). PEFT methods are designed to reduce the computational and memory requirements by fine-tuning only a small subset of model parameters, which makes them more suitable for environments with limited resources. However, even though PEFT methods claim to not significantly affect model performance, in practice, our research speed and sometimes even performance are affected by our hardware limitations. This challenge is even more noticeable in RL, where memory demands can be especially high. To address this, we also use quantization techniques to further reduce the memory footprint of our models . Despite these strategies, resource constraints remain a practical challenge throughout our work.

3 Methodology

3.1 Introduction

In this chapter we begin by presenting a brief overview of the dataset generation, outlining our motivation behind it, the process we follow and finally the resulting distribution. This is followed by the model selection process and criterion behind the eventual selection. Since evaluating an LLM response requires checking action correctness and plan validity, next we describe how responses are processed. For this, we rely heavily on the `unified-planning` environment which we will briefly touch upon.

This chapter further elaborates on the methodology we follow in this work across the three main experimental setups: prompting, Supervised Fine-Tuning, and Reinforcement Learning. The objective is to understand how each of these approaches contributes to reasoning and plan generation in the Blocksworld domain. To this regard, we describe the SFT setup, including pre-processing steps, model input formatting and training details. Lastly, we look at the RL setup, more specifically Group Relative Policy Optimization, explaining how the environment, reward structure, and learning objectives are defined. Additionally, we also outline the structure and role of different variations of one-shot prompts in producing action plans from language models.

3.2 Dataset Generation

Each of the following sections builds upon the dataset introduced in its previous section and, together they provide a basis to evaluate the model's planning ability.

3.2.1 Introduction

Blocksworld is a legacy environment used to study planning and reasoning. It is a well-defined domain and yet it contains enough complexity to challenge reasoning models. For example, through sub-goals that can conflict with one another and tasks that naturally break down into smaller sub-problems. Given that the objective of our research is to optimise the reasoning capabilities of the selected model, we choose to generate

a custom dataset within the Blocksworld domain. Usually, problem statements from Blocksworlds domain consist of a set of named blocks that can be stacked on top of one another or placed on a table. The goal is to move the blocks from an initial configuration to a target configuration using a sequence of valid actions, namely: **pick-up, stack, put-down, unstack**. Any action can only be performed after required pre-conditions have been met, upon performing the action the relevant state variables are updated accordingly to reflect the new configuration. Though the rules are simple, solving a Blocksworld task often requires several steps of planning, making it a useful test-bed for evaluating how well models can reason through intermediate goals to reach a final state.

We are, by no means, the first ones to use Blocksworld domain problems for testing reasoning abilities of models. The domain has been widely used in classical planning research (Fikes and Nilsson 1971), and more recently in learning-based approaches where models attempt to generate action sequences or reason through symbolic transitions (Asai and Fukunaga 2018; Y. Zhou, Paduraru, Boteanu, et al. 2020). It has also featured in neurosymbolic settings that aim to integrate neural networks with planning algorithms (Valmeekam, Zhan, Brafman, et al. 2022). Along with program synthesis tasks requiring interpretable intermediate steps (Lample and Charton 2019). This consistent use across decades and methodologies makes Blocksworld a reliable benchmark for evaluating reasoning models. A more detailed assessment on this topic can be found in the chapter pertaining to related work in chapter 2.

3.2.2 Blocksworld Dataset Generation

As the blocksworld domain is well-known there are pre-exisitng datasets that have been used for various studies, as mentioned above. This is freely available to use for academic purposes. Although there's different versions of this dataset available, they are not sufficiently vast enough for our purposes. We intend to fine-tune LLMs and further proceed with Reinforcement Learning. Both of these paradigms require an extensive amount of data points to get ahead of issues such as overfitting and to ensure generalization across diverse problem configurations. The limited size of available datasets thus would have restricted the robustness and reliability of our experiments. This emphasizes the need for either synthetic data generation or augmentation methods to support our experimental objectives.

Owing to this insight, we proceed to create our own blocksworld dataset with namely:

- 3 bock configurations

- 4 block configurations
- 5 block configurations

Thus, our generated dataset contains initial and goal configuration pairs of n-number of blocks, where n ranges from $3 \ldots 5$. Within an nth-block subset our aim is to get an exhaustive set of Blocksworld configurations possible for that n.

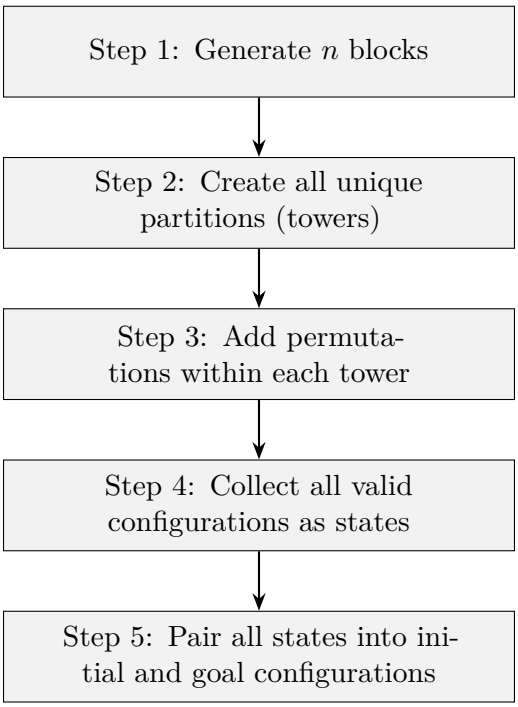

Figure 2: Flowchart for generating Blocksworld initial–goal state pairs.

Algorithm for init, goal State Pairs

Let's breakdown the steps used in the dataset generation algorithm.

Step 1: Generate Blocks

We begin by generating a list of n unique blocks. These blocks are initially represented as integers from $1 \ldots n$.

Step 2: Create All Unique Partitions (towers)

Next, we calculate all possible ways to divide these blocks into one or more tower of blocks or stacks. Each tower is represented as a group of blocks. In this step we recursively

generate all valid partitions, such that no block is repeated. The order of towers is considered irrelevant at this point.

Step 3: Add All Permutations Within Towers
For towers that contain more than one block, we further generate all permutations of the block order within that tower. For example: a tower containing blocks: (1, 2) also gives us (2, 1) as a valid variation. [1] These permutations represent different ways blocks can be stacked on top of one another.

Step 4: Combine All Valid Configurations
We collect all unique configurations, including original partitions and the new permutations, into a set of complete block arrangements. Each configuration reflects a **possible state** in the Blocksworld domain.

Step 5: Form All Possible States into Initial–Goal State Pairs
Finally, we pair each state configuration with every other state configuration to form exhaustive initial and goal state combinations. We naturally avoid pairing a configuration with itself. These pairs represent the full set of Blocksworld planning problems that our system has used throughout this research.

Block Annotation via Colours Additionally, to name and identify these blocks we ascribe colours to them. The idea is to keep the model prompt textual. Each subset of n colours was randomly selected from a larger set of 20 available colours. After generating a fixed subset of n colours, we mapped each block number (from 1 to n) to one of the selected colours in order. For example, if the original configuration was (1,2,3) and the chosen colours for 3-blocks dataset were: violet, teal and brown (which they are) then the mapping will be: {1: 'violet', 2:'teal', 3: 'brown'}. Consequently, the coloured configuration of blocks will be: ('violet','teal', 'brown').

Problem Representation

As previously mentioned, in language model training it is generally advisable to maintain Natural Language (NL) flow in input prompts. On these lines, we convert all the tuple block representation to natural language description followed by other researches when working with LLMs in the Blocksworld domain.

Let's continue with our previous example configuration of ('violet','teal', 'brown'). This tuple of 3-blocks in natural language representation would look like so: *the brown block is clear, the hand is empty, the violet block is on the table, the teal block is on top of the violet block, the brown block is on top of the teal block.*

[1] We represent blocks stacked from bottom to top as arranged from left to right.

Both the initial and goal states are represented in this natural language format. Each instance in the dataset is stored as a pair of strings: one for the initial state and one for the goal state. These are then saved in a structured CSV format. This allows the dataset to be easily parsed and used across different training pipelines.

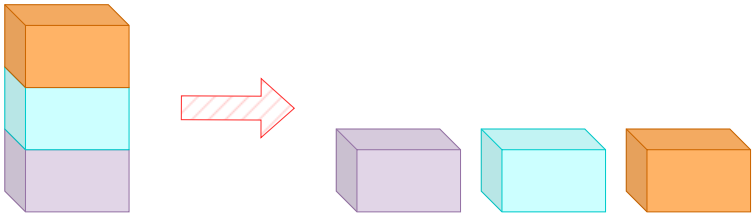

Figure 3: Visualisation of example block configuration with initial state (left) to goal state (right)

Example block configuration as represented by natural language

"As initial conditions I have that, the brown block is clear, the hand is empty, the violet block is on the table, the teal block is on top of the violet block, the brown block is on top of the teal block.
My goal is to have that the violet block is on the table, the teal block is on the table, the brown block is on the table."

Figure 4: Natural Language representation of example state configuration

3.2.3 From Dataset to Prompt

Here we describe how structured Blocksworld state pairs are converted into natural language prompts for LLMs, detailing the instructions, metadata, demonstration formatting, and final prompt template used to support model reasoning.

Prompt Design Template

Now that we have seen the natural language representation of our problem statement, we can dive deeper into this and see the structure of the prompt's design template. We decide on the template by referring to prompt formats used by fellow researchers working on planning and reasoning tasks with LLMs. More specifically, we relied on the prompt template used in the Planbench paper adapting it slightly to suit our task setup.
Each prompt follows a consistent template that includes four main parts:

- a brief introduction to the task
- a list of possible actions the model can use
- the rules of the Blocksworld environment
- the specific problem statement with initial and goal state configurations in NL text

One-shot Prompting

We realise early on during our experimentation that one-shot prompting evidentially performs better for our problem than zero-shot. Especially in enforcing action compliance. That is, providing a demonstration helps the model adhere more strictly to the list of allowed actions. So this is the set-up we choose for this research. The prompt template remains the same, with the only difference being an addition of a one-shot example placed right before the actual problem statement.
Since it's one-shot prompting, we also need to include the gold plan[2] corresponding to the demonstration problem. This plan serves as a reference example to guide the model in generating a valid sequence of actions for the actual test input. Following the task information and the demonstration, the final part of the prompt is the problem statement. As shown in the figure below, the problem statement is appended with the starting `[PLAN]` tag to indicate where the model should begin generating the output plan.

One-shot Prompt for Plan Generation

```
I am playing with a set of blocks where I need to arrange the blocks into
stacks.  Here are the actions I can do
↪ Pick up a block
↪ Unstack a block from on top of another block
↪ Put down a block
↪ Stack a block on top of another block
I have the following restrictions on my actions:
I can only pick up or unstack one block at a time.
I can only pick up or unstack a block if my hand is empty.
I can only pick up a block if the block is on the table and the block is
↪ clear.
A block is clear if the block has no other blocks on top of it and
↪ if the block is not picked up.
I can only unstack a block from on top of another block if the block I am
↪ unstacking was really on top of the other block.
I can only unstack a block from on top of another block if the block I am
```

[2]Generation of the gold plan is a separate topic discussed in more detail in the upcoming sections.

```
↪ unstacking is clear.
Once I pick up or unstack a block, I am holding the block.
I can only put down a block that I am holding.
I can only stack a block on top of another block if I am holding the block
↪ being stacked.
I can only stack a block on top of another block if the block onto which I
↪ am stacking the block is clear.
Once I put down or stack a block, my hand becomes empty.
Once you stack a block on top of a second block, the second block is no ↪
longer clear.

[STATEMENT]
As initial conditions I have that, the green block is clear, the hand is
↪ empty, the red block is on the table, the pink block is on top of the
↪ red block, the green block is on top of the pink block.
My goal is to have that the green block is on the table, the red block is
↪ on top of the green block, the pink block is on top of the red block.

My plan is as follows:

[PLAN]
unstack the green block from on top of the pink block
put down the green block
unstack the pink block from on top of the red block
put down the pink block
pick up the red block
stack the red block on top of the green block
pick up the pink block
stack the pink block on top of the red block
[PLAN END]

[STATEMENT]
As initial conditions I have that, the brown block is clear, the hand is
↪ empty, the violet block is on the table, the teal block is on top of
↪the violet block, the brown block is on top of the teal block.
My goal is to have that the violet block is on the table, the teal block
↪ is on the table, the brown block is on the table.

My plan is as follows:

[PLAN]
```

Figure 5: One-shot prompt template used for plan generation

3.2.4 Dataset splits

This section explains how the dataset is split into training, validation, test, and contaminated sets. We decide on ratios that ensure diverse training and effective evaluation.

Split Overview

As is the norm in AI projects, we split our dataset into train, validation and test sets across our three different block-configuration sizes: 3, 4, and 5 blocks. As we saw earlier in subsection 3.2.2, within each block size, the number of states and total plan combinations are calculated using all pairwise combinations of distinct initial and goal states. This results in a near-exhaustive set of possible planning problems for each configuration size. The train, validation, and test splits follow a 70-10-20 ratio. This ensures that the model has access to a wide variety of examples during training while still being evaluated on unseen (and sometimes seen) scenarios during validation and testing. The details of the splits are as follows:

- **5-blocks:** The 5-block configuration produces 436 unique states, resulting in $436 \times 435 = 189{,}660$ problem statements. This is then split into:
  - Training set: 132,762 problems
  - Validation set: 18,966 problems
  - Test set: 37,932 problems
- **4-blocks:** The 4-block configuration produces 70 unique states, resulting in $70 \times 69 = 4{,}830$ problem statements. This is then split into:
  - Training set: 3,381 problems
  - Validation set: 483 problems
  - Test set: 966 problems
- **3-blocks:** The 3-block configuration produces 13 unique states, resulting in $13 \times 12 = 156$ problem statements. This is then split into:
  - Training set: 110 problems
  - Validation set: 15 problems
  - Test set: 31 problems

Contaminated Test Set As eluded to earlier, there is a slight adjustment made to the test set in these splits. More specifically, the test set is composed of two equal parts:

- 10% of the total data points form the **clean test set**
- 10% of the total data points form the **contaminated test set**.

This contaminated test set (i.e 10% of total examples) is included in the training data to simply observe whether performance improves on data the model has already seen during training - hence, the name. This design is to evaluate the model performance on examples it has seen during training versus entirely unseen ones.

Since we conduct one-shot prompting experiments, our effective training data size is halved to avoid any contamination between the demonstration and the actual problem statement within a single prompt. For the validation and test sets, we use examples from the training data to construct demonstrations. This is a deliberate move. Prior work suggests that the demonstration mostly helps the model learn the structure of action sequences, rather than contributing directly to planning ability. Moreover, since the model had already seen these examples during training, reusing them as demonstrations poses no risk of data leakage or unfair advantage.

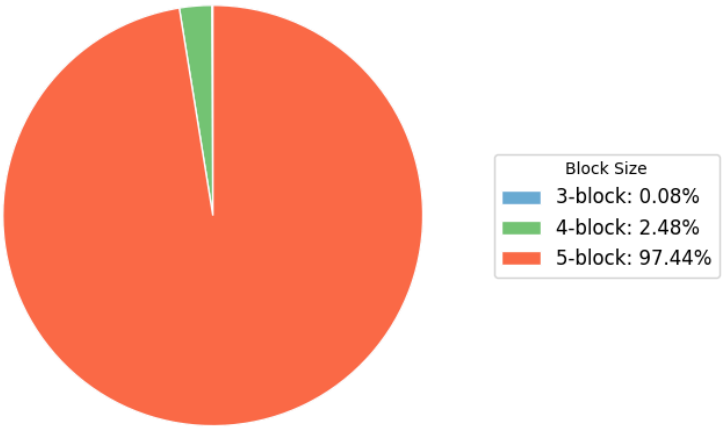

Figure 6: Overall % of dataset composition based on block size

Dataset Split Generation

In order to create our dataset splits, we first divide the set of unique problems into training, validation, and test sets according to the above mentioned percentages of 70-10-20. These splits are replicated across all block-set sizes.

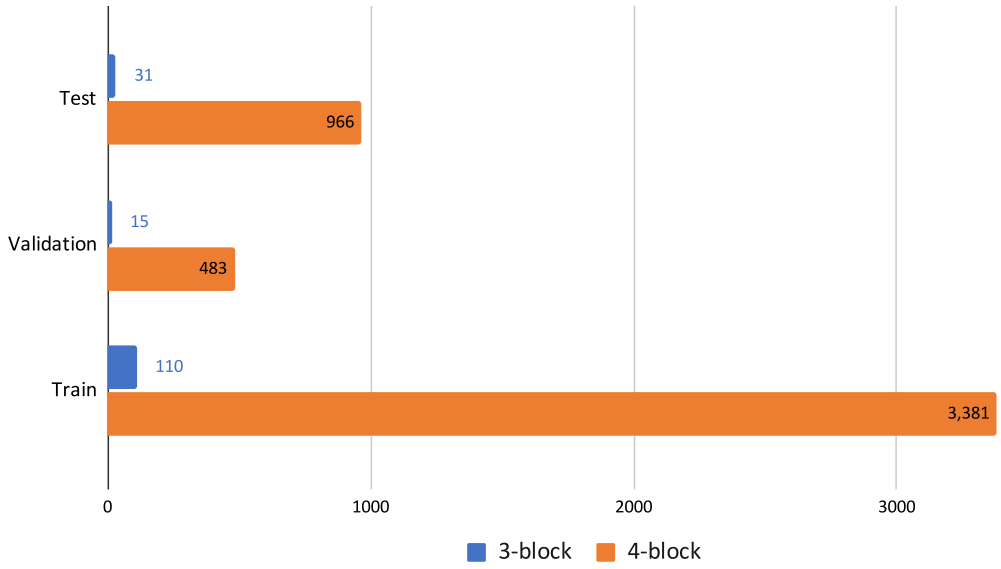

Figure 7: Train, validation, and test distribution for 3- and 4-block problems

For the training and validation sets, each unique problem is instantiated by randomly selecting one out of two colour sets. For example, a configuration involving blocks (1,2,3) has two possible mappings. The first maps numbered blocks to 1: 'violet', 2: 'teal', 3: 'brown', and the second maps them to 1: 'pink', 2: 'green', 3: 'red'. At the time of instantiation, the system randomly selects one of these two colour-maps. So the block configuration after colouring might look like (pink,green,red) or (violet,teal,brown) depending on the selection.

For the test set, we follow a slightly different strategy. Firstly, the test set is randomly split into two, pure test set and contaminated test set. For the test set, we follow a split-in-two strategy to create both clean and contaminated test subsets. We then begin colouring the configurations like so:

- **Clean test set (10%):** Each problem is instantiated using a single colour set. This problem is used only in the test set. These examples remain completely unseen by the model during training.
- **Contaminated test set (10%):** Each problem is instantiated twice using two different colour sets. One version is added to the training set, while the other is used as a test example. This allows us to analyse whether the model performs differently on problems it has already seen (albeit in different colours) during training versus the ones it has not.

This approach ensures that half of the test set consists of completely unseen examples (clean), while the other half includes problems the model has been exposed to (contaminated), but under a different colour configuration.

Scaling Effect with Increasing Block Count

As shown in the earlier section of Split Overview 3.2.4 and further elaborated on in figure 7 the data distribution across different block size subsets is highly skewed. While increasing the number of blocks: n, leads to a combinatorial growth in the number of possible states. Higher values of n also introduces significantly more complexity. More specifically a higher value of n tend to require longer and more intricate plans, as the goal states become more randomized and are often much further from the initial state in action steps. As explained in the upcoming sections, to avoid inundating the model at times the scope is limited to configurations with only 3 and 4 blocks.

3.3 Model Selection

3.3.1 Selection Criterion

We work with limited computational resources, including memory and training power, we have to confine ourselves to relatively smaller models (more specifically, in the less than 9B parameters range). Furthermore, since our task is that of multi-step planning it is crucial that the model is instruction fine-tuned and capable of following structured prompts. This combination narrows down our search to a select few open-source instruct models that are widely used. The following subsections describe the shortlisted models and the rationale behind selecting them for our experiments. But before we move to that, we need to take a small detour and look at how we extract coherent action sequences from LLM text and which techniques are used to parse, clean, and validate the generated plans for further evaluation.

3.3.2 Mathematical Representation of Planning Problems

In classical planning, a problem is commonly described by the tuple $P = \langle D, I, G \rangle$, where D is the domain, I is the initial state, and G is the goal. The domain itself is defined as $D = \langle F, A \rangle$, where F is the set of fluents—essentially predicates used to describe the world—and A is the set of possible actions. The state space of the problem is defined by all possible truth assignments over the fluents.
Each action is defined by its name, a set of variables, preconditions, and effects. The preconditions determine when the action is allowed (i.e., what must already be true in the current state), and the effects specify how the world changes after the action is applied.

The effects are further split into an `add` list (predicates made true) and a `delete` list (predicates made false). When the variables in an action are replaced with specific objects from the domain, the action is considered *grounded*; otherwise, it remains in its more abstract, *lifted* form.
For example, in the Blocksworld domain, the action of stacking a block on top of another block is written in our implementation of the framework as follows:

```
(:action stack
  :parameters (?below ?above)
  :precondition (and (clear ?below) (holding ?above))
  :effect (and (hand-empty) (clear ?above) (on ?above ?below)
               (not (clear ?below)) (not (holding ?above))))
```

Listing 3.1: Sample unified-planning implemented for blocksworld action for `stack`

In the above snippet of the `stack` action, the parameters define the objects being acted on—in this case, the block to be stacked and the block it will be placed upon. As mentioned earlier and further illustrated in the example, predicates (or fluents) are defined by their assigned Boolean values. In our case, there are five fluents: `on`, `clear`, `on_table`, `holding`, and `arm_empty`. In the snippet, first the preconditions state that the hand must be holding the above block in order to stack it on top of the below one (i.e., the predicate `(holding ?above)` is true). The effect specifies that after this action is executed the arm will be empty. The above block will now be successfully on top of the below block, which will no longer be clear. The arm will however be empty and thereby making the above block clear.
Likewise, all other actions namely: unstack, pick-up, put-down are defined in the framework to build executable plan objects. We will see in the coming subsections how one can link LLM responses to these well-defined actions. Following which we build simulation objects to evaluate validity of model generated action plans. Furthermore, we also avail the solver functionality of our framework to ground our problems and estimate optimal plans.

An Overview of our framework: unified-planning Library

The `unified-planning` library is a Python framework that allows us to model, execute and even solve classical planning problems in a standard way. As established above, we define the Blocksworld domain within this framework using PDDL i.e fluents, actions and domain objects. This enables the library to provide us with technical support to simulate and solve our problem statements. We will further see how we use it in this project to interpret, validate, and simulate the plans produced by the language model.

3.3.3 Translating Model Output into Executable Plans

As seen earlier in Figure 3.2.3, we use this general prompt template. Once the model is prompted, using regex simply first extract the plan text that appears between the `[PLAN]` and `[PLAN END]` tags.
Next, we split the extracted plan into individual action-sequences using newline characters as separators. Each action sequence is then converted into an action tuple. By removing unnecessary words we simplify the phrasing to create a stripped down version. Each tuple takes the form: `(action, object_1, object_2)`, where `object_2` is optional. This is because some actions, such as `put-down` or `pick-up`, only act on a single object.
After multiple rounds of experimentation with LLMs we realise that our expectations around the phrasing of action-sequences need to be less rigid. Improving with iterations, the model is now not required to follow the exact wording from the pre-defined action list, as long as the core meaning is preserved and the sequence remains logical. For instance, instead of requiring the full phrase "unstack the block1-name block from on top of the block2-name block," we also accept shorter versions such as "unstack block1-name from block2-name."

3.3.4 Building Executable Plans from Actions

Now that we have our action tuples, we can move on to building executable plan objects from them. Each tuple is passed through a mapping function that links it to a valid action in our implementation of the Blocksworld domain using the `unified-planning` library.
Once the individual action instances are generated, we use this list of actions to create a unified plan object. This plan object is then used to compute the relevant evaluation metrics required for assessing plan quality. This will be elaborated on further in chapter 4.
Now that we have established how model output responses are interpreted and verified within our framework, we can return to the core question of model selection. With this foundation in place, we are better equipped to assess which models are suited for high-quality generations, off course within our limitations.

3.3.5 Benchmarking Candidate Models

This section explores the process of selecting a suitable language model for plan generation. Several viable instruction-tuned models are tested based on output quality, planning coherence, and resource efficiency. We describe the models considered, the

evaluation strategy followed, and the final choice made.

As discussed earlier, all candidate models are open-source and fall within the 7–9B parameter range to ensure practical inference times and manageable memory requirements. We evaluate each candidate on its ability to produce structured syntactically and semantically correct action sequences. The models considered include: Qwen2-7B-Instruct, Qwen2.5-7B-Instruct, google/Gemma-1.1-7B-it and google/Gemma-2-9B-it, microsoft/phi-4-multimodal-instruct, Mistral-7B-Instruct-v0.1 and Mistral-7B-Instruct-v0.2.

Evaluation Metrics and dataset

To evaluate the model output quality, we define the following metrics. Some of these metrics are reused across other experimental set-ups as well, so it is crucial to introduce them clearly at this stage.

- **Attempts:** The number of attempts required by the model to produce a parsable and hence, format-compliant response. This reflects the model's consistency and reliability in adhering to the expected output structure.
- **Diff in plan length:** The absolute difference in the number of steps between the model-generated plan and the corresponding gold (ground-truth) plan. This is like a distance metric used to check for optimality conditions.
- **Terminate:** A boolean value checking whether generated plan eventually terminates to the goal state. A terminating plan indicates correct sequencing of actions that satisfy all goal conditions.
- **Action-steps away from goal:** An integer value indicating additional action steps required to reach the goal state from the current state of the environment. This gives a decent estimate of how far off the generated plan is from achieving the desired goal, with lower values indicating better performance.
- **Action-steps in gold plan:** The total number of steps in the ground-truth (gold) plan. This serves as a baseline for comparing the length and efficiency of the model-generated plan. Furthermore, this value provides us with an estimate of the complexity of the problem statement itself.

We construct experiments with initially, the basic version of the example prompt 3.2.3 and later add an additional helper string to it. We keep the temperature value fixed to 0.0 to enforce reproducibility. For this evaluation we use a 15 sample big subset of

3-block train dataset, as it is smaller in size while still posing a reasonable challenge for the models to handle.

Model Setup	**Mode (Attempts)**	**Mean (Diff in Plan Length)**	**Mean (Terminate)**	**Median (Steps from Goal)**	**Mode (Gold Plan Steps)**
Mistral-7B-Instruct-v0.3	3	—	0.00	4	4
Mistral-7B-Instruct-v0.1	3	—	0.00	4	4
microsoft/phi-4-multimodal	1	3.00	0.07	4	4
alibaba/QWEN-2.5-7b	1	0.50	0.13	4	4
alibaba/QWEN-2-7b	1	0.50	0.13	4	4
google/gemma-2-9b	**1**	**0.00**	**0.27**	**4**	**4**
google/gemma-1.1-7b	**1**	**0.00**	**0.27**	**4**	**4**

Table 3.1: Aggregated evaluation metrics for different candidate models

Findings from the Evaluation metrics

It is noteworthy that neither Mistral models are able to produce any coherent plans let alone plans that terminate even after mostly trying until the max attempts to query the model, i.e 3 queries per problem statement. For these models, the other columns such as terminate and steps from goal are not very meaningful, as they only offer insights once a comprehensible plan is available, which is entirely missing in this case.
In contrast, Microsoft's phi-4 model typically uses only a single attempt to generate coherent plans. While this plan does reach the goal state, it is not very optimal. However, this is balanced by the median (steps from goal) value of 4, which is a strong indicator of potential. This shows us that even though most model produced plans here do not terminate, they were only a few steps away from termination. Hence proving its metal in producing long traces of valid action plans.
Following this, both QWEN models exhibit similar performance across both sizes. Even though they are able to produce coherent plans mostly within a single attempt, they are sub-optimal and very few in number.
Finally, the google/gemma models perform the best. They have the highest terminate rate and all of the plans are within the optimal length. This is a strong sign of their promise as suitable model choice. On top of that, the low median (steps from goal) value further supports their reliability.

With these initial findings, we can clearly see potential in the QWEN, Gemma models and to some extent even phi-4 showing some potential. After minor prompt refinements[3] and a final round of evaluations, we initially select google/gemma-2-9b-it as our preferred model. However, early into our experimentation with prompting we discover that our API call to the gemma-2-9b are being redirected to gemma-3-12b-it. Luckily, since **gemma-3-12b-it** remaines within our hardware constraints and continued to perform well, we proceed with it as our final model choice.

3.4 Prompting Setup

3.4.1 Chain-of-Thought Prompting

We experiment with seven different prompt incantation to encourage the model generation of Chain of Thought reasoning trace. The first string is kept as the most widely used CoT incantation from existing literature. Building upon this, we design the other variants tailored more specifically to our planning-based use case. We also adapt the original CoT incantation proposed in the seminal CoT paper to better align with our task requirements. Additionally, we explore a delayed reasoning approach inspired by a recent study discussed in the Related Work 2 section, where the model is forced to "wait" before generating a plan, to evaluate whether such a simple method could perhaps, enhance reasoning quality.

We append each CoT incantation string to our base one-shot prompt example 3.2.3 along with the helper string. Then evaluate this on the 3-block validation dataset containing 15 problems. This dataset is ideal for running small-scale preliminary experiments. We compare the number of problems solved successfully by our chosen model with and without the incantation. This change in correctness rate helps us quantify the extent to which the incantation truly makes a difference. The seven CoT incantations used for initial experiments on the small batch are listed below, along with the respective change in correctness rate they bring as compared to baseline results.

[3] We append a helper string to promote format compliance: "Answer within [PLAN] [PLAN END] tags". With this minor adjustment our chosen model successfully produces almost all generations according to the expected format.

S. No.	Incantation	Change in Rate
1.	Let's think step by step	0
2.	**Step by step reason about the actions you choose to take**	**+2**
3.	Step by step reason about actions you play.	0
4.	Step by step reason about actions you take.	-1
5.	**Strategise at each step before you choose an action, then form a final plan.**	**-4**
6.	Wait and then	-6
7.	Wait, reason step-by-step about the actions you choose to take. Only answer within '[PLAN] [PLAN END]' tags.	+2

Table 3.2: Impact of CoT Incantations on Plan Correctness Rate Relative to Baseline (No CoT Prompt)

Following our preliminary experiments, we want to further evaluate how the performance trend scales on larger datasets. Specifically, we focus on two CoT incantations whose results were particularly noteworthy: one that showed a clear performance improvement (hereafter called the : "good" incantation). The other that intuitively seemed promising but performed poorly in practice (hereafter called the: "not-so-good" incantation). Both of these have been highlighted in the table above. Our goal is to investigate how these effects change with scale—whether the improvements hold, diminish, or become more noticeable. We also want to understand whether the observed gains can be truly ascribed to the quality of the incantation or if any added prompt string could yield similar results. To this end, we compare the change in correctness metric again, this time for both incantations across datasets.

For this evaluation we use both subsets from both 3 and 4 block configurations. Our aim is to evaluate on a large enough dataset for our purposes and yet not highly complex like the 5-block configuration. Additionally, since this setup uses inference-only generation, there is no risk of data contamination. Similar to the previous experiment, the CoT incantation's result is then compared again to the baseline for the respective data subset. Following this comparison, we analyse how the improvements behave with scale.

The table illustrates the percentage of correctly terminating plans (i.e., plans that successfully reached the goal state). The values are computed under three conditions: baseline (no incantation), with the effective CoT incantation, and with the not-so-good one. We observe that across all splits, both incantations improve plan correctness significantly over the baseline. So in general, we note that it is better to include some

Split, n = Number of blocks	**Baseline (no CoT)**	**"Good" incantation**	**Not-so-good incantation**
Val, n = 3	7/15 = 0.46	8/15 = 0.53	6/15 = 0.40
Test, n = 3	10/31 = 0.32	13/31 = 0.42	16/31 = 0.52
Val, n = 4	56/483 = 0.12	134/483 = 0.28	153/483 = 0.32
Test, n = 4	95/966 = 0.098	262/966 = 0.27	298/966 = 0.31

Table 3.3: Effect of CoT incantations on plan correctness with Scale

form of guided reasoning rather than none at all. Even though the two strings perform quite differently, both manage to shift the model's output closer to valid plans.
However, the extent of this improvement is strongly influenced by the scale of the task. While in the 3-block configuration the effective string outperforms the weaker one by 10%. As the complexity increases and the data size increases i.e in the 4-block configuration we can see that the performance of both incantations begins to converge, with a relatively minute difference in correctness percentages. This suggests that with sufficient task complexity and scale, even less effective prompts may benefit from guided reasoning.

3.5 Supervised Fine-Tuning (SFT)

Following the prompting setup, we now describe our Supervised Fine-Tuning pipeline, which adapts the model to structured reasoning through example-based learning. As discussed in Chapter 2, SFT is a widely used approach for adapting large language models to downstream tasks. SFT has been studied to enable LLM to internalise task-specific patterns beyond what is possible through prompting alone. In our context, this ability plays a crucial role in reinforcing reasoning strategies.

In the following sections, we detail the preparation of input data for SFT, along with the model architecture, training and evaluation setups.

3.5.1 Overview of SFT

SFT is a distinctly different approach to planning with llms than the previously discussed setup. In this paradigm, we hope to elicit reasoning from models through a much more direct exposure to expected output. SFT helps the model learn how to reason by training it on examples of correct behaviour. While prompting gives the model instructions at test time, SFT goes a step further by showing the model many examples during training, in the hopes that it learns patterns more deeply. This is especially useful for tasks that

require careful, step-by-step thinking, like planning. Expectation is that with enough training, model will eventually start to pick up the logic and structure behind good answers.

Going into this task, this aptly sums up our motivation. We aim to reinforce the reasoning process by letting the model see well-structured examples of input and output pairs. As illustrated in the previous section describing data generation 3.2.3, with the aid of `unified-planning` library we are able to produce ground truth, also referred to as gold plans. These gold plans act as expected outputs in our SFT setup. In doing so, we hope the model begins to realise useful strategies. Strategies such as, how to break down a goal into smaller steps, or how to choose valid actions. Like any supervised learning tasks, these learned tactics can be applied at test time, when solving new problems.

3.5.2 Preparation of Input Data

Pre-processing Data

An earlier section has already detailed the process of transforming our generated the dataset into input prompts, culminating in the final prompt template shown in Figure 3.2.3. For SFT purposes, we further process this prompt to fit the format expected by the chosen model during training. This includes pairing each prompt with its corresponding ground truth output thus ensuring the input-output pairs follow the causal language modelling format. Furthermore we add special tokens if required by the tokenizer, and finally, tokenize this resulting data file.

We use a chat template to format our inputs before giving them to the model. This template follows a structure similar to how users and assistants interact in a chat. It wraps the prompt in special tokens like `<|user|>` and `<|assistant|>`, which help the model understand who is speaking and what kind of response is expected.
Using a chat template is especially helpful for instruction fine-tuned models like ours. These models have been trained on large datasets of user-assistant conversations. As a result, they are known to perform better when inputs follow a familiar chat format. By sticking to this style, we want to make it easier for the model to follow instructions and generate responses that align with our planning task.

In our task, **causal language modelling** is the training method we use during SFT. This technique works by teaching the model to predict the next word (or token) in a sequence, one step at a time, from left to right. This fits well with our planning task,

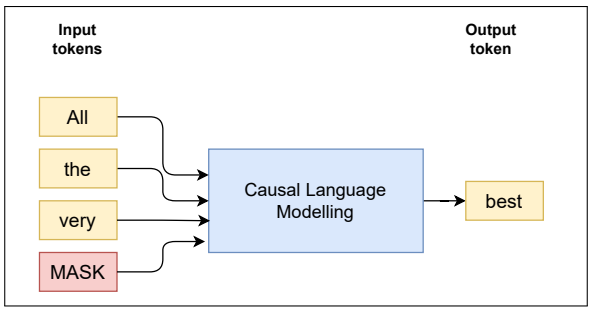

Figure 8: This figure illustrates causal language modelling, where the model predicts the next token in a sequence using all previous tokens as input.

where each action depends on the ones before it. On this regard, after pairing our one-shot prompt with its respective ground truth plan, we append this expected outcome to get the model input. For the expected outcome, also referred to as labels, we simply duplicate the input however, this time the input tokens are masked. Masking ensures that the model does not see those tokens at training time. This is a standard practice in causal language modelling tasks like ours. This practice is used to make sure the model focuses on learning how to generate the correct continuation, rather than unnecessarily reproducing tokens from the input.

Tokenisation Pipeline

Once the input-output pairs are formatted using the chat template, we convert them into token IDs using the Gemma-3 model's tokenizer provided by hugging-face. [4] This step transforms our textual data into a numerical format that the model can understand and process. Tokenisation is done in a way that preserves the structure of the prompt, including special tokens like eos, `<|user|>` and `<|assistant|>` etc. Even though it is not a concern for our project due to the model choice and dataset being used, tokenisation also ensures the final sequence stays within the model's context length.

To better understand the structure and complexity of our dataset, we will now analyse the distribution of input token lengths across the 3, 4, and 5-block configurations. This helps verify that the inputs fall within the model's context window. Furthermore, help us make decisions around batch size and training stability. Longer prompts intuitively reflect more complex planning scenarios. This may increase the reasoning burden on the model.

[4] Please note, at train time we also need to pass a task specific custom data-collator object from HuggingFace to the model trainer class to enable causal language modelling at train time.

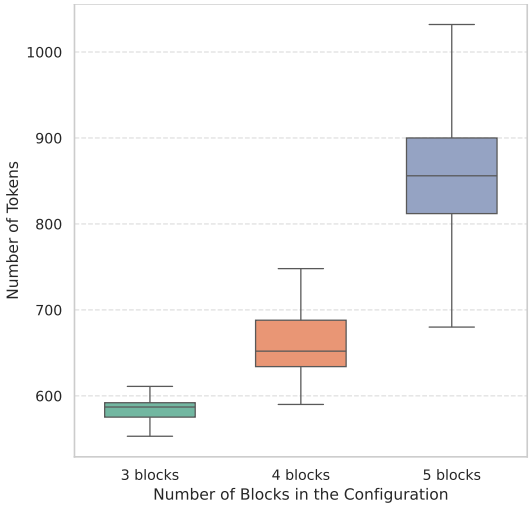

Figure 9: Token length distribution across block-set configuration

The following figure (3.5.2) shows how the number of input tokens increases with the number of blocks in the planning problem. Each box plot represents a different block configuration: 3, 4, 5 blocks and the respective distribution of token lengths for each. As expected, problems with more blocks result in longer input prompts. This is because more blocks introduce additional objects, relations, and actions, which naturally lead to longer state descriptions and plans.

The token count gradually increases from around 580 tokens for 3 blocks, to 650–750 for 4 blocks, and further to 850–1000+ for 5 blocks. The variability also grows with complexity, especially in the 5-block case. Since we use causal language modelling for SFT, input and output sequences must be aligned in length. Based on this distribution, we set a padding threshold of 1200 tokens for all inputs. We use the default pad token with id 0. This is further explicitly added as an ignore index at training time. This comfortably covers the longest examples in our dataset, even for the 5-block configuration. Moreover, our chosen model, `gemma-3b-12b-it`, supports long context windows, so this sequence length remains well within its capabilities.

3.5.3 SFT Training Implementation

We implement SFT using the Hugging Face transformers and trl libraries. As established in the section about Model Selection (section 3.3) the model we will be working with from this point onward is `gemma-3b-12b-it` from Hugging Face. The training objective

follows causal language modelling, where the model is trained to predict the next token in the sequence given all previous tokens. Experimentation is done on NVIDIA A40 GPUs, which come with a few constraints in terms of memory capacity and compute power. Due to this consideration, we decide to use a Parameter-Efficient Fine-Tuning (PEFT) technique: Low-Rank Adaptation (LoRA) for our train implementation.

Low-Rank Adaptation

LoRA is a technique that helps reduce the number of trainable parameters and makes training more efficient without needing to update the full model. In the attention layers, this PEFT method takes a large matrix and decomposes it into two smaller low-rank matrices (3.5.3), which results in a drastic reduction of several parameters, which needs fine-tuning. This making it feasible to tackle large models even with conservative hardware setups. In their paper, the LoRA authors also claim that this method achieves comparable accuracy to full fine-tuning across various tasks. Whether this stands true for our project, is yet to be tested. A deeper understanding of LoRA can be developed in the literature survey chapter 2.

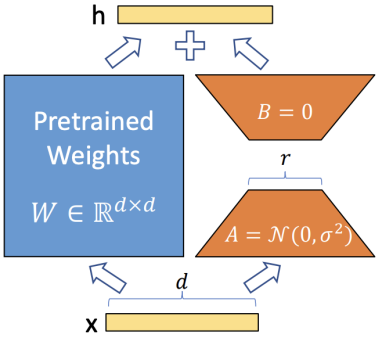

Figure 10: Representation of LoRA, where only A and B matrix weights are trained.

Understanding Loss Computation in SFT

During training, the model is given an input and an expected output. It tries to predict the next word in the output, one word at a time. To achieve this, the expected output is shifted by one position so the model can learn to guess the next word correctly. The model's guesses (called logits) are compared with the actual next words using the cross-entropy loss function. This shows how far off the model was. The model then updates its weights and biases to reduce this error and improve its predictions for the next iteration. Following is the algorithm's mathematical representation:

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{T} \log P_\theta(y_t \mid x, y_{<t}) \tag{3.1}$$

This equation represents the standard loss used during Supervised Fine-Tuning. The model, with parameters θ, is trained to generate the correct output sequence $y = (y_1, y_2, ..., y_T)$ given an input prompt x. At each timestep t, the model predicts the next token y_t conditioned on the input and all previously generated tokens $y_{<t}$. The training process adjusts the model's weights to maximise the likelihood of the correct output sequence across all training examples.

In our case, the input x consists of a long prompt containing the Blocksworld state, goals, and reasoning instructions, while y is the desired response, such as a Chain-of-Thought trace or plan. This formulation allows the model to learn fine-grained patterns over long sequences and helps it internalise the structure of valid reasoning.
The exact values for key hyperparameters like the number of training epochs, batch size, learning rate, gradient accumulation steps, weight decay etc. will be discussed in the following chapter. Choices are made to balance performance and memory usage, especially considering that we train on relatively long sequences. To reduce memory load, we enable gradient checkpointing and use LoRA for PEFT. This allows us to train large models like gemma-3b-12b-it even on limited hardware.

While this section covers the base setup, and gives an understanding of the workings of SFT; further specifications with regard to experimentations are discussed in the Experimentation and Results chapter 4.

3.6 Reinforcement learning with GRPO

We have now established an understanding of two major setups. Next attempt at eliciting reasoning from LLMs involves stepping into the paradigm of Reinforcement Learning. In

the literature survey (chapter 2) we discuss multiple studies that demonstrate the use of reinforcement learning with language models. As discussed in the previous chapter, this tactic has gained traction in the language modelling space primarily through the Reinforcement Learning from Human Feedback (RLHF) framework. In this framework, a reward model trained on human preferences guides the policy optimisation of a language model. This approach was notably used in training InstructGPT (Ouyang, J. Wu, Jiang, et al. 2022) and later in ChatGPT, leading to significant improvements in alignment and helpfulness. More recent work has explored strategies besides RLHF for reasoning and planning. Strategies such as: Reinforcement Learning with AI Feedback (RLAIF), Reinforcement Learning with Verified Rewards (RLVR) and Agentic learning.

These efforts mark a clear shift from purely imitation based fine tuning towards iterative interaction and feedback, allowing models to refine their behaviour based on long-term objectives. In our work, we adopt the Reinforcement Learning with Verified Rewards approach, specifically using the method recently published by the DeepSeek team (DeepSeek-AI, D. Guo, D. Yang, et al. 2025) and promoted by fellow researchers for tasks like ours, GRPO. This approach offers academic research a novel opportunity to enhance the abilities of smaller models. Leveraging our choice of a classical planning dataset Blocksworld to our advantage we harness the capabilities of `unified-planning` library to simulate state-wise game conditions and generate granular, tangible reward signals for training our system.

In the upcoming sections we discuss GRPO's algorithm, input data preparation and finally, designing a reward signal. Additionally we further elaborate our training and evaluation setups.

3.6.1 Introduction to GRPO

Guided Reinforcement with Policy Optimisation (GRPO) is a method for improving language models using reinforcement learning. Instead of relying solely on human feedback, it guides the model with clear, verifiable rewards that check if the output meets the desired format and/or solves the task correctly. This makes it easier to train models for reasoning and planning without needing large amounts of human-labelled data.

Traditional methods like RLHF rely heavily on human feedback to guide the model towards desired behaviour. While this is proven to be effective for alignment, it is an expensive and slow approach. In the literature survey chapter we also decode how this method often struggles with tasks that need precise reasoning or planning, because humans cannot always quantitively verify every step or long-term outcome.

As a result, over time, research evolved towards methods like RLAIF and RLVR, where AI-generated or programmatically verifiable feedback replaces human evaluation. GRPO builds on this idea by using structured, verifiable rewards that check whether outputs follow the correct format and/or solve a problem step by step. This evolution has made reinforcement learning scalable, cheaper, and more reliable for reasoning tasks. In our

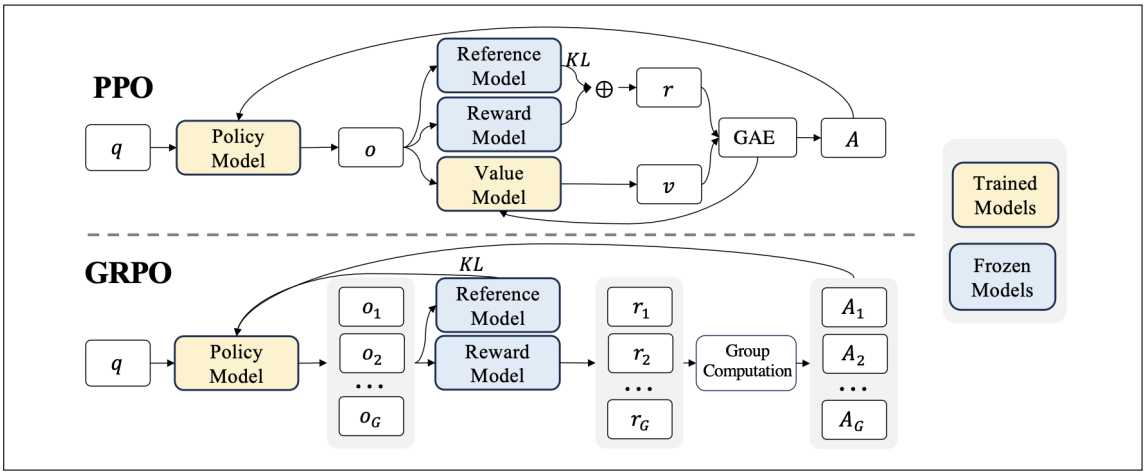

Figure 11: This figure from the GRPO paper (Z. Shao, P. Wang, Zhu, et al. 2024) demonstrates the difference between Proximal Policy Optimization (PPO) (existing) and GRPO (new) RL algorithms. It is important to note that, GRPO forgoes the value model instead, estimating baseline from group scores. This significantly reduces training resources.

work, GRPO fits perfectly because our Blocksworld planning tasks can provide clear, state-wise rewards through `unified-planning`, allowing our smaller model to learn effectively without constant human supervision. Furthermore, GRPO uses average rewards over multiple generations as a guiding signal, thus eliminating the need for a trainable value model. This makes the training process lighter, and well-suited for our academic experimentation with limited resources. Since its launch, various articles have been published by the industry on eliciting reasoning in smaller models using this training algorithm. These articles highlight the potential for improving structured reasoning in planning tasks, and low-resource fine-tuning. We have also utilised these articles as a reference point for this emerging technology.

3.6.2 Input Data Preparation

Data Adaptation for GRPO

In a previous section we explain how our generated dataset was converted into input prompts, leading to the final prompt template shown in Figure 3.2.3. For GRPO purposes, we further alter this prompt template to align with the expected input by the algorithm. On this note, we add a system prompt inspired by the DeepSeek paper to encourage generation of reasoning traces by the model within the <think> </think> tags. These reasoning traces are vital in our GRPO setup because, similar to DeepSeek-R1, they are expected to force the model to "think" before forming a final action plan. In our case, the <think> tags capture the model's planning steps in Blocksworld, the plan following this can be automatically validated against the simulated state transitions in `unified-planning`.

System Prompt for GRPO

```
I am a blocksworld plan generator.  I first think about the reasoning
process in the mind and then provide the user with the plan.  The
reasoning process and plan are enclosed within <think> </think> and
[PLAN] [PLAN END] tags, respectively, i.e., <think> reasoning process
here </think> [PLAN] plan here [PLAN END].
```

Figure 12: System Prompt for GRPO

This system prompt will be appended to our base one-shot prompt template to generate input prompt for GRPO implementation. [5] Furthermore, as discussed in the previous section outlining SFT, our base model is an instruction fine-tuned model. As a result of which, we use a chat template to format our inputs before giving them to the model. This template follows a structure similar to how users and assistants interact in a chat. This wrapping formats the prompt with special tokens: <|user|> and <|assistant|>, to emulate a chat-like conversation between the model and the user.

Tokenisation Process

As established earlier, we use HuggingFace to deploy language model related tools. After adding the necessary instructions and task-specific tokens to the input prompt, we move on to tokenising our dataset. Unlike SFT, the GRPO setup does not expect

[5] It is to be noted that, even though the expected outcome from the model is a plan in the Blocksworld domain along with its reasoning behind generating that particular plan, the model only receives a problem statement followed by a correct plan. We **do not provide** any reasoning traces to the model during training.

input–output pairs as model input. As a result, the algorithm does not require symmetry in input sequence lengths, so we do not need to apply padding. However, we do compute the maximum prompt length and pass it as an important parameter to the trainer. This is done to ensure proper batching and prevent the model from exceeding its context window during training. Even though we tokenise our data beforehand using the Gemma-3 tokeniser provided by Hugging Face, manual inspection revealed that the GRPO implementation computes input tokens dynamically at training time. Therefore, in addition to pre-tokenised data, we also pass a processor class object that contains the tokeniser to the training pipeline. The trainer's processor is passed so it can handle tokenisation dynamically.

The exact distribution of tokens, apart from the fixed system prompt, stays the same as shown in the earlier diagram (3.5.2). A more detailed look at scaling complexity shows that token length, batch size, and policy update steps directly affect GRPO training speed. Longer inputs increase memory use and computation because the model must generate outputs, calculate log-probabilities, and compute rewards for every token. Choosing the right maximum prompt length and batch size is important to keep training efficient and avoid running out of GPU memory. A problem we look at in more detail in the upcoming chapter.

3.6.3 Reward Signal Design

In GRPO, the reward signal guides the model towards producing outputs that meet the task requirements. As eluded to earlier, each generated response is scored by a reward function, which in our setup checks for correct output format and task completion. We achieve the former using regular expressions and the latter is accomplished with the help of `unified-planning`. Exact matches in format reward receive higher rewards, while partial matches receive lower rewards. Furthermore, for plan rewards we follow a process reward strategy instead of a outcome-based reward method. Designing the reward signal requires care and consideration. Rewards that are too simple and frequent, have been shown to lead to reward hacking. This phenomenon is when the model exploits shortcuts to maximise the score without truly solving the intended task. For example, our model could repeatedly unstack and stack the same block to produce valid actions, without ever progressing towards the desired goal state. Similarly, sparse or inconsistent rewards can slow down learning and cause unstable updates.To avoid this, rewards are normalised within each batch of completions, and a KL-divergence penalty is used to prevent the model from drifting too far from the base policy.

This balanced design ensures that the reward signal is both informative and

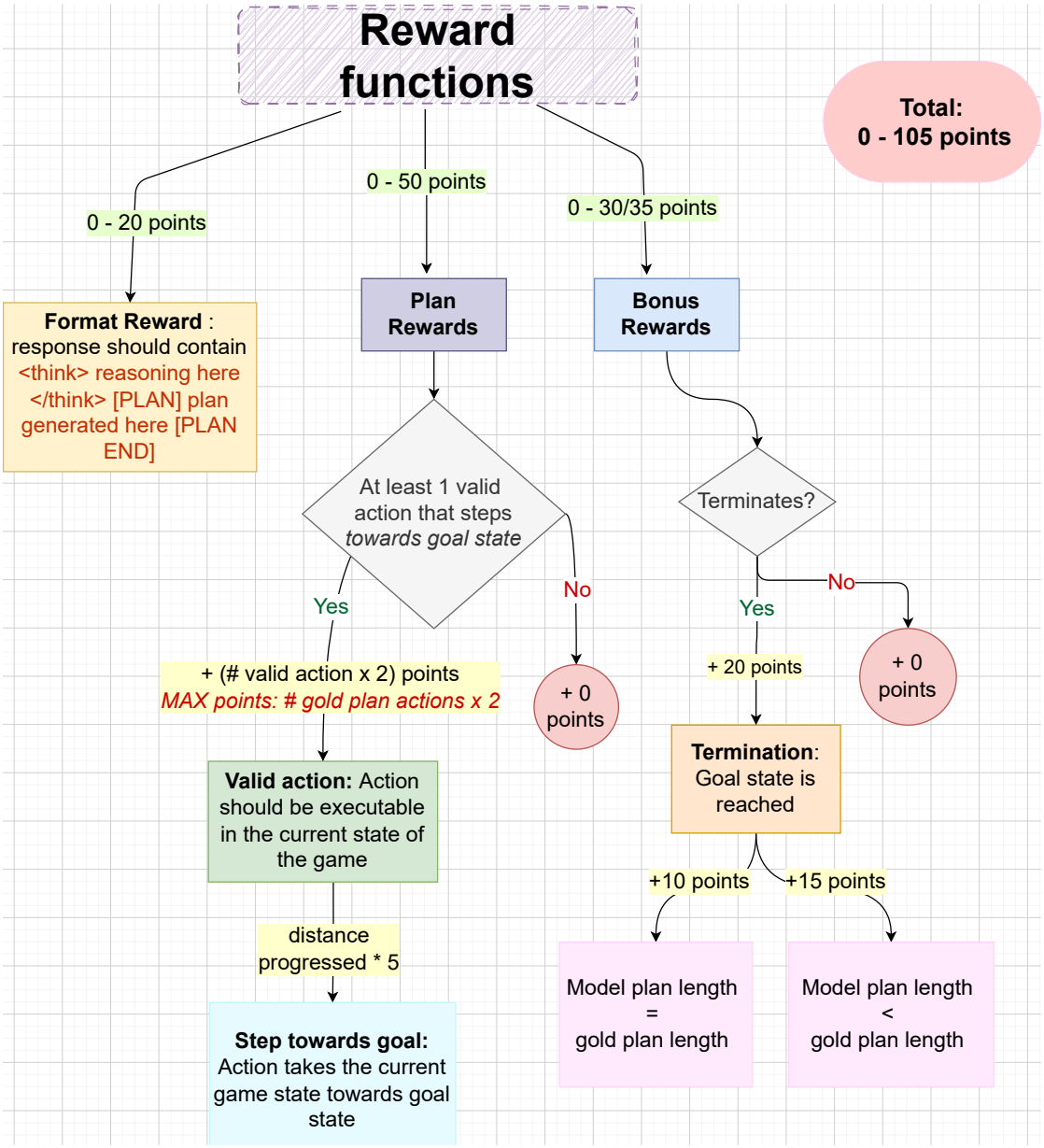

Figure 13: This figure shows the three different reward functions used to guide GRPO training of our language model.

stable, allowing GRPO to effectively guide the model towards generating useful and correctly formatted outputs. To this extent, we experiment with multiple reward signal renditions. Finally, we settle at a reward signal that lies in the range of 0 to 100 or 105 points. The is structure we eventually land on is illustrated in Figure 13.

Format Reward Function

Let us first look at the format reward function. Format rewards lies in the range of 0 to 20 points, where 0 is the lowest and 20 is the highest value a model response can be awarded by this function. The motivation behind assigning specific values is to encourage the model to generate clean, well-structured, and as a result easily parsable responses. We use structured start and end tags as this makes it easier to extract information from the output. Since LLMs often generate gibberish or inconsistent formats, having such a reward function is especially important in RLVR setups. In our prompt to the model (Figure 3.6.2) we have two main sections in the expected output: think and plan. Hence, in our format reward design we have awarded points based on output's structural adherence to these two sections. Additionally, we have deducted points for responses that contain text within the tags albeit with gibberish around them. This is put in place to discourage unnecessary consumption of tokens on nonsensical ramblings that sometimes these models are prone to do. The following table gives a clear description of possible scenarios with regards to model response structure and points assigned in the respective situations.

Table 3.4: Format rewards bi-furcation

Gibberish	Plan tags	Think tags	Points
No	No	No	0
No	No	Yes	7
No	Yes	No	3
No	Yes	Yes	20
Yes	No	No	0
Yes	No	Yes	5
Yes	Yes	No	2
Yes	Yes	Yes	15

Table 3.5: This table presents the format reward scores assigned to model outputs based on the presence or absence of specific response tags. This encourages the generation of think traces and promotes well-structured outputs. It simplifies parsing and further allows for a more effective assignment of plan rewards.

As we can see in the figure, production of think traces is rewarded solely through this

reward function. As a result we notice that in the assigned reward points, generating think traces is valued more than producing plan sequences in format reward function. This is a deliberate design choice to encourage the model to reflect and reason before committing to an action. This is a method, studies claim, that produces the coveted 'AHA' moment [6] in models. We are hoping to evaluate this phenomena's occurrence for our use case. Based on our above function, to receive a maximum of 20 points, the model needs to produce reasoning trace, followed by a plan action sequence enclosed within the proper tags and with no other text outside these tags.

Plan Reward Function

Now, we move on to the plan reward function. This component is responsible for evaluating whether the model-generated action sequences lead toward a valid plan navigating its way to the desired goal. Everything contained within the start and end plan tags is processed by this function. Both plan validity and checking our distance from desired goal state is computed with the help of `unified-planning` library. For each model response, we create a new simulation and run the sequence of actions one by one. Plan rewards are computed dynamically to allow the model complete flexibility to generate plans that deviate from the gold plan, as long as they achieve the desired outcome. Additionally, it is important to note that we do not rely solely on final outcome rewards. Instead, we follow a process reward strategy. This means that if a model-generated response contains six action steps, and only four of them are correct, the model is still rewarded for those four correct steps. We believe process-based rewards are more suitable for our use case, as they allow us to nudge gradual progress toward the goal rather than expecting the model to get everything right in one go. This aligns well with how humans also learn, by improving step by step. The plan reward is used to check if the model is producing useful actions that help reach the goal. It can give up to 50 points. To get any reward here, the model must begin with a valid action (Figure: 14). A valid action means that this action can be performed in the current state of the game. If there are no valid actions, the model gets 0 points. Once a valid action is established, the model earns 2 points per valid action, up to a maximum of twice the number of actions in the gold plan. This upper limit is set to clip the possibility of gaining higher rewards through redundant actions like stacking and unstacking the same object consecutively. Furthermore, each valid action that moves the game state closer to the desired state, earns 5 points. This movement is discerned by computing the number of actions needed to reach the goal state from our current game state. Notice that we

[6] The 'AHA' moment refers to a sudden flash of insight or understanding that occurs during reasoning. In the context of language models, this is a point where the model, after engaging in step-by-step thinking, arrives at a correct or insightful conclusion that it may not have reached otherwise.

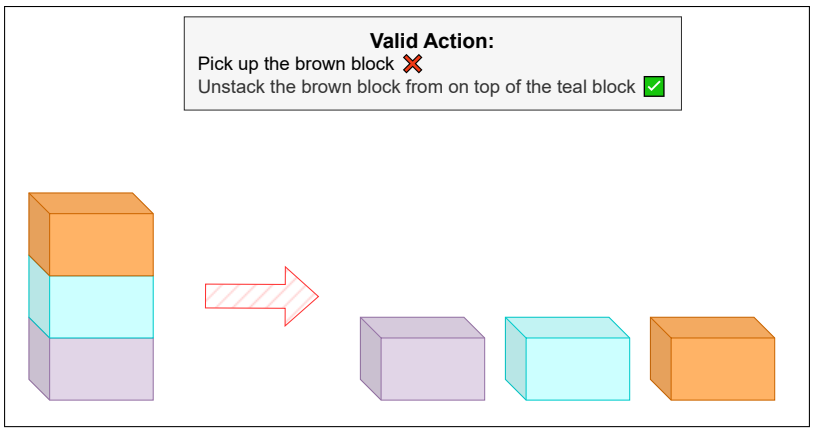

Figure 14: This figure illustrates the same example configuration in our Blocksworld domain, as seen in a previous section (Figure 4). This time with this example configuration we can understand meaning of valid and invalid actions in a given state of blocks. To recap quickly: in this configuration, the left state of blocks is initial configuration and the right one is goal configuration. Given the initial state of blocks, top action is deemed invalid, whereas bottom one is valid in our domain.

deliberately assign a slightly higher score to distance progressed (Figure 15) compared to valid plan actions. This is done to encourage the model to go beyond just using the correct syntax or action format. We want to reward it more for actually planning in a way that moves towards the goal state. After we compute the total plan reward for the model's response, we also calculate the reward for the gold (reference) plan. The model's score is then normalised using the gold plan score so that it fits within the set range of 0 to 50 points. This helps us fairly compare different responses and ensures consistency across examples.

This reward is structured in a way to help guide the model to not only make correct moves but also to move in the right direction, step by step, toward the final goal. It avoids rewarding random or irrelevant actions and its designed to ensure the plan is actually useful in solving the task.

Bonus Reward Function

Finally, we can decode the bonus reward function. This function is set in place to reward the model separately, and heftily for reaching the desired goal state. This extra reward hopefully helps elicit complete, optimal and correct plans by making successful end

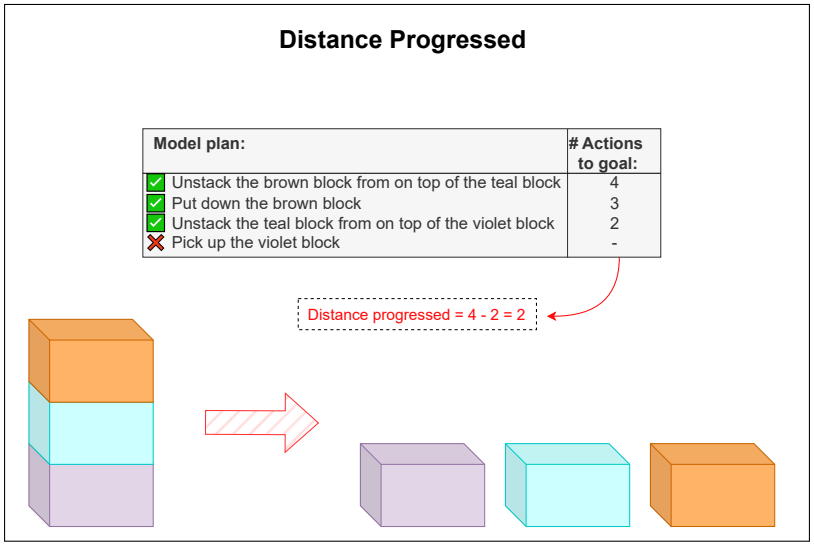

Figure 15: In the same initial and goal configuration of blocks, we now showcase a clear depiction of how distance progressed metric is computed for a model response. As the valid actions are run on a simulated game, we compute number of actions to the goal state for each. Finally, we take the difference between actions to goal from last valid action and initial state. This difference is distance progressed by acting on model generated plan.

states more desirable than partially correct intermediate steps.

This function can give up to thirty five points. When a generated plan terminates successfully, model receives a positive score of twenty; otherwise, it is awarded no points. Additionally, for these terminated plans we check their optimality in number of actions. When the number of actions used to reach termination (i.e desired goal state) is consistent with those in gold plan, the model is awarded an additional ten points. However, in presumably rare occasions[7] when the model plan outperforms the gold plan with regards to optimal plan length, it is able to garner an additional fifteen points instead of ten points. Our motivation behind setting such values for this function is to go beyond plan validation, we intend to encourage the model to strive for optimality.

3.6.4 Training Setup

Our model selection remains consistent across all three setups. As detailed in the model selection section (Section 3.3), we utilise the `gemma-3b-12b-it` model from Hugging Face. Consistent with our previous approach in SFT, we apply PEFT using LoRA. For training, we employ Hugging Face's implementation of GRPO through the transformers and trl libraries.

RL: Memory and Compute Considerations

Initial experiments are conducted on NVIDIA A40 GPUs, which offer limited memory and compute capacity. To mitigate these constraints, we use LoRA to reduce training resource requirements. However, we quickly observe that GRPO introduces novel memory consumption requirements and patterns. Unlike standard training, this algorithm requires the model to generate multiple outputs per input. These inputs are then evaluated, based on which our model makes an update in the weights and biases. This increases computational and storage demands. To address this, we initially utilise Unsloth, an optimisation library. Unsloth is a Python library designed to accelerate and optimise fine-tuning of LLMs through various optimisation strategies. It claims to make model training and fine-tuning extremely fast, memory-efficient, and easy, especially on limited hardware like ours. These features met with our initial need for fine-tuning the model with reduced memory requirements and faster training times.

Despite its advantages, with Unsloth, we observe unpredictable memory consumption and performance degradation due to aggressive optimisations. As a result

[7] In rare cases, it is possible that the generated gold plan has redundancy. This phenomenon is noticed more as the number of block-set size and problem complexity increases.

beyond our dummy experiments, we decide to drop Unsloth and switch to a different solution. To overcome pur existing limitations, we transition to a more powerful system equipped with NVIDIA H100 GPU. On the H100, we conduct PEFT training without Unsloth. We even put a few memory optimisation techniques that we observe are missing from our version of trl's GRPO trainer. This transition allows us to handle GRPO training more efficiently, reducing memory bottlenecks and improving training speed. The enhanced hardware resources are particularly beneficial, as reinforcement learning-based methods like GRPO are generally slower than standard supervised training.

Policy Updates in GRPO: Brief Overview

In Group Relative Policy Optimization, the update step is a crucial phase that refines the policy to improve performance on the target task. Unlike traditional algorithms such as PPO, GRPO eliminates the need for a separate value function or critic network. Instead, the algorithm leverages the relative ranking of rewards across a group of sampled trajectories.
During each train iteration, the model samples a batch of trajectories or generations by interacting with the environment using the current policy model. For each generation, GRPO computes the total reward. It then compares the rewards within the batch or group, ranking plan rewards according to their performance rewards. The policy update is performed using a group-wise relative advantage, which encourages the policy to increase the probability of actions leading to higher rewards within the sampled set.

$$\theta' = \theta + \alpha \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[A_{\mathrm{GR}}(\tau)] \tag{3.2}$$

Mathematically, the policy parameters are adjusted using a gradient ascent step based on these relative advantages. This approach helps stabilize training by focusing updates on action-plans that are empirically better within each group, rather than relying on absolute value estimates. As a result, the agent can efficiently learn improved behaviours, even in environments where reward signals are sparse or noisy. Furthermore, as eluded to in the reward signal design section, the algorithm applies these updates conservatively using KL-divergence regularisation method to avoid steering too far away from existing policy.

$$\mathcal{L}_{\mathrm{GRPO}}(\theta) = -\frac{1}{\sum_{i=1}^{G} |o_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \left[\frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{[\pi_\theta(o_{i,t}|q, o_{i,<t})]_{\text{no grad}}} \hat{A}_{i,t} - \beta \mathbb{D}_{\mathrm{KL}} \left[\pi_\theta \parallel \pi_{\mathrm{ref}}\right] \right] \tag{3.3}$$

The exact values for key hyperparameters like the number of training epochs, batch size, learning rate, number of generations, gradient accumulation steps, weight decay etc. will be discussed in the following chapter. Choices are made to balance performance and memory usage, especially considering that we train on relatively long sequences. Furthermore, we will also discuss strategies employed for efficient train runs on limited hardware.

While this section covers the base setup, and gives an understanding of the workings of GRPO; further specifications with regard to experimentations are discussed in the Experimentation and Results chapter 4.

4 Experimentation and Results

GRPO: memory optmization techniques, maybe put in limitations

5 Final discussion, Limitations and Future Work

Acronyms

AI Artificial Intelligence. 1, 2

CoT Chain of Thought. 2, 4

GPT Generative Pre-Trained Transformer. 1

GRPO Group Relative Policy Optimization. 2, 36, 45–47

LLM Large Language Model. 1, 2, 13, 14, 16, 30, 35, 41, 45

LoRA Low-Rank Adaptation. 34, 35, 45

LRM Language Reasoning Model. 1, 2

NL Natural Language. 16, 18

PEFT Parameter-Efficient Fine-Tuning. 34, 35, 45

PPO Proximal Policy Optimization. 37, 46, 51

RL Reinforcement Learning. 2, 13

RLAIF Reinforcement Learning with AI Feedback. 36

RLHF Reinforcement Learning from Human Feedback. 36

RLVR Reinforcement Learning with Verified Rewards. 36

SFT Supervised Fine-Tuning. 2, 13, 30, 31, 33, 35, 45

List of Figures

List of Tables

Bibliography

Ahn, M., A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, and et al. (2022). "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances." In: *arXiv preprint arXiv:2204.01691*.

Asai, M. and A. Fukunaga (2018). "Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Chevalier-Boisvert, M., D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen, and Y. Bengio (2019). "BabyAI: A Platform for Studying the Sample Efficiency of Grounded Language Learning." In: *International Conference on Learning Representations (ICLR)*.

Cobbe, K., V. Kosaraju, M. Bavarian, M. Chen, H. Jun, Ł. Kaiser, M. Plappert, J. Tworek, J. Hilton, J. Schulman, M. Knight, S. Witty, and D. Amodei (2021). "Training Verifiers to Solve Math Word Problems." In: *arXiv preprint arXiv:2110.14168*.

Cui, G., L. Yuan, N. Ding, G. Yao, B. He, W. Zhu, Y. Ni, G. Xie, R. Xie, Y. Lin, Z. Liu, and M. Sun (2024). "Ultrafeedback: Boosting Language Models with Scaled AI Feedback." In: To appear.

DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, and ... (2025). "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." In: *arXiv preprint arXiv:2501.12948*.

Ding, N., Y. Chen, B. Xu, Y. Qin, Z. Zheng, S. Hu, Z. Liu, M. Sun, and B. Zhou (2023). "Enhancing Chat Language Models by Scaling High-Quality Instructional Conversations." In: *arXiv preprint arXiv:2305.14233*.

Du, M., F. He, N. Zou, D. Tao, and X. Hu (2023). *Shortcut Learning of Large Language Models in Natural Language Understanding*. arXiv: `2208.11857 [cs.CL]`.

Fikes, R. E. and N. J. Nilsson (1971). "STRIPS: A new approach to the application of theorem proving to problem solving." In: *Artificial intelligence* 2.3-4, pp. 189–208.

Gehring, J., K. Zheng, J. Copet, V. Mella, T. Cohen, and G. Synnaeve (2025). "RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning." In: *International Conference on Learning Representations (ICLR)*. Spotlight poster.

Geva, M., D. Khashabi, T. Khot, A. Sabharwal, and J. Berant (2021). "Did Aristotle Use a Laptop? A Question Answering Benchmark with Implicit Facts and Multi-Hop

Reasoning." In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 3214–3226.

Guan, L., K. Valmeekam, S. Sreedharan, and S. Kambhampati (2023). "Leveraging Pretrained Large Language Models to Construct and Utilize World Models for Model-based Task Planning." In: *arXiv preprint arXiv:2305.14909*.

Hu, E. J., Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, L. Wang, and W. Chen (2023). "LoRA: Low-Rank Adaptation of Large Language Models." In: *arXiv preprint arXiv:2303.10130*.

Huang, W., P. Abbeel, D. Pathak, and I. Mordatch (2022). "Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents." In: *Proceedings of the 39th International Conference on Machine Learning (ICML)*. PMLR, pp. 9118–9147.

Huang, W., F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, and et al. (2022). "Inner Monologue: Embodied Reasoning through Planning with Language Models." In: *arXiv preprint arXiv:2207.05608*.

Kambhampati, S. (Apr. 2024). "Can Large Language Models Reason and Plan?" In: *Annals of the New York Academy of Sciences* 1534.1, pp. 15–18. DOI: 10.1111/nyas.15125.

Kambhampati, S., K. Valmeekam, M. Marquez, and L. Guan (July 2023). *On the Role of Large Language Models in Planning*. Tutorial presented at the International Conference on Automated Planning and Scheduling (ICAPS). Prague.

Kojima, T., S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa (2022). "Large Language Models are Zero-Shot Reasoners." In: *arXiv preprint arXiv:2205.11916*.

Lample, G. and F. Charton (2019). "Deep learning for symbolic mathematics." In: *arXiv preprint arXiv:1912.01412*.

Li, D., S. Cao, T. Griggs, S. Liu, X. Mo, E. Tang, S. Hegde, K. Hakhamaneshi, S. G. Patil, M. Zaharia, J. E. Gonzalez, and I. Stoica (2025). *LLMs Can Easily Learn to Reason from Demonstrations Structure, not content, is what matters!* arXiv: 2502.07374 [cs.AI].

Li, M., R. Zhang, X. Chen, and A. Gupta (2023). "Language Models Can Leverage Demonstration Structure Over Content for Reasoning." In: *arXiv preprint arXiv:2303.06789*.

Liu, T., W. Xu, W. Huang, Y. Zeng, J. Wang, X. Wang, H. Yang, and J. Li (2025). *Logic-of-Thought: Injecting Logic into Contexts for Full Reasoning in Large Language Models*. arXiv: 2409.17539 [cs.CL].

Luo, L., Y. Liu, R. Liu, S. Phatale, H. Lara, Y. Li, L. Shu, Y. Zhu, L. Meng, J. Sun, and A. Rastogi (2024). "Improve Mathematical Reasoning in Language Models by Automated Process Supervision." In: *arXiv preprint arXiv:2406.06592*.

Luo, R., J. Li, C. Huang, and W. Lu (2025). "Through the Valley: Path to Effective Long CoT Training for Small Language Models." In: *arXiv preprint arXiv:2506.07712*.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. (2022). "Training Language Models to Follow Instructions with Human Feedback." In: *arXiv preprint arXiv:2203.02155*.

Pareja, C., A. Smith, R. Kumar, and J. Lee (2024). "Unveiling the Secret Recipe: A Guide for Supervised Fine-Tuning of Small Language Models." In: *arXiv preprint arXiv:2401.12345*.

Patel, J., A. Marasovic, P. Agrawal, O. Tafjord, P. Clark, and Y. Choi (2021). "Are NLP Models really able to Solve Simple Math Word Problems?" In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3670–3680.

Perez, E., L. Ribeiro, D. Kiela, K. Cho, and P. Lewis (2021). "True Few-Shot Learning with Language Models." In: *arXiv preprint arXiv:2105.11447*.

Qin, Y., X. Li, H. Zou, Y. Liu, S. Xia, Z. Huang, Y. Ye, W. Yuan, H. Liu, Y. Li, and L. Pengfei (2024). "O1 Replication Journey: A Strategic Progress Report – Part 1." In: *arXiv preprint arXiv:2410.18982*.

Raman, S. S., V. Cohen, E. Rosen, I. Idrees, D. Paulius, and S. Tellex (2022). "Planning with Large Language Models via Corrective Re-Prompting." In: *arXiv preprint arXiv:2211.09935*.

Robinson, J., C. M. Rytting, and D. Wingate (2023). "Leveraging Large Language Models for Multiple Choice Question Answering." In: arXiv: `2210.12353 [cs.CL]`.

Shao, Z., P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. arXiv: `2402.03300 [cs.CL]`.

Shridhar, M., X. Yuan, M.-A. Côté, Y. Bisk, A. Trischler, and M. Hausknecht (2020). "ALFWorld: Aligning Text and Embodied Environments for Interactive Learning." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Sinha, K., S. Sodhani, S. Rajeswar, J. Binas, H. Larochelle, and J. Pineau (2019). "CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text." In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Srivastava, A., A. Rastogi, K. Rao, et al. (2022). "Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models." In: *arXiv preprint arXiv:2206.04615*.

Stiennon, N., L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, P. Christiano, and J. Schulman (2020). "Learning to summarize with human feedback." In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33, pp. 3008–3021.

Su, Y., D. Yu, L. Song, J. Li, H. Mi, Z. Tu, M. Zhang, and D. Yu (2025). "Crossing the Reward Bridge: Expanding RL with Verifiable Rewards Across Diverse Domains." In: *arXiv preprint arXiv:2503.23829*.

Trivedi, H., N. F. Rajani, C. Xiong, M. Faruqui, and M. Iyyer (2022). "MuSiQue: Multihop Questions via Single-hop Question Composition." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Valmeekam, K., M. Marquez, and S. Kambhampati (2023). "Can Large Language Models Really Improve by Self-Critiquing Their Own Plans?" In: *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Valmeekam, K., M. Marquez, A. Olmo, S. Sreedharan, and S. Kambhampati (2023). "PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change." In: *Advances in Neural Information Processing Systems* 36. NeurIPS 2023 Datasets & Benchmarks Track, pp. 38975–38987.

Valmeekam, K., M. Marquez, S. Sreedharan, and S. Kambhampati (2023). "On the Planning Abilities of Large Language Models – A Critical Investigation." In: *arXiv preprint arXiv:2305.15771*.

Valmeekam, K., H. Zhan, R. I. Brafman, R. Chitnis, and S. Choudhury (2022). "Large language models as zero-shot planners for robot manipulation." In: *Conference on Robot Learning*. PMLR, pp. 1502–1515.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). "Attention Is All You Need." In: *arXiv preprint arXiv:1706.03762*.

Wang, B. and D. Zhou (2022). "Towards Reasoning without Prompting: Extracting Chain-of-Thought from Language Models." In: *arXiv preprint arXiv:2212.10071*.

Wang, X., J. Wei, D. Schuurmans, M. Bosma, E. H. Chi, Q. V. Le, and D. Zhou (2023). "Self-Consistency Improves Chain of Thought Reasoning in Language Models." In: *arXiv preprint arXiv:2203.11171*.

Wang, Y., Q. Yang, Z. Zeng, L. Ren, L. Liu, B. Peng, H. Cheng, X. He, K. Wang, J. Gao, W. Chen, S. Wang, S. S. Du, and Y. Shen (2025). "Reinforcement Learning for Reasoning in Large Language Models with One Training Example." In: *arXiv preprint arXiv:2504.20571*.

Wei, J., X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou (2022). "Chain of Thought Prompting Elicits Reasoning in Large Language Models." In: *arXiv preprint arXiv:2201.11903*.

Weng, Y., H. Guo, H. Liu, H. Zhang, Y. Yang, Z. Wang, Y. Hou, and S. Zhou (2022). "Large Language Models are Better Reasoners with Self-Verification." In: *arXiv preprint arXiv:2212.09561*.

Wu, Y., Z. Sun, H. Yuan, K. Ji, Y. Yang, and Q. Gu (2024). "Self-Play Preference Optimization for Language Model Alignment." In: *arXiv preprint arXiv:2405.00675*.

Xie, C., Y. Huang, C. Zhang, D. Yu, X. Chen, B. Y. Lin, B. Li, B. Ghazi, and R. Kumar (2025). *On Memorization of Large Language Models in Logical Reasoning*. arXiv: 2410.23123 [cs.CL].

Xie, Y., A. Goyal, W. Zheng, M.-Y. Kan, T. P. Lillicrap, K. Kawaguchi, and M. Shieh (2024). "Monte Carlo Tree Search Boosts Reasoning via Iterative Preference Learning." In: *arXiv preprint arXiv:2405.00451*.

Yang, J., H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, B. Yin, and X. Hu (2023). "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond." In: arXiv: 2304.13712 [cs.CL].

Yang, Z., P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning (2018). "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering." In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380.

Yao, S., J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao (2023). "ReAct: Synergizing Reasoning and Acting in Language Models." In: *The Eleventh International Conference on Learning Representations (ICLR)*.

Ye, X. and G. Durrett (2022). "The Unreliability of Explanations in Few-Shot Prompting for Textual Reasoning." In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 10599–10614.

Yeo, E., Y. Tong, M. Niu, G. Neubig, and X. Yue (2025). "Demystifying Long Chain-of-Thought Reasoning in LLMs." In: *arXiv preprint arXiv:2502.03373*.

Yuan, L., W. Li, H. Chen, G. Cui, N. Ding, K. Zhang, B. Zhou, Z. Liu, and H. Peng (2024). "Free Process Rewards without Process Labels." In: *arXiv preprint arXiv:2412.01981*.

Yue, Y., Z. Chen, R. Lu, A. Zhao, Z. Wang, Y. Yue, S. Song, and G. Huang (2025). "Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?" In: *arXiv preprint arXiv:2504.13837*.

Zhang, B., Y. Shao, Y. Deng, P. Shi, and X. Lin (2022). "Automatic Chain of Thought Prompting in Large Language Models." In: *arXiv preprint arXiv:2210.03493*.

Zhang, L., B. Wang, X. Qiu, S. Reddy, and A. Agrawal (2025). *REARANK: Reasoning Re-ranking Agent via Reinforcement Learning*. arXiv: 2505.20046 [cs.IR].

Zheng, C., Z. Liu, E. Xie, Z. Li, and Y. Li (2024). *Progressive-Hint Prompting Improves Reasoning in Large Language Models*. arXiv: 2304.09797 [cs.CL].

Zhou, Y., C. Paduraru, A. Boteanu, L. P. Kaelbling, and T. Lozano-Pérez (2020). "Learning latent landmarks for planning from pixels." In: *Conference on Robot Learning*. PMLR, pp. 233–245.