Generalized Task Learning for Robots:

Unifying Task Hierarchies through Contrastive Learning

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Ryan Alexander
1827474

Supervised by Prof. Richard Klein and Dr. Steven James

2025-06-05

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science

Abstract

This dissertation addresses the challenge of enabling robots to generalize across unseen household tasks by learning abstract task structures from demonstration data. We develop a three-stage pipeline that translates natural language instructions and demonstrations into hierarchical task representations using large language models, clustering, and parameterized generalization. Our approach is tested and evaluated on the ALFRED benchmark [Shridhar *et al.* 2020]. ALFRED acts as a standardized measure used for training models to comprehend and follow instructions in natural language. It leverages first-person perspective visual input to carry out a series of actions for various household tasks. While this approach doesn't represent the state-of-the-art, it establishes a foundation for future research to build upon.

Declaration

I, Ryan Alexander, hereby declare the contents of this research proposal to be my own work. This dissertation is submitted for the degree of Master of Science in Computer Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

06/06/2025

Acknowledgements

Thanks to my supervisors, Prof. Richard Klein and Dr. Steven James, for their help and insight each week.

Contents

Preface
Abstract i
Declaration ii
Acknowledgements iii
Table of Contents iv

1 Introduction **1**

2 Background **4**
2.1 ALFRED Benchmark 4
2.2 Hierarchical Task Networks 5
2.3 **H**ierarchical **D**omain **D**efinition **L**anguage 6
2.4 PANDA Framework 8
2.5 Large Language Models 9
2.6 Improving text embeddings with LLMs 13
2.7 LLM augmented clustering 14
2.7.1 Large Language Models Enable Few-Shot Clustering 15
2.8 Framer: Planning Models from Natural Language Action Descriptions . . 16
2.9 Conclusion 17

3 Related Work **19**
3.1 Embodied Instruction Following 19
3.2 Agent Generalization 21
3.3 Planning with Large Language Models 22
3.4 Conclusion 24

4 Methodology **25**
4.1 Problem Statement 26
4.2 Hypothesis 26
4.3 Research Questions 26
4.4 Generalized Method Creation 26
4.5 Natural Language to Formal Language 27
4.6 Method Construction 31
4.7 Clustering and Parametrising 33
4.7.1 Clustering 34
4.7.2 Generalized Method Creation 35

4.7.3 Conversion to HDDL . . . 38
4.8 Runtime Pipeline . . . 41
4.8.1 Vision Subtask . . . 43
4.8.2 Language Subtask . . . 44
4.8.3 Planning Subtask . . . 46
4.9 Conclusion . . . 50

5 Results **51**
5.1 Assessing clustering capability . . . 51
5.1.1 Evaluation criteria . . . 51
5.1.2 Comparison of Clustering Algorithms . . . 52
5.1.3 Impact of Data Representation . . . 52
5.1.4 Method Variations Analysis . . . 54
5.1.5 Data Augmentation Techniques . . . 54
5.2 Analyzing template consistency . . . 55
5.2.1 Evaluation criteria . . . 55
5.2.2 LLM Analysis . . . 56
5.3 Evaluating the accuracy of semantic mapping . . . 56
5.3.1 Evaluation criteria . . . 57
5.3.2 Analysis . . . 58
5.4 Verifying plan integrity . . . 58
5.4.1 Evaluation criteria . . . 59
5.4.2 Analysis . . . 59
5.5 ALFRED benchmark performance analysis . . . 60
5.5.1 Metrics . . . 60
5.5.2 Baselines . . . 60
5.5.3 Evaluations . . . 61
5.6 Conclusion . . . 61

6 Conclusion and Future Directions **63**
6.1 Problems with Natural Language Understanding . . . 63
6.2 Difficulty Measuring Correctness . . . 64
6.2.1 LLM templates . . . 64
6.2.2 Generated Plans . . . 65
6.3 PANDA scaling issues . . . 65
6.4 LLM consistency . . . 66
6.5 Problem generating entire HDDL domain files . . . 66
6.6 Contributions . . . 67

Appendices **78**

Chapter 1

Introduction

Robots in the modern world are used primarily in factories and other such industrial settings. These robots are hard-coded and unable to adapt to unseen tasks or environments. The ability of robots to adapt and be general agents is still a grand challenge in the world of robotics [Brohan *et al.* 2023]. There are many difficulties in creating a general-purpose robot: the world is stochastic and unpredictable, robots have to adapt to new environments [Gogoll *et al.* 2020], accomplish tasks they have never been trained to do [Agarwal *et al.* 2021], plan long-horizon tasks and reason about abstract goals [Shridhar *et al.* 2020]. Despite advances in embodied instruction following, existing systems struggle to generalize across task variations. This work addresses the problem of deriving reusable, abstract hierarchical plans from demonstrations.

Robots with natural language capabilities are likely to play an increasingly important role in daily life over the next decade. Using language as an interface to robots will allow more people to interact with them without requiring expert knowledge. One such avenue of use for these instruction-following robots will be helpers in the home [Brohan *et al.* 2023; Kidd and Breazeal 2008; Gates 2007]. These robots would need to be truly versatile, and capable of handling any task a user might request, even those they weren't specifically programmed to perform. What's more, they must navigate the incredible variability of home environments - not just the differences between a kitchen and a living room, but also the unique layouts across different homes, and the constant changes that occur within these spaces over time. This combination of unpredictable tasks and ever-changing, unstructured environments significantly amplifies the challenge of developing robots with genuinely adaptable capabilities.

Many tasks in the real world have an inherent hierarchical nature, for example, Figure 1.1 shows an example of a task from ALFRED with the breakdown of the smaller subtasks required to accomplish the goal. These task descriptions are accompanied by images showing full trajectories to teach an agent how to complete the task. The goal of "Rinse off a mug and place it in the coffee maker" can be decomposed into six different subtasks. These subtasks can be further decomposed, such as "walk to the coffee maker on the right" consisting of simple actions like stepping forward and turning right.

Figure 1.1: Demonstration of high-level goal task with accompanying step-by-step instructions from ALFRED [Shridhar *et al.* 2020].

A formalism convenient for this hierarchical representation is Hierarchical Task Networks (HTN) [Erol *et al.* 1994]. The hierarchical decomposition in HTNs provides a way to represent a task's structure and the associated knowledge and constraints. This provides a powerful representation that is human interpretable. The interpretability stems from the fact that HTNs are typically specified by hand by experts; this is useful for smaller constrained domains but cannot scale to autonomous or general agents.

Though traditional planners are very effective they can lack generalizability and world knowledge. Language models, however, contain enormous amounts of information about the world that can be helpful to agents [Huang *et al.* 2022]. By leveraging the language model's encoded world knowledge, the integration of text and speech enables seamless interaction between individuals and assistant robots. This integration not only allows for better human-to-robot communication but also enhances the robot's capabilities to tackle intricate and abstract tasks, thereby bolstering its overall performance [Brohan *et al.* 2023].

Learning how to accomplish tasks from scratch can be difficult and time-consuming. Learning from Demonstration (LfD) [Schaal 1996] allows the agent to learn by imitating experts. Though this gives us a quicker learning time, the learned tasks are grounded in their specific demonstration and do not generalize well. We seek a method to lift these representations into a more abstract representation, allowing us to generalize.

Previous methods address this challenge with varying success but have their downfalls. LANCON-LEARN [Silva *et al.* 2021] utilizes GloVe embeddings [Pennington *et al.* 2014] to encode goal sequences, but requires high similarity between language commands across training and transfer tasks. Suddrey *et al.* [2016] use a method for learning complex hierarchical tasks through natural language instructions using HTNs similar to

our approach, but relying on a manually-updated fixed lexicon. While other approaches to the ALFRED benchmark like Min *et al.* [2021] and Inoue and Ohashi [2022] achieve higher performance, they sacrifice generalizability by employing predetermined templated subtasks specifically designed for each of the ALFRED tasks.

To remedy these issues we present a novel framework for autonomously learning generalized HTNs. Our pipeline consists of three key modules: First we use an LLM to convert natural language instructions into formalized commands; We then build hierarchical methods derived from ALFRED's demonstrations; Finally we group similar methods, then combine and parameterize these groups to create more general, reusable methods. The work is not State-of-the-Art (SOTA) as alternative methods show better generalisation ability across the benchmark task, but it presents a basis on which future research can be established.

Our contributions are the following: A pipeline to autonomously extract generalized HTNs from demonstrations, an evaluation framework for clustering and plan accuracy, and insights into the limitations of semantic mapping and plan generalization.

In the subsequent chapters, this dissertation will unfold as follows: Chapter 2 provides essential background information forming the foundation of this work. Chapter 3 examines relevant literature and related research. Chapter 4 details our methodological approach. Chapter 5 presents our research findings. Chapter 6 concludes with a discussion of limitations, potential avenues for future research, and final reflections on our work.

Chapter 2

Background

This chapter provides foundational knowledge essential for understanding the presented work. It begins with introductory explanations of key concepts including ALFRED, Hierarchical Task Networks (HTNs), Hierarchical Domain Definition Language (HDDL), and the PANDA Framework. These sections establish the necessary background for comprehending the Methodology Chapter.

The chapter then breaks down the architecture and training methodologies of Large Language Models (LLMs) to establish a clear understanding of their function. Following this, it explores the application of LLMs for embedding and clustering techniques and concludes with an analysis of how LLMs have been previously deployed as evaluative tools. We will be using these techniques throughout the presented work.

2.1 ALFRED Benchmark

Numerous benchmarks that use simulated physical environments have been constructed to advance instruction following within domains such as navigation and embodied question-answering. The **A**ction **L**earning **F**rom **R**ealistic **E**nvironments and **D**irectives (ALFRED) [Shridhar *et al.* 2020] benchmark, located within the AI2-THOR [Kolve *et al.* 2017] simulator, involves the execution of complex household activities in realistic indoor settings using a mobile manipulator guided by natural language instructions.

AI2-THOR (**T**he **H**ouse **O**f inte**R**actions) offers immersive 3D indoor environments that closely resemble real-life visuals. Within these environments, AI agents can navigate and interact with objects, enabling them to accomplish various tasks.

Implemented on the AI2-THOR framework, the ALFRED benchmark facilitates the transfer from simulated environments to real-world applications. The tasks within ALFRED exhibit long horizons and compositionality, characterized by irreversible state changes, thereby rendering them considerably more complicated in terms of task length, action space, and language components compared to existing benchmarks.

ALFRED contains an extensive database of demonstrations by experts in interactive visual settings, featuring 25,000 instructions in natural language. These instructions

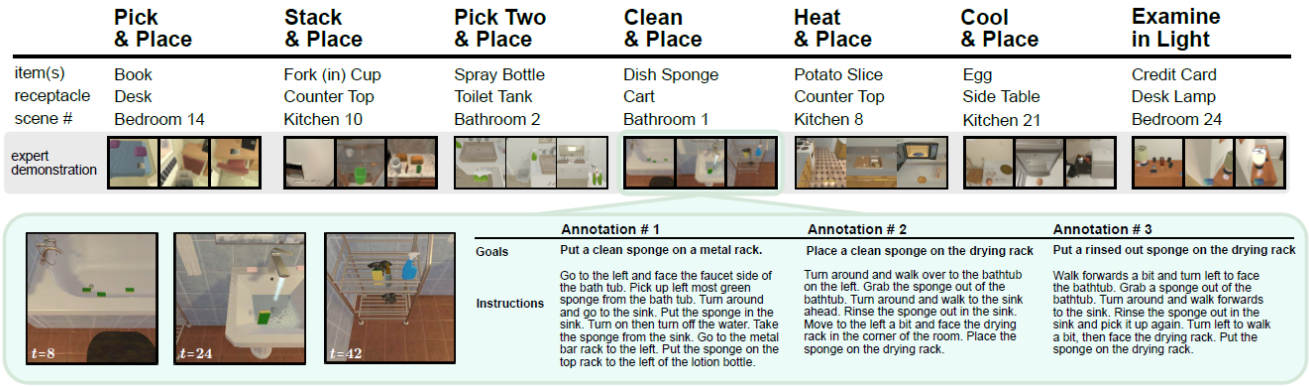

Figure 2.1: The benchmark consists of 120 scenes with a total of 7 distinct task types (Pick & Place, Stack & Place, Pick Two & Place, Clean & Place, Heat & Place, Cool & Place, Examine in Light) each parameterized by 84 object classes. All ALFRED tasks follow a consistent structure: they specify the item(s) requiring manipulation and a receptacle where the item must be placed. The figure further illustrates the instruction format through a Clean & Place task example. Every task contains a goal statement (a high-level instruction identifying both the primary item and its target receptacle) accompanied by a sequential list of completion instructions. Each task includes annotations from 3 different annotators [Shridhar *et al.* 2020].

include 7 distinct task types (explanation and examples can be seen in Figure 2.1). The data of the benchmark is split into official train, validation and test sets. The train set consists of 21 023 demonstrations in 108 scenes. The validation set consists of 820 seen demonstrations in 88 seen scenes and 821 unseen demonstrations in 4 new scenes. The test set consists of 1 533 seen demonstrations in 107 seen scenes with 1 529 unseen demonstrations in 4 new scenes.

As ALFRED tasks are decomposable into subtasks, we need an effective formalism to represent this structure. An effective framework for representing such a hierarchy is Hierarchical Task Networks (HTN) [Erol *et al.* 1994].

2.2 Hierarchical Task Networks

Hierarchical Task Networks (HTNs) [Erol *et al.* 1994] are a formalism used in artificial intelligence and automated planning to represent and reason about complex tasks and their subtasks. HTNs provide a hierarchical decomposition of tasks, allowing the representation of high-level goals and the decomposition of those goals into smaller, more manageable subtasks.

In an HTN, tasks are represented as a hierarchy of task networks (example seen in Figure 2.2). The hierarchy starts with a high-level goal, which is decomposed into subgoals. Each subgoal is further decomposed until a set of primitive tasks is reached. Primitive tasks are basic actions that can be executed directly. HTNs make use of task methods, a method is a 4-tuple; m = (name, task, precond, subtasks). Name: the

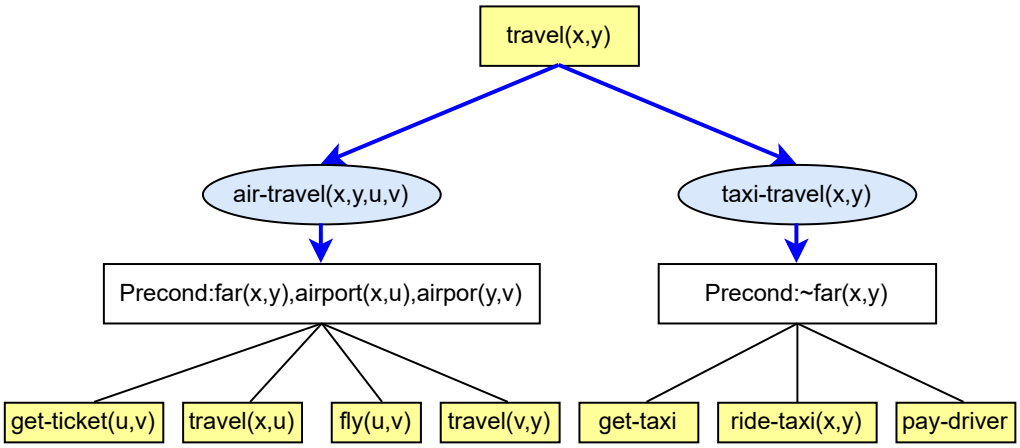

Figure 2.2: Example of a travel method. The method shows there are two ways to accomplish the task, either by air or by taxi. If you are far and there a two airports specified the preconditions for `air-travel` method. This method can be decomposed into four subtasks; `get-ticket, travel, fly & travel`. If you are not far then the `taxi-travel` preconditions are met. `taxi-travel` can be decomposed into `get-taxi, ride-taxi, pay-driver`. The variables `x,y` represent the start and end locations respectively while `u,v` represent the locations of the airports.

method names followed by parameters. Task: the nonprimitive task that the methods perform. precond: a list of states that must be true for the method to be applicable. Subtasks: a sequence of tasks the method accomplishes. Methods can be defined at different levels of abstraction, allowing for varying degrees of decomposition. At higher levels, methods can represent abstract strategies or plans, while at lower levels, they can represent more complex and specific sequences of actions.

HTN methods have been widely used in AI planning systems [Nau *et al.* 2003], intelligent agents [Hayashi *et al.* 2005], robotics [Lallement *et al.* 2014], game AI [Kelly *et al.* 2008], and other domains where complex tasks need to be decomposed and planned systematically. HTN planners manage this decomposition and planning by utilizing solvers that apply methods such as plan space search [Bercher *et al.* 2014a 2017], progression search [Höller *et al.* 2018 2019], and conversion to propositional logic [Behnke *et al.* 2018ab] to address planning problems.

2.3 Hierarchical Domain Definition Language

Hierarchical planning requires expressive domain modelling languages to capture task decomposition, ordering constraints, and other hierarchical relationships. While **P**lanning **D**omain **D**efinition **L**anguage (PDDL) [McDermott *et al.* 1998] has served as the de facto standard for classical planning, hierarchical planning has lacked a unified input language. This fragmentation has impeded system comparisons, benchmarking, and practical adoption of hierarchical planning approaches.

Hierarchical **D**omain **D**efinition **L**anguage (HDDL) [Höller *et al.* 2020a] addresses this gap by extending PDDL's formalism. HDDL introduces syntax for abstract tasks, decomposition methods, and task networks while maintaining compatibility with PDDL constructs. The language requires explicit definitions of abstract tasks and supports both totally and partially ordered task networks. HDDL includes method preconditions and variable constraints while remaining extensible for future features.

HDDL's adoption as the standard input language for the International Planning Competition's hierarchical track in 2020 marked a significant step toward standardization. Multiple planning systems now support HDDL across different algorithmic approaches, including Plan Space Search methods [Bercher *et al.* 2017], State Space Search methods [Höller *et al.* 2020b] and SAT translations [Behnke *et al.* 2018a]. This standardization facilitates systematic comparison of hierarchical planners and provides a foundation for evaluating new algorithmic advances in the field.

HDDL uses specific syntax to define hierarchical planning elements. Keywords begin with a colon (:), while variables start with a question mark (?). Variable types appear after the variable name.

For example, HDDL indicates:

- Keywords with a colon prefix (e.g., `:method`, `:parameters`, `:types`)
- Variables with a question mark prefix (e.g., `?p - package`, `?lp - location`)
- Task identifiers/names without special prefixes (e.g. `drive`, `CleanObject`)

Here are samples of HDDL task and method definitions:

```
(:task deliver :parameters (?p - package ?l - location))

(:method m-deliver
  :parameters (?p - package ?lp ?ld - location)
  :task (deliver ?p ?ld)
  :ordered-subtasks (and
    (get-to ?lp)
    (pick-up ?lp ?p)
    (get-to ?ld)
    (drop ?ld ?p)))
```

This example defines the task `deliver` which has two parameters `?p` & `?l` of type `package` and `location` respectively. The task is accomplished by the method `m-deliver`. This method takes in the variables; `?p` of type `package` and `?lp` (location of package) & `?ld` (delivery location) of type `location` with `?p` and `?ld` corresponding to the parameters of the task `deliver`. The method decomposes the task into four ordered sub-tasks: getting to the package at location `?lp`, picking it up, getting to its destination (`?ld`), and then finally dropping the package.

2.4 PANDA Framework

The **P**lanning and **A**cting in a **N**etwork **D**ecomposition **A**rchitecture (PANDA) framework, introduced by Höller *et al.* [2021] is a versatile planning framework that incorporates various functionalities such as Problem Definition, Preprocessing, Automated Planning, Goal Recognition and more. The most important of these functionalities are the Problem Definition and Preprocessing functionalities as they provide a common infrastructure for all other functions to use as the basis.

The Problem Definition enables a common language across components. HDDL [Höller *et al.* 2020a] was created as this standard input language. A standardized problem description language aids developers in benchmarking solvers and allows users to model their problems before selecting the best planning system, avoiding initial system constraints.

The Preprocessing component comprises grounding and reachability analysis, building upon the foundations established by Behnke *et al.* [2020]. During the grounding phase, variables within the definition are substituted with all conceivable constants. For example, if you have an action "move(x, y)" where x and y are location variables, grounding would replace these with all possible location constants. Ideally, the grounding would contain precisely those instantiations that may appear in at least one solution; however, deriving this exact set is impractical. Therefore, an over-approximation is computed. In classical planning, techniques for state-based reachability analysis are available and can be adapted for use in the hierarchical context. Hierarchical reachability serves to further minimize the grounding, as only those elements that are accessible via state and hierarchy need to be incorporated into the grounded model.

The automated planning component consists of three distinct solvers; Heuristic Search in Plan Space [Bercher *et al.* 2017], Heuristic Progression Search [Höller *et al.* 2020b] and Translation to Propositional Logic [Behnke *et al.* 2018a].

Plan space search uses a partial order among tasks at nodes for a compact search space representation, eliminating the need to depict every task sequence. However, this approach lacks detailed insight into the "current state" as action sequences are unspecified, complicating heuristic development. The PANDA plan space search algorithm is detailed by Bercher *et al.* [2014ab].

In Progression Search, the planner builds the plan forward, managing only tasks without predecessors in the task network. This ensures a fully ordered initial plan segment and defines the current state, enabling heuristic calculations. This method is based on classical planning techniques [Höller *et al.* 2018 2019].

Although HTN planning is in general undecidable [Alford *et al.* 2015], PANDA limits the decomposition depth of potential solutions, to make the HTN problem decidable. If the generated formula is satisfiable, a viable plan is returned; otherwise, the bound is raised. Due to HTN planning's undecidability, maintaining completeness while stopping is generally not feasible. Encodings exist for fully ordered tasks [Behnke *et al.* 2018a], different problem types [Behnke *et al.* 2018b], and minimal actions [Behnke *et al.* 2019].

Though we have methods to solve HTNs, the problem is that these HTNs are hand-crafted and therefore grounded to their specific domains, so they cannot scale to autonomous agents as they lack knowledge of the world. We, therefore, need to look at ways to construct lifted methods that are abstracted and have tacit knowledge. To craft our HTNs autonomously, we look toward Large Language Models.

2.5 Large Language Models

Since the introduction of the transformer architecture [Vaswani *et al.* 2017], the state-of-the-art in Natural Language Processing (NLP) has been rapidly improving.

The transformer architecture introduced a new encoder-decoder paradigm that operates solely using self-attention. Self-attention enables token interactions within the model. Tokens examine others using attention, gather context, and update their own representation. Formally, attention is implemented using query-key-value attention. Let's denote:

- Q as the query vector (representing what we're searching for),
- K as the key vector (representing what we're searching in),
- V as the value vector (representing what we retrieve).

The attention by Vaswani *et al.* [2017] is specifically "Scaled Dot-Product Attention", The input consists of vectors Q and K of dimension d_k, and V of dimension d_v. It is calculated as follows;

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V. \tag{2.1}$$

Instead of computing attention once with original dimensionality, the multi-head attention mechanism projects queries, keys, and values h different times through separate learned linear transformations. Each projection changes the dimensionality to d_k, d_k and d_v respectively. The attention function is then applied in parallel across all these projections, producing d_v-dimensional outputs for each head. These separate attention outputs are concatenated together and projected once more through another linear transformation to produce the final result as in figure 2.4.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(head_1, ..., head_h)W^O \\ \text{where} \quad head_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \end{aligned} \tag{2.2}$$

where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Multi-head attention enhances transformer models by allowing them to focus on different aspects of input simultaneously. By splitting attention into several parallel heads, the model gains the ability to capture diverse relationships (syntactic, semantic, positional) within the data at once, rather than averaging these signals together. Each head projects the input into a different subspace, learning to attend to distinct patterns,

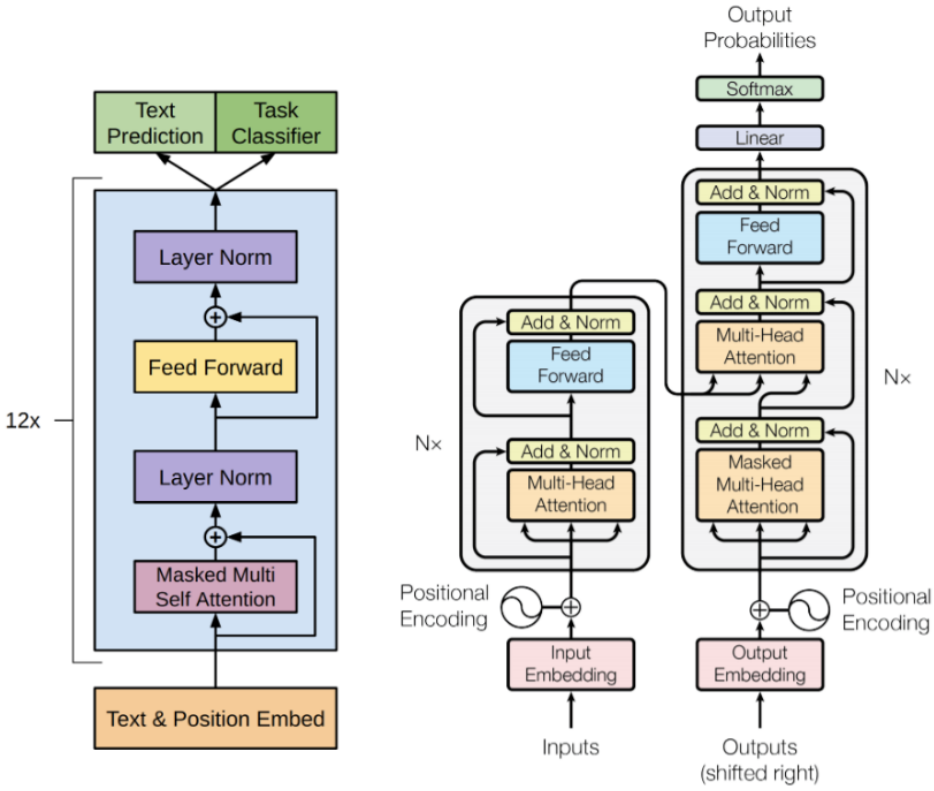

Figure 2.3: The key difference between decoder-only and encoder-decoder transformer models lies in their architecture and information flow. Decoder-only models (left image) use a single stack of layers with masked self-attention that prevents looking ahead, making them suitable for text generation tasks where tokens are produced sequentially based on previous outputs. In contrast, encoder-decoder models (right image) [Vaswani *et al.* 2017] utilize two separate components: an encoder that processes the entire input sequence with bidirectional attention (accessing all context in both directions), and a decoder that generates outputs while attending both to previously generated tokens and the encoder's representations. This dual structure makes encoder-decoder models particularly effective for sequence-to-sequence tasks like translation or summarization where input and output can have different lengths or structures, while decoder-only models excel at continuation-based generation.

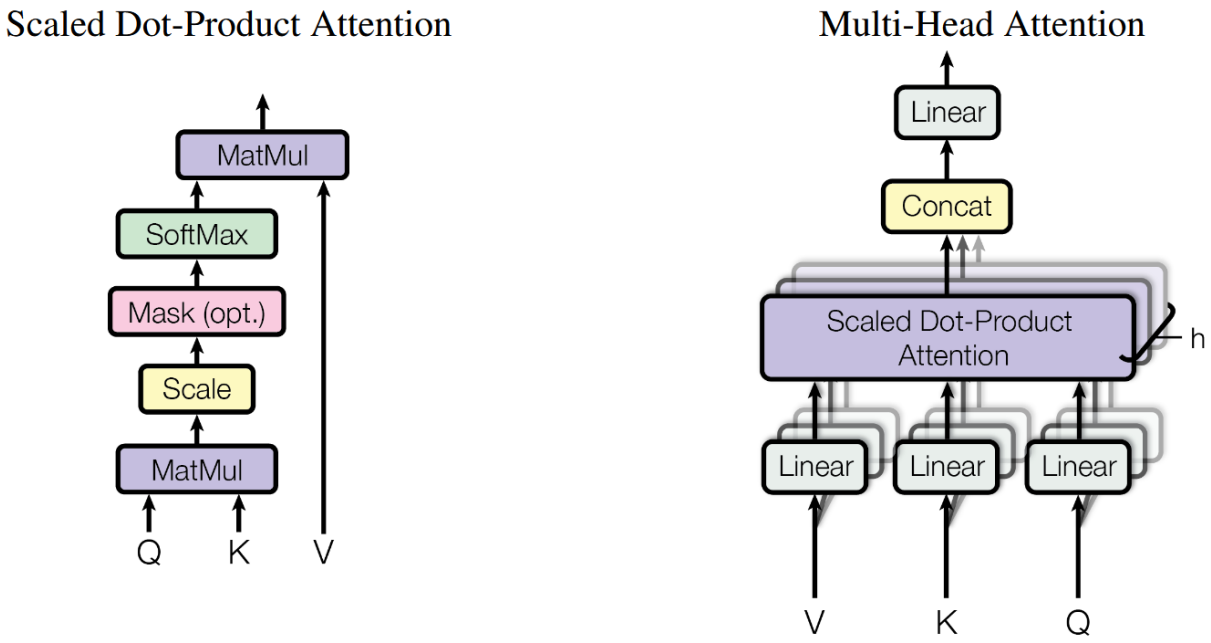

Figure 2.4: (left) Scaled Dot-Product Attention. (right) Multi-head attention consists of several attention layers running in parallel. [Vaswani *et al.* 2017]

which collectively creates a richer representation than a single attention mechanism could achieve. This parallel processing not only improves the model's capacity to learn complex dependencies but also provides a form of ensemble effect that increases stability during training and enhances generalization to new data.

Each layer in both the encoder and decoder contains a feed-forward neural network component in addition to the attention mechanisms (can be seen in figure 2.3). This feed-forward network processes each position in the sequence independently and in the same way. It has a simple structure consisting of two linear transformations with a ReLU activation function between them.

$$\text{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{2.3}$$

This simple but effective architecture has spawned Large Language Models (LLMs) such as BERT [Devlin *et al.* 2019], LaMDA [Thoppilan *et al.* 2022], PaLM [Chowdhery *et al.* 2023] and GPT-3 [Brown *et al.* 2020]. All use decoder-only transformer architectures.

LLMs from as early as BERT [Devlin *et al.* 2019] to the most recent models like the Llama-3 family [Dubey *et al.* 2024] share a similar pre-training. The models learn without explicit labels using a technique called "masked language modelling" or "next token prediction". A sequence of text is presented with certain tokens concealed or masked, requiring the prediction of these missing tokens using the surrounding context. This forces the model to develop an understanding of grammar, semantics, factual knowledge, and reasoning.

After this pre-training step, certain models (such as ChatGPT [Achiam *et al.* 2023] & Llama Instruct variants [Dubey *et al.* 2024]) undergo a further fine-tuning process.

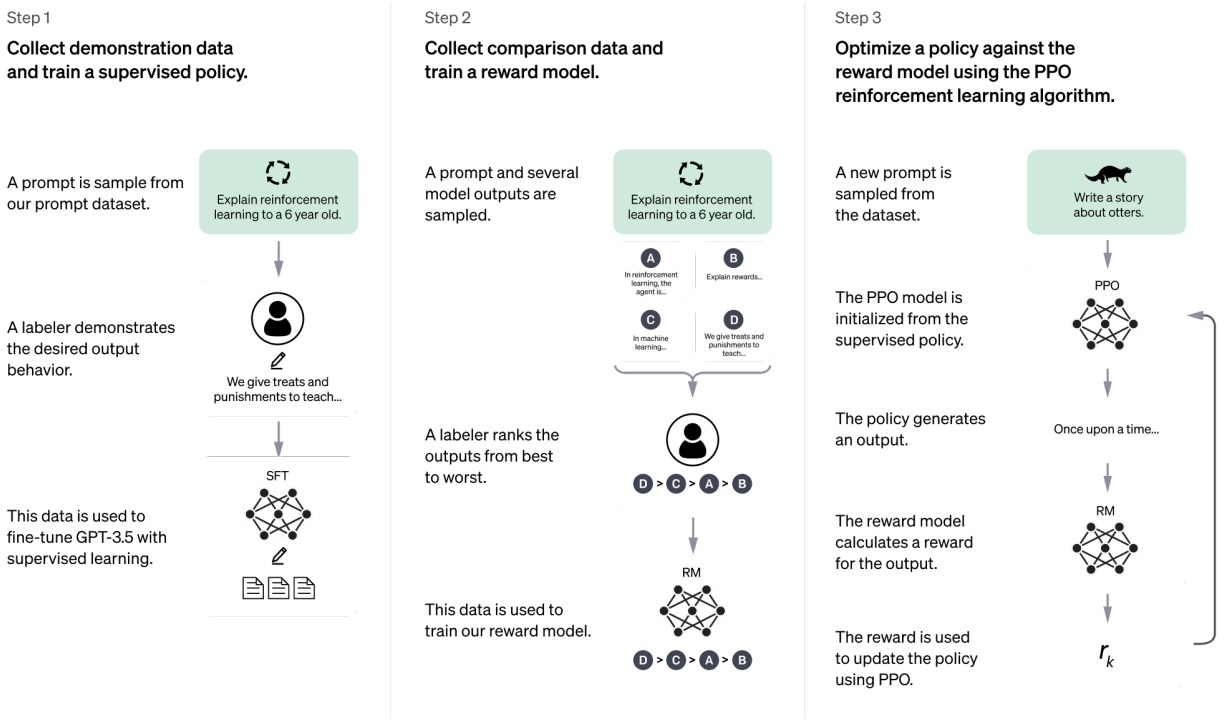

Figure 2.5: This Figure serves as an example of one possible fine-tuning process, specifically the process used by InsturctGPT [Ouyang *et al.* 2022]. The Figure depicts the three-stage methodology: (1) supervised fine-tuning (SFT), (2) reward model (RM) training, and (3) reinforcement learning using proximal policy optimization (PPO) with the reward model. The arrows indicate data flows used for model training. In the second stage, boxes labelled A through D represent samples generated by the models that human labellers rank in order of quality. While this provides background context, the key findings discussed in subsequent Sections relate to using the fully trained models rather than the training process.

Modifying language models by finetuning them on a range of datasets framed as instructions demonstrates enhanced model performance and better generalization to unfamiliar tasks [Chung *et al.* 2024]. These processes can range from supervised fine-tuning [Chung *et al.* 2024] to more multi-step processes such as the one demonstrated in Figure 2.5.

The text generation capability is the primary focus of LLMs, but another powerful aspect obtained from LLMs is text representation. This representation comes in the form of text embeddings, which are numerical representations that encapsulate the meaning and semantic essence of a given text. They act as a set of numbers that encode the underlying concepts conveyed by the text. Text embeddings provide the means to transform unstructured textual data into a structured format. By utilizing embeddings, you gain the capability to compare various forms of text, ranging from individual words [Devlin *et al.* 2019] to entire documents [Carvallo *et al.* 2020], including sentences [Li *et al.* 2020] and paragraphs. Though these embeddings are powerful, it has been shown that they are anisotropic [Ethayarajh 2019], meaning they tend to cluster in certain regions or directions, creating an uneven distribution. Works such as Gao *et al.*

[2021] and Fang *et al.* [2020] have attempted to remedy this by combining the word embeddings of LLMs with contrastive learning techniques to create superior embeddings. Also, work by Wang *et al.* [2023] and Harris *et al.* [2024] focused on using LLMs to enhance embeddings instead of using the LLM embeddings themselves. Our requirement for robust embeddings stems from our approach to HTN method processing, as we represent these methods textually and need semantically rich embeddings to effectively distinguish between similar methods.

2.6 Improving text embeddings with LLMs

Text embeddings have seen major progress thanks to the incorporation of LLMs. Several key strategies have emerged using LLMs to enhance existing embedding methods, directly employing LLMs to create embeddings, and applying LLMs to better understand embedding behaviour [Nie *et al.* 2024]. For instance, researchers have found that preprocessing text with models like ChatGPT [Achiam *et al.* 2023] before embedding can improve results for specific datasets [Harris *et al.* 2024]. Another successful approach involves using LLMs to create synthetic training data spanning multiple languages and tasks, which has proven effective even without labelled data [Wang *et al.* 2023]. These advances have substantially improved universal text embeddings, which play a vital role in applications like retrieval-augmented systems [Cao 2024]. Despite this progress, researchers still face difficulties in creating embeddings that work well across different tasks and domains, indicating that more work lies ahead in this dynamic study area. We will take a closer look at the work presented by [Wang *et al.* 2023] as it is the method of embedding we will use in our work.

They introduce an efficient method for generating high-quality text embeddings using artificia data and minimal training. In contrast, traditional approaches often involve extensive pre-training on billions of text pairs, followed by fine-tuning with labelled datasets. In contrast, the approach by Wang *et al.* [2023] eliminates the need for complex training pipelines and manually collected data, which often have limited task variety and language coverage.

The method employs LLMs to generate diverse synthetic training data, spanning hundreds of thousands of embedding tasks across 93 languages. Researchers then train open-source, decoder-only language models on this data using contrastive learning. Testing shows that this approach performs strongly on major text embedding benchmarks without requiring any labelled data. When they combined both synthetic and labelled data for fine-tuning, their model achieved record-breaking results on the BEIR [Thakur *et al.* 2021] and MTEB [Muennighoff *et al.* 2023]benchmarks.

The process involves query-document pairs in retrieval tasks. For each pair (q^+, d^+), where q^+ is a random user search query and d^+ is its relevant document, a template is applied to the original query q^+ into q^+_{inst}:

$$q^+_{inst} = \begin{array}{r} \text{Instruct} : \{\text{task\_definition}\} \\ \text{Query} : \{q^+\} \end{array} \tag{2.4}$$

Where $\{task\_definition\}$ is the temporary placeholder for a concise one-sentence summary of the embedding task. The document side is not prefixed with any instruction, so the index can be prebuilt, meaning only the task needs to be customized based on the query.

To generate embeddings, a [EOS] token is appended to both the query and document. These augmented inputs are then processed through the pre-trained LLM, with their respective embeddings ($h_{q^+_{inst}}, h_{d^+}$) obtained by extracting the vector corresponding to the [EOS] token from the model's final layer. Training is performed using the standard InfoNCE loss function [Oord *et al.* 2018]:

$$\min L = -log\frac{\phi(q^+_{inst}, d^+)}{\phi(q^+_{inst}, d^+) + \sum\limits_{n_i \in N}(\phi(q^+_{inst}, n_i)} \tag{2.5}$$

In this equation, N represents the collection of all negative examples, while $\phi(q, d)$ denotes a scoring function that calculates the similarity between a query q and document d:

$$\phi(q, d) = \exp(\frac{1}{\tau}\cos(h_q, h_d)) \tag{2.6}$$

where τ is the temperature hyper-parameter. Which is fixed to 0.02.

Beyond enhancing embedding quality, LLMs have demonstrated their versatility across various research domains, and they've shown particular promise in advancing clustering algorithms. This advancement is crucial because even high-quality embeddings can be challenging to cluster effectively when dealing with highly similar sentences, making the development of robust clustering methods essential.

2.7 LLM augmented clustering

In our work, we employ clustering techniques to identify and group similar methods together, enabling us to combine and generalize these related tasks. Here we give insight on how LLMs can be used to enhance clustering performance.

ClusterLLM leverages LLM feedback to improve clustering quality and understand user preferences through carefully designed prompts [Zhang *et al.* 2023]. Similarly, the work by Viswanathan *et al.* [2024] demonstrates that LLMs can enhance few-shot semi-supervised text clustering by improving input features and providing constraints to the clusterer. Petukhova *et al.* [2025] investigate the impact of LLM embeddings on text clustering, finding that they excel at capturing nuances in structured language. However, increasing embedding dimensionality and summarization techniques do not consistently improve clustering efficiency. CACTUS, proposed by Tipirneni *et al.* [2024], introduces a context-aware clustering approach using open-source LLMs, featuring a novel augmented triplet loss function and a self-supervised clustering task. This method outperforms existing unsupervised and supervised baselines on various e-commerce datasets, demonstrating the potential of LLMs in improving text clustering techniques.

In the following section, we explore in detail the main method we will be using in this work, Large Language Models Enable Few-Shot Clustering [Viswanathan *et al.* 2024].

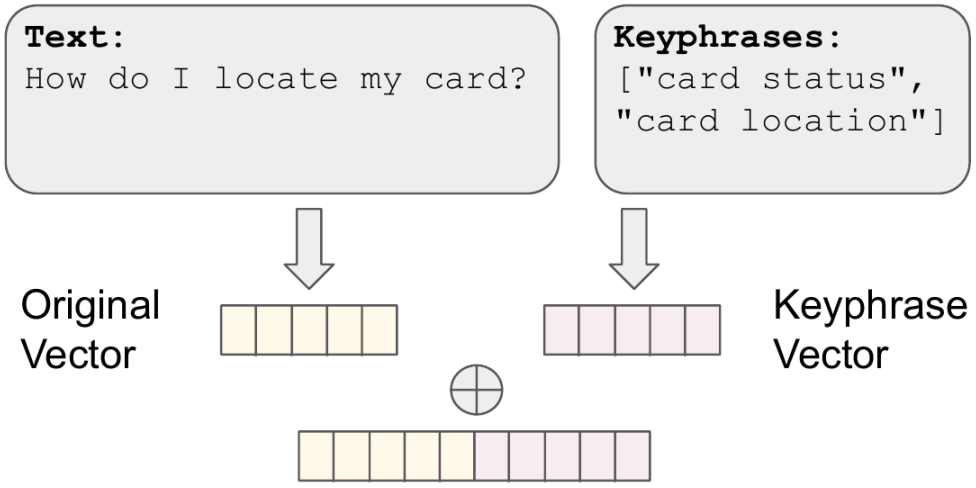

Figure 2.6: Enhancing document representations by appending keyphrase embeddings generated by a large language model.[Viswanathan *et al.* 2024]

2.7.1 Large Language Models Enable Few-Shot Clustering

Language models are transforming how we approach semi-supervised clustering, particularly for text data. While traditional clustering methods work independently, semi-supervised approaches incorporate user guidance to shape how data is grouped [Bae *et al.* 2020]. The challenge has been that this typically demands significant expert input. However, recent research by Viswanathan *et al.* [2024] demonstrates that large language models (LLMs) can enhance clustering performance even with limited supervision.

Their study explores three key integration points for LLMs: preprocessing (feature enhancement), during the clustering process (constraint generation), and post-processing (cluster refinement). The findings indicate that early LLM involvement yields the best results, while also helping optimize the tradeoff between computational cost and accuracy. Building on this, We specifically concentrate on using LLMs for keyphrase expansion during clustering. This approach recognizes that domain experts often have prior knowledge about important document characteristics. Rather than letting clustering algorithms work from scratch, we can use this expertise as a starting point. The process involves using an LLM to modify document content to better reflect clustering objectives by adding relevant context. These LLM-generated keyphrases are then encoded and integrated with the original document representations.

To generate keyphrases, Viswanathan *et al.* [2024] utilize GPT-3.5-turbo-0301 with a carefully crafted prompt structure. The prompt begins with a specific task instruction, for example: "I am trying to cluster online banking queries based on whether they express the same intent. For each query, generate a comprehensive set of keyphrases that could describe its intent, as a JSON-formatted list." They then provide example keyphrases for reference and context (these examples are illustrated in Figure 2.6). Once generated, these keyphrases undergo vector encoding and are combined with the original text's vector representation. To ensure a fair evaluation of the LLM's contri-

bution versus the encoding process itself, they employ the same encoder for both the keyphrases and the source text. This consistent encoding approach allows them to isolate the impact of the LLM-generated keyphrases from any potential benefits derived from the encoding mechanism. This straightforward approach demands minimal overhead and supervision, which are the primary reasons we selected it for our clustering needs.

2.8 Framer: Planning Models from Natural Language Action Descriptions

Lindsay *et al.* [2017] present an approach for learning domain models from natural language instructions (an overview can be seen in Figure 2.7). The first step in their pipeline is to generate action templates, which capture the main action, objects and their roles in the sentence. These templates are generated using CoreNLP graph dependencies, from CoreNLP annotation, verb, subject and object of the sentence form the action name and arguments.

The subsequent phase involves determining an optimal grouping of the simplified sentences that most accurately represent them, to develop a set of consistent and generalized operator descriptions. This is accomplished by establishing a distance metric between sentence pairs and applying it to identify distinct clusters. Lindsay *et al.* [2017] describe a similarity function to measure the distance between terms:

$$\texttt{SIM}(w_0, w_1, \text{POS}) = \frac{1}{n} \sum_{i=1}^{n} S_i(w_0, w_1, \text{POS}) \tag{2.7}$$

which estimates the similarity between words w_0 and w_1, and S_i is a source of synonyms from online lexical resources which can be parameterized by the Parts of Speech (POS) of the generated synonyms.

To calculate the similarity measure between temples ((τ_0, τ_1)) they define:

$$\delta(\tau_0, \tau_1) = \gamma \times \texttt{SAN}(\tau_0, \tau_1) + (1 - \gamma) \times \texttt{SSL}(\tau_0, \tau_1) \tag{2.8}$$

where $\texttt{SAN}(\tau_0, \tau_1)$ is the similarity between the action names and $\texttt{SSL}(\tau_0, \tau_1)$ is the average similarity for each non-action word. These values depend on the previously defined similarity measure $\texttt{SIM}(w_0, w_1, \text{POS})$, which is applied to action names using the parameter POS=verb and to roles using the parameter POS=* (representing any part of speech). γ is used to control the importance of the similarity between $\texttt{SAN}(\tau_0, \tau_1)$ and $\texttt{SSL}(\tau_0, \tau_1)$

The next step involves clustering similar templates. This process relies on a distance matrix to measure the distance between pairs of templates, which eliminates the need to define a projection of the templates into a specific space. The method they employ is the Partitioning Around Medoids (PAM) implementation of k-medoids. This technique divides the data into k clusters, each representing a central data point that is considered the most representative of the cluster. The main advantage of this approach is that it

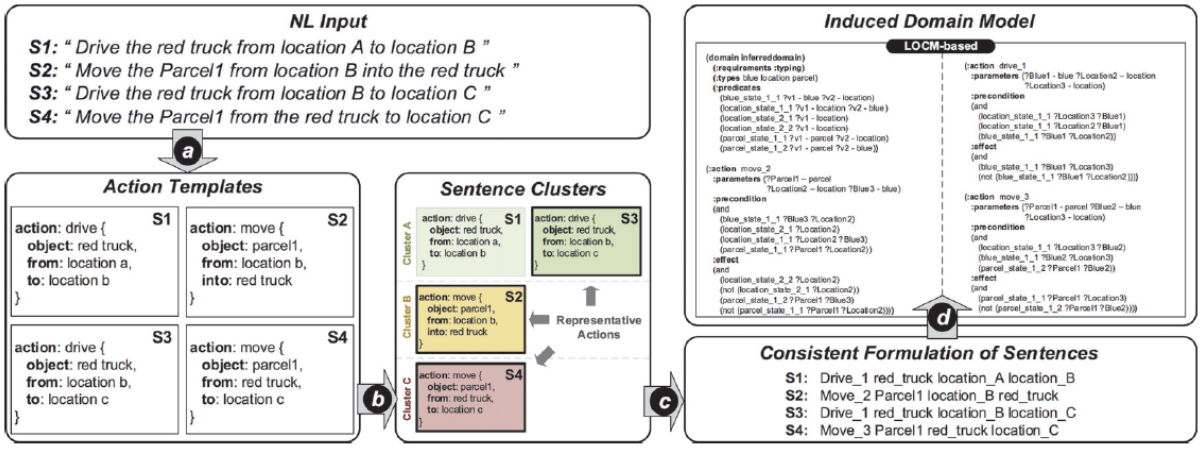

Figure 2.7: Natural language sentences are converted into simplified action templates (a) that undergo similarity-based clustering (b). From these clusters, consistent formulations of the original sentences are extracted (c), which are then used to generate a PDDL domain model through the LOCM domain model acquisition tool (d).[Lindsay *et al.* 2017]. This is the basis of our Natural Language to Formal Language (Section 4.5) pipeline.

allows for clustering based on a dissimilarity matrix of the data points, enabling the use of any arbitrary distance measure.

To create a consistent sentence formulation, the medoid of each cluster is selected as the foundation. The name is formed by concatenating (with a "_") all the terms in the action slot. The rest of the sentence translation involves joining the name with each word from the other template slots, separated by spaces. For translating other templates τ', a mapping is established between the slots of the medoid template τ^M and τ'. This is achieved by finding the best match between the slot labels of the templates, maximizing the overall similarity of the labels using the function: $\texttt{SIM}(\text{slot-label}_{\tau^M}, \text{slot-label}_{\tau'}, *)$.

The output of this process is sequences of action headers. Research by Cresswell and Gregory [2011] (LOCM) demonstrates that, under specific constraints, these action header sequences provide adequate evidence of planning domains' dynamic structure and can be directly utilized for domain model induction. LOCM's primary function is then to detect crucial relationships between action parameters, which it subsequently encodes as predicates within the induced domain model.

2.9 Conclusion

In this chapter, we present explanations of essential concepts necessary for comprehending our research. We detail the ALFRED benchmark, clarifying the size of the dataset and how the demonstrations are structured. We provide a formal definition and explanation of HTNs, showing why they are a useful formalism for our problem. A breakdown with examples of HDDL is given, to give context as to why it is the ap-

propriate language for the problem. The PANDA planner is then examined, giving an overview of how the framework finds a plan to solve HTNs.

We concluded by exploring the fundamentals of LLM architecture and training methodologies. We elaborated on techniques that we will be using for leveraging LLMs to enhance text embeddings and clustering.

Chapter 3

Related Work

In this chapter, we review related work addressing comparable challenges. We start by examining Embodied Instruction Following, a research area focused on enabling machines to understand and execute human instructions for physical tasks (demonstrated by benchmarks like ALFRED). We then explore the literature on agent generalization and the diverse approaches to developing generalist agents. Finally, we discuss recent advances in utilizing Large Language Models for planning-related tasks.

3.1 Embodied Instruction Following

Embodied Instruction Following (EIF) refers to a field of research and development that focuses on creating agents or robots capable of understanding and executing instructions given by humans through demonstrations or verbal communication. The goal is to enable machines to perceive and interpret human instructions and perform tasks accordingly.

Traditional instruction following methods often rely on explicit programming or rule-based approaches, where specific instructions are predefined and programmed into the system. However, Embodied Instruction Following takes a different approach by emphasizing the robot's ability to understand and interpret instructions in a more dynamic and context-dependent manner [Zhang *et al.* 2022; He *et al.* 2015; Brohan *et al.* 2023; Zellers *et al.* 2021].

There are many benchmarks for EIF such as TEACh [Padmakumar *et al.* 2022], Habitat [Savva *et al.* 2019] and ALFRED [Shridhar *et al.* 2020]. FILM [Min *et al.* 2021] proposed to tackle the problem using modular methods (splitting into language, vision and planning subtasks), which vastly outperformed the end-to-end approaches [Suglia *et al.* 2021; Zhang and Chai 2021; Pashevich *et al.* 2021] previously attempted. Min *et al.* [2021] propose an architecture that integrates four key modular components working in concert: Language Processing transforms instructions into structured formats, Semantic Mapping converts first-person visual inputs into semantic metric maps, the Semantic Search Policy predicts target locations and the Deterministic Policy generates navigation and interaction commands. Through its modular design incorporating

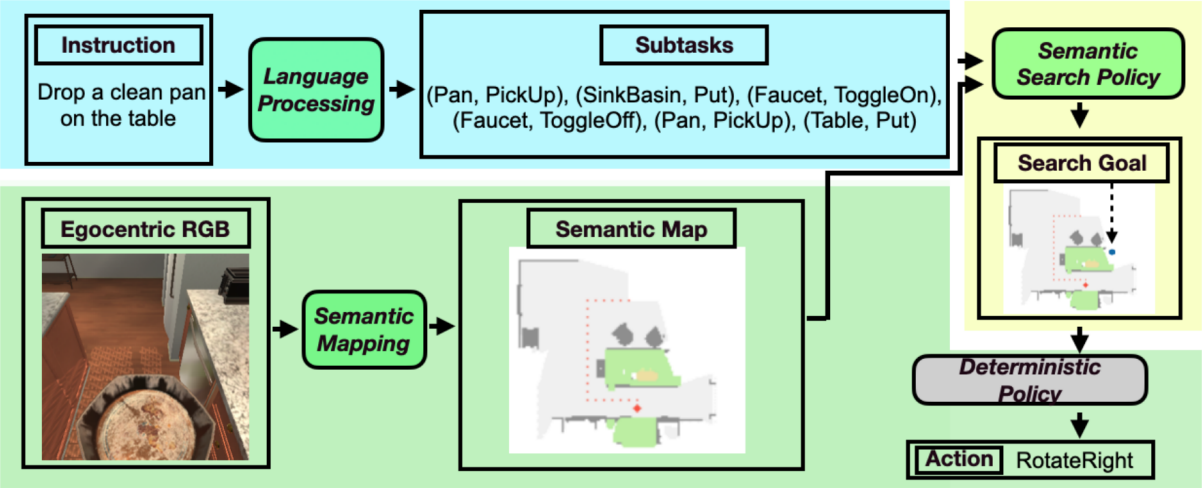

Figure 3.1: The Figure gives an overview of the FILM method, in which our runtime pipeline is built. The colour coding (blue, green, and yellow) represents different temporal granularities. Blue components operate at episode start, green components function For every interval, and yellow components work on a 25-step interval. When an episode begins, the Language Processing module converts instructions into subtasks. During each interval, Semantic Mapping transforms egocentric RGB into a bird's-eye view semantic map. The semantic search policy determines xxploration targets at the coarser interval. The Deterministic Policy then selects the subsequent action. Bright green modules are learned components, while the deterministic policy (shown in grey) is not learned.[Min *et al.* 2021]

structured spatial elements, FILM addresses limitations found in previous approaches. A significant advantage of FILM over existing Embodied Instruction Following methods is its independence from sequential guidance inputs such as expert trajectories or detailed low-level language instructions. On the ALFRED benchmark FILM achieves a 24.46% success rate.

The modular design of FILM has influenced many SOTA approaches on the ALFRED benchmark. Inoue and Ohashi [2022] developed one such approach that aims to reduce training data requirements. Their first contribution, FILM++, enhances the original FILM framework with modifications that require no additional data. While maintaining the same data-driven modules, FILM++ achieves more than twice the performance of the original FILM. They further developed Prompter, which substitutes FILM++'s semantic search component with language model prompting. Since language models benefit from extensive and diverse pretraining data, Prompter's semantic search capability eliminates the need for specific data collection and demonstrates greater robustness against distribution overfitting. Prompter achieves a 42.64% success rate on the ALFRED benchmark.

Murray and Cakmak [2022] developed an approach that leverages visible environmental landmarks to enhance exploration efficiency when searching for objects specified in natural language instructions. Their method also incorporates a pose adjustment policy during manipulation planning to compensate for visual observation noise. When

evaluated on the ALFRED benchmark, this approach demonstrates a 35.14% success rate.

Despite extensive research focused on these benchmarks, developing an agent with sufficient generality to solve them remains a significant challenge, as evidenced by the current leading approach on the ALFRED benchmark by Zhao *et al.* [2024], which achieves a success rate of only 62.35%.

3.2 Agent Generalization

Agent generalization refers to the ability of an artificial intelligence (AI) agent to transfer its knowledge and skills from one environment or task to another. It involves the capacity of the agent to apply the knowledge it has learned in one specific scenario to different, yet similar, situations.

Developing agents with strong generalization capabilities is an active area of research in AI and machine learning. Techniques such as transfer learning [Ada *et al.* 2019], meta-learning [Li *et al.* 2018], and domain adaptation [Truong *et al.* 2021] are often employed to enhance the generalization abilities of AI agents. These approaches aim to create models that can transfer knowledge and skills across different domains, tasks, or datasets, thereby improving their overall performance and applicability.

Language-driven generalization has gained significant attention, with **L**AN**C**ON-**L**EARN [Silva *et al.* 2021] introducing an innovative approach that transforms natural language instructions into semantically relevant goal embeddings for multi-task learning agents. Their method's effectiveness stems from applying attention mechanisms to language-specified goals, activating task-relevant skills by focusing on key components within language commands. This approach enhances both human-robot interaction for novice users and the ability of multi-task learning agents to adapt to unfamiliar tasks using established skills through Reinforcement Learning techniques in manipulation contexts. However, a limitation of their methodology is the requirement for substantial similarity between language commands used in transfer tasks and those employed during training.

Neural Task Programming (NTP) by Xu *et al.* [2018] combines few-shot learning from demonstration with neural program induction [Graves *et al.* 2014; Devlin *et al.* 2017]. This approach takes a task specification, such as a video demonstration, and systematically breaks it down into increasingly granular sub-task specifications through recursive decomposition. These specifications are then processed by a hierarchical neural program architecture where the lowest-level programs function as executable subroutines that directly interact with the environment.

Like our proposed work, Suddrey *et al.* [2016] presents an approach to learning hierarchical tasks through natural language instructions. This approach uses OpenCCG to parse natural language inputs with a fixed lexicon that must have all necessary lexical categories predefined. When the agent encounters an unknown task, it enters a learning state where it: Stops executing commands, stores subsequent commands as an explanation, processes these to generate a representation of the unknown task, and

then handles nested unknown tasks recursively. The system uses an extended SHOP2 [Nau *et al.* 2003] task formalism that enables dynamic task construction and problem resolution during execution. It also employs a novel hybrid HTN/forward planning algorithm that executes learned tasks and resolves issues during execution. Key limitations include the fixed lexicon requirement and the assumption that instructions explicitly contain the relations needed to derive implicit arguments during execution.

With the increased capabilities of LLMs, more research is being done on using LLMs for generalization and planning in robotics.

3.3 Planning with Large Language Models

With LLMs now able to write stories, create summaries and generate code [Wang *et al.* 2022], the abilities of these LLMs seem endless. However, their ability to plan and reason is under much discussion, with papers such as Valmeekam *et al.* [2022] questioning their abilities, while works such as, Huang *et al.* [2022] and Kojima *et al.* [2022] conduct experiments to showcase LLMs ability to be zero-shot learners and planners. Huang *et al.* [2022] find that sufficiently large pre-trained language models, when properly prompted, can effectively break down high-level tasks into mid-level plans without additional training. However, since these plans do not always correspond precisely to admissible actions, they propose a procedure that leverages existing demonstrations to semantically translate the plans into valid actions. While Kojima *et al.* [2022] shows that LLMs are decent zero-shot reasoners by simply adding "Let's think step by step" before each answer.

In recent years there has been an influx of working on language-conditioned Learning from Demonstration (LfD) [Schaal 1996] such as; Stepputtis *et al.* [2020] who introduce a method for integrating unstructured natural language into imitation learning involving experts providing demonstrations along with verbal descriptions to convey the underlying intent (e.g., "go to the large green bowl"). During training, this approach links the two modalities, encoding the relationships between language, perception, and motion. The resulting language-conditioned visuomotor policies can be adapted during runtime to new human commands and instructions, offering more precise control over the trained policies and reducing situational ambiguity. While this method uses GloVe embedding newer models such as those by Devlin *et al.* [2019] or Meng *et al.* [2024] can be used as replacements.

There has also been a recent interest in combining the tacit knowledge of an LLM with the explicit knowledge of classical planning techniques [Jin and Zhuo 2022]. LLM+P [Liu *et al.* 2023] is the first work to integrate LLMs into classical planning. The LLM+P system receives an explanation of a planning problem in natural language and generates an accurate or optimal plan to solve that problem. To accomplish this, LLM+P initially converts the explanation into a Planning Domain Definition Language (PDDL) [Aeronautiques *et al.* 1998] file. It then efficiently utilizes classical planners to rapidly discover a solution and subsequently translates the obtained solution back into natural language. It has been demonstrated that LLM+P outperforms standalone LLMs by

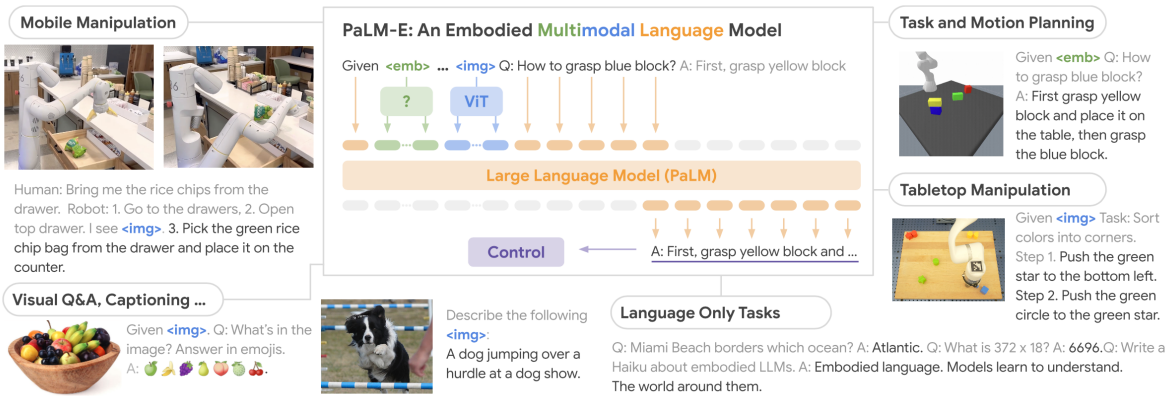

Figure 3.2: PaLM-E is a versatile multimodal language model designed for embodied reasoning, visual-language tasks, and language tasks. It transfers knowledge from visual-language domains to embodied reasoning, enabling tasks such as robot planning in environments with complex dynamics and physical constraints and answering questions about the observable world. PaLM-E processes multimodal sentences, where inputs from various modalities (e.g., images, neural 3D representations, or states, in green and blue) are combined with text tokens (in orange) and used as input to a large language model, trained end-to-end.

successfully generating accurate solutions for a significantly larger number of planning problems.

There are also efforts to rely solely on the LLM for planning. PaLM-E [Driess *et al.* 2023] introduces embodied language models that directly integrate continuous sensor inputs from an embodied agent, allowing the language model to make more grounded inferences for real-world sequential decision-making. Inputs such as images and state estimates are embedded into the same latent space as language tokens and processed by the self-attention layers of a Transformer-based LLM, just like text. PaLM-E leverages the 540B-parameter PaLM [Chowdhery *et al.* 2023] LLM alongside the 22B-parameter Vision Transformer [Dehghani *et al.* 2023] to create a 562B-parameter vision-language model.

There is also the work by Zhao *et al.* [2024] that proposes a hierarchical framework using LLMs for long-horizon decision-making. Their approach trains two LLMs: one to decompose complex tasks into subgoals and another to predict low-level actions. To extract training signals from unannotated data, they introduce an LLM-based reward model that integrates multimodal environmental feedback to generate automatic reward signals. Additionally, they propose Environment Preference Optimization (EPO), a method that ranks actions and subgoals based on estimated rewards and constructs a preference dataset to refine LLM-based agent training. By leveraging both annotated and unannotated data, their framework enhances agent performance with a broader training set. The work demonstrates state-of-the-art performance on the ALFRED benchmark, achieving a 62.35% success rate.

3.4 Conclusion

In this chapter, we review related work in the areas of Embodied Instruction Following (EIF), agent generalization, and planning with large language models (LLMs). The EIF section primarily focuses on research related to the ALFRED benchmark, including FILM Min *et al.* [2021] and its inspired works, alongside a general overview of the state-of-the-art in ALFRED.

Next, we explore agent generalization, highlighting various emerging methods that integrate language for improved generalization. Finally, we examine the growing field of LLM-based planning, outlining three key approaches. The last approach represents the current best-performing model on the ALFRED benchmark, demonstrating that LLMs are currently the most effective strategy for ALFRED.

Chapter 4

Methodology

This chapter outlines the system developed to enable generalized task learning for robots by abstracting hierarchical task methods from demonstration data and natural language instructions. The goal is to construct reusable, formalized plans that can generalize to new tasks and environments.

The methodology is organized around a three-stage pipeline:

1. Natural Language to Formal Language (Section 4.5) – Natural language instructions are converted into structured templates using prompting with large language models (LLMs);
2. Method Construction (Section 4.6) – Demonstration trajectories are transformed into symbolic representations, which are clustered and generalized into reusable task methods;
3. Clustering and Parametrising (Section 4.7) – These methods are converted into Hierarchical Domain Definition Language (HDDL) and executed by a symbolic planner.

In parallel, the system also supports runtime execution, which is handled by three operational subtasks: vision (semantic environment mapping), language (instruction parsing and matching), and planning (symbolic plan execution). While these two triads—the method creation pipeline stages and the runtime subtasks—serve different purposes, they interact closely. The method creation pipeline produces the abstract methods and structures that are consumed during online execution.

A key enabler across both processes is prompting with LLMs, which is used at multiple stages of the system:

1. Instruction Parsing: to convert natural language to structured HTN method templates;
2. Keyphrase Augmentation: to enhance clustering performance by generating semantic descriptors;
3. Evaluation Assistance: to assess plan accuracy by semantically analyzing outputs.

The remainder of this chapter details each of these components, beginning with a precise formulation of the problem being addressed, followed by an explanation of the proposed hypothesis, research questions, and the full system pipeline.

4.1 Problem Statement

The challenge lies in enabling zero-shot generalisation of household task execution using demonstration data. current approaches are grounded in specific trajectories and fail to generalise. the objective is to abstract general HTN methods from demonstrations that can be applied to new, unseen tasks and environments.

4.2 Hypothesis

We hypothesize that merging and parameterising all similar methods learnt from the demonstrations will create generalised methods that can accomplish unseen tasks in unseen environments. In this context, a method refers to a hierarchical task structure defined in HDDL that decomposes high-level tasks into subtasks usable by a classical planner.

4.3 Research Questions

To evaluate the performance and limitations of the proposed system, the following research questions were posed and explored;

1. How effectively can large language models translate natural language instructions into formal task representations suitable for hierarchical planning?
2. To what extent do clustering demonstration-derived methods enable generalization across similar but unseen tasks?
3. What impact do different embedding and augmentation strategies have on the quality and consistency of method generalization?
4. How well does the complete pipeline perform when executing plans on the ALFRED benchmark, and what are its limitations?
5. What are the key failure points in the pipeline, and how do individual components—such as semantic mapping and plan validation—affect overall task success?

4.4 Generalized Method Creation

Our approach to learning generalized methods makes use of three modules (figure 4.1);

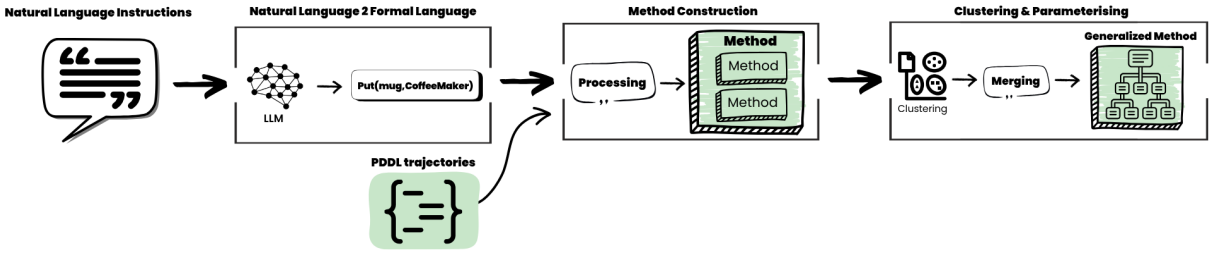

Figure 4.1: High-level overview of our proposed pipeline, showing the three modules. The Natural Language to Formal Language module (NL2FL) processes natural language instructions through an LLM with additional processing to produce formalized instructions. The Method Construction module leverages PDDL trajectories from ALFRED to create simple non-hierarchical methods, which are then combined into more complex hierarchical methods using the formalized instruction from NL2FL as the method name. Finally, the Method Clustering and Generalization module collects all constructed methods, clusters similar ones together, and combines and parameterizes these clusters into larger general methods.

- **Natural Language to Formal Language** (Section 4.5) is a process by which natural language instructions are transformed into formalized parameterized templates using LLMs. These templates simplify the processing during method generation and runtime execution.
- **Method Construction** (Section 4.6) where we use our training data to construct our grounded methods. We use these grounded methods as the basis of our generalized methods.
- **Clustering and parameterizing** (Section 4.7) is done to transform our collection of grounded methods into a few generalized methods. These generalized methods will be used during runtime to accomplish the given instructions.

4.5 Natural Language to Formal Language

In this section, we will discuss the natural language to formal language (NL2FL) pipeline as shown in Figure 4.2. The proposed pipeline is an updated version of the work done by Lindsay *et al.* [2017].

The first step is to process the natural language instructions into templates. Lindsay *et al.* [2017] used Named Entity Recognition and Parts of speech tagging to extract the templates. This approach is effective when dealing with simple and uniform sentences, but the complexity of the ALFRED instructions [Shridhar *et al.* 2020] makes it very difficult for these techniques to perform well. We look to improve on the work by Lindsay *et al.* [2017] by a more general pipeline that can handle complex and diverse sentences. As language models now lead the field in natural language understanding, excelling at sentiment analysis, named entity recognition and question answering [Ren

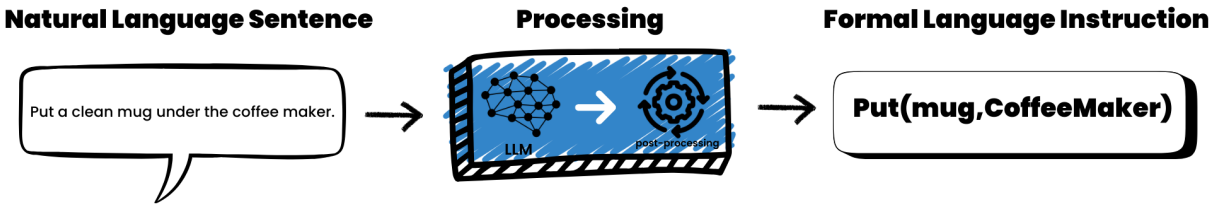

Figure 4.2: Overview of the Natural Language to Formal Language module. The module processes natural language instructions by sending them through a LLM to generate a templated sentence. This templated output then undergoes several stages of post-processing to ensure it fully conforms to the ALFRED conventions

2024; Patil and Gudivada 2024] we tested them and found great success using LLM to create the templates.

For the sake of cost and convenience, we elect to use open-source models that can be run locally (a full list of models can be seen in the results chapter 5). We implement a one-shot prompting method as it allows us to give the model examples of the two templates we want, so depending on the actions in the sentence they will be converted into either an action template or a navigation template (Figure 4.3). The model is given the following system prompt;

```
<s>[INST] <<SYS>>
The task is to identify the; Action, Object and location in the sentence if there
is one. For example, the sentence, "Pick up the yellow-handled knife against
the wall to the right of the sink."  will be in the form
    Action: pick up
    Object: yellow-handled knife
    From: the wall

For sentences that instruct navigation, "Turn left to face the dresser.".
the template should be
    Action: turn
    Direction: left
    Preposition: face
    Landmark: dresser

only output the answer in JSON format
<</SYS>>

Pick up the empty toilet paper roll from the floor. [/INST]
```

The foundational behaviour of an LLM is established through a system prompt, which is contained between <<SYS>>and <</SYS>> tokens. Model instructions are separately marked using <s>[INST] and [/INST]. To enhance consistency, we include an example sentence (“Pick up the empty toilet paper roll from the floor”) following the system prompt. Our testing shows that the model will produce more correct and consistent

outputs the more examples are in the context length. To improve speed and memory use, we opt to only use a single extra sentence.

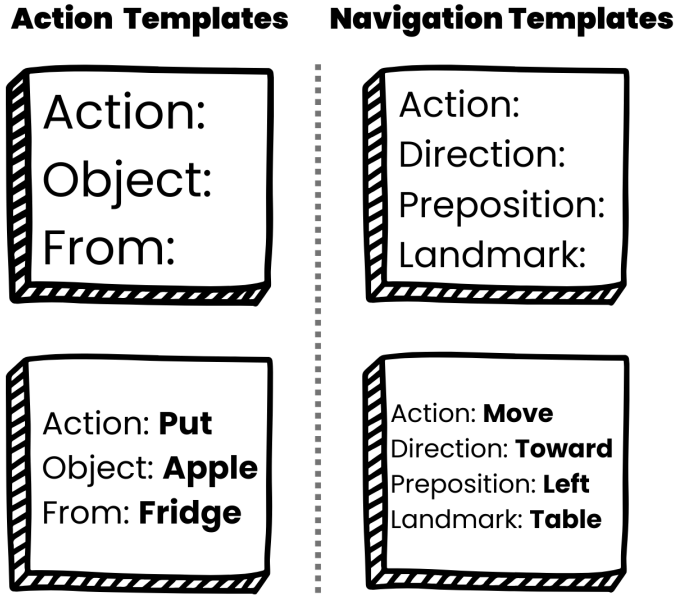

Figure 4.3: The system uses two templates (Action and Navigation) to handle different types of commands. It can quickly identify navigation commands by checking for a `Direction` key, then converts these into location instructions for the ALFRED environment.

Our pipeline utilizes separate Action and Navigation templates to streamline processing. This is a departure from Lindsay *et al.* [2017] as they only use a single action template. We opt for a dual-template approach as it simplifies the parsing mechanism, as the pipeline can identify Navigation commands by detecting the `Direction` key. When present, this key enables straightforward conversion to a `GoToLocation` method for execution within the ALFRED environment. This approach enables us to eliminate the clustering step found in the Lindsay *et al.* [2017] pipeline, as the various templates allow each template class to be transformed into consistent formulations.

LLMs often create correct templates but have issues with consistency and ALFRED item name errors. Therefore, before we can use the generated template post-processing is required. Our post-processing pipeline consists of three steps; formalizing, parameterizing and argument checking.

The formalization is an algorithmic process that transforms the template {"Action": "place", "Object": "spatula", "Into": "the sink"} into the phrase "place spatula sink". We achieve this by concatenating JSON components based on variables such as "Action", "Direction", "Object", among others, to generate the sentence in our desired format. Subsequently, articles (e.g., an, and, the, in...) are removed to enhance clarity and usability. This allows us to create a formalized sentence similar to Lindsay *et al.* [2017].

The parameterization step aims to output in an Action(object, location) format for easier conversion to HDDL methods. This is a straightforward translation since we recognize that each sentence begins with an action, followed by the `Object` and `Location`

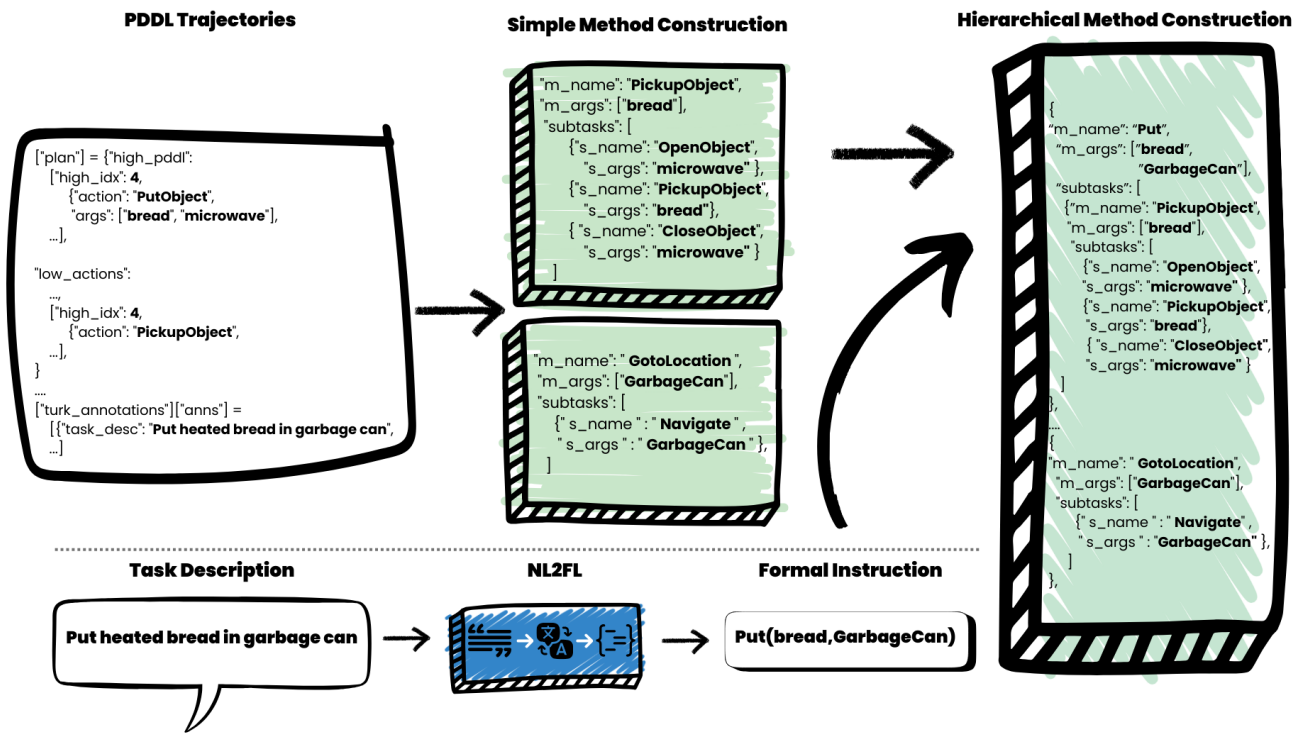

Figure 4.4: The Method Construction module functions through a two-phase process. Initially, it takes the PDDL trajectories provided for each task and converts them directly into simple non-hierarchical methods. These methods are then combined and organized into more complex hierarchical methods for each task in the training set. This approach allows the system to build sophisticated task representations by first establishing basic procedural components and then assembling them into comprehensive task structures.

variables. There may also be instances with a single variable, wherein the resulting format can be `Action(location)` or `Action(object)`. The final product is a JSON object, structured, for instance, as {“m_name”: “place”, “m_args”: [“spatula”, “sink”]}.

The concluding phase involves verifying that the arguments comply with ALFRED’s naming convention. Objects in ALFRED use CamelCase, so it is essential to align our variables accordingly. For instance, “spatula” should be updated to “Spatula”, and “counter_top” should be changed to “CounterTop”.

Having established our approach for translating natural language into formal language through the NL2FL pipeline, an essential component used both during method development and for processing new instructions at runtime. We can now proceed to examine our method construction process.

4.6 Method Construction

The initial stage of the Method Construction process involves developing basic non-hierarchical methods, which are methods where all subtasks are primitive tasks that cannot be broken down further. In the initial phase of constructing the simple non-hierarchical method, we utilize the trajectories from the ALFRED training dataset. The code segment (Listing 4.1) shows a portion of an ALFRED trajectory JSON file. This file contains details regarding the scene including object positions, instructions, task type, and images captured during the execution. However, our primary focus is on the plan expressed through PDDL.

```
1  ["plan"] = {"high_pddl":
2      ["high_idx": 4,
3           {"action": "PutObject",
4            "args": ["bread", "microwave"],
5      ...],
6  "low_actions":
7      ...,
8      ["high_idx": 4,
9           {"action": "PickupObject",
10     ...],
11 }
12 ....
13 ["turk_annotations"]["anns"] =
14     [{"task_desc": "Place heated bread on counter top",
15     ...]
```

Listing 4.1: ALFRED trajectory file snippet

As can be seen in Listing 4.1 the plan is separated into `high_pddl` (line 1) and `low_actions` (line 9). The `high_pddl` contains high-level actions like `GoToLocation` and `PutObject`, which require the execution of several primitive actions to complete. The `low_actions` are the primitive actions needed to complete the `high_pddl` actions. The `high_pddl` planner actions are linked to their corresponding `low_actions` via a `high_idx` (can be seen in lines 3 and 11 of Listing 4.1) variable which also links to the images in the training set used for the demonstration. So to construct our simple methods we use the `high_pddl` as the methods names and their corresponding `low_actions` as the subtasks, giving the JSON object shown in Listing 4.2;

```
1  {
2      "m_name": "PickupObject",
3      "m_args": ["bread"],
4      "subtasks": [
5          {
6              "s_name": "OpenObject",
7              "s_args": "microwave"
8          },
9          {
```

```
            "s_name": "PickupObject",
            "s_args": "bread"
        },
        {
            "s_name": "CloseObject",
            "s_args": "microwave"
        }
    ]
}
```

Listing 4.2: Example simple method

We generate the hierarchical methods by aggregating the simple methods derived from each ALFRED trajectory file. The process works as follows:

1. We take the `task_desc` (line 20 in Listing 4.1) and process it through the NL2FL pipeline.
2. This pipeline extracts both the method names and their corresponding arguments.
3. The previously constructed simple methods are then incorporated as subtasks within the larger hierarchical method structure.

This approach allows us to preserve the natural task hierarchy present in each trajectory while maintaining the formal representation of both high-level goals and their subtasks. We store these hierarchical methods in JSON files, with the final output being shown in Listing 4.3.

```
{
    "m_name": "put",
    "m_args": ["apple", "countertop"],
    "subtasks": [
        {
            "m_name": "GotoLocation",
            "m_args": [ "garbagecan"],
            "subtasks": [
                {
                    "s_name": "Navigate",
                    "s_args": "garbagecan"
                }
            ]
        },
        {
            "m_name": "PickupObject",
```

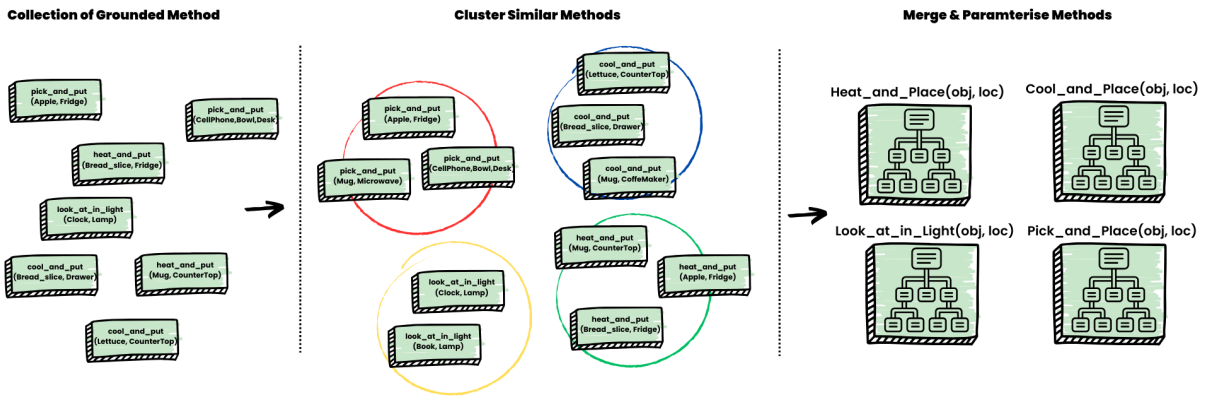

Figure 4.5: The Clustering Module transforms our collection of grounded methods into a smaller set of generalized methods through a multi-step process. Initially, it clusters all similar methods together based on their functional similarities. Once these clusters are formed, the module combines all methods within each cluster into unified methods. These combined methods are then parameterized to create general-purpose methods capable of handling any unseen task within the ALFRED environment.

```
            "m_args": ["apple"],
            "subtasks": [
                {
                    "s_name": "PickupObject",
                    "s_args": "apple"
                }
            ]
        },
    ......
```

Listing 4.3: Example hierarchical method

Having generated 21 023 hierarchical methods from the ALFRED trajectories, we turn to two key refinement steps: clustering similar methods together and parameterizing them. This process of consolidation and generalization allowed us to distil the large collection into a smaller set of generalized, reusable methods.

4.7 Clustering and Parametrising

Through this stage of the pipeline, we transform our collection of grounded hierarchical methods into just seven generalized methods. These seven methods are based on the 7 task types of AFLRED (see Figure 2.1), they are designed with sufficient flexibility and generalization to theoretically handle any task presented in the ALFRED dataset.

In order to make the clustering and generalization process tractable, we constrained the method learning to a fixed subset of 7 object classes. These were selected based on frequency and coverage across ALFRED demonstrations to provide a representative

yet manageable sample. While this simplification enables more consistent clustering results and reduces noise in downstream plan generation, it also introduces a degree of domain-specific bias. This reflects a broader theme in system design: Some domain knowledge was intentionally embedded to support feasibility and evaluability at the cost of full generality. We acknowledge this as a trade-off, and revisit its implications in Section 6.5.

4.7.1 Clustering

To enable clustering, we first convert the JSON-formatted hierarchical methods into sentences. This is done by a space-separated concatenation of the method's subtasks, leaving out the arguments. This approach ensures our clustering focuses on action sequences rather than the specific objects involved. We chose to adopt the Clustering via the LLM Keyphrase Expansion method presented by Viswanathan *et al.* [2024] (full explanation in Section 2.7.1).

To generate our keyphrases we gave the LLM the following prompt;

```
What is the unique action between each of these sentences? For a given user
query, provide a comprehensive set of keyphrases that could describe this
query's intent. These keyphrases should be distinct from those that might
describe queries with different intents. Generate the set of keyphrases as
a JSON-formatted list.

Query: "put GotoLocation PickupObject GotoLocation SliceObject GotoLocation
PutObject GotoLocation PickupObject GotoLocation CoolObject GotoLocation
PutObject."

Keyphrases: ["Cool", "cool object"]

Query: "examine GotoLocation PickupObject GotoLocation ToggleObject"

Keyphrases: ["Toggle", "examine object"]

Query: "heat GotoLocation PickupObject GotoLocation SliceObject GotoLocation
PutObject GotoLocation PickupObject GotoLocation HeatObject GotoLocation
PutObject."

Keyphrases: ["Heat", "heat object"]
```

We once again implement a few-shot prompting approach to solve this challenge, to give the LLM the desired output and format. After evaluating multiple language models, we selected GPT-4o-mini [Achiam *et al.* 2023] as it provided the highest quality results. Since the keyphrases could be generated ahead of time rather than during runtime, we can utilize the batch API, optimizing both cost and processing time. Once the keyphrases are generated they are stored in text files for later use.

We then append the keyphrases to the sentence representation of our methods so that they can be encoded together. We use the SFR-Embedding-Mistral [Meng *et al.* 2024] model[1] for our embeddings. This model uses the same methods that were described in Section 2.6, building on top of the e5 model proposed by Viswanathan *et al.* [2024] and Mistral [Jiang *et al.* 2023]. We apply the same template as Equation 2.4, with our task_definition being "Identify the unique action of the given sentences" and the query consisting of our modified sentences, we feed this new prompt into SFR-Embedding-Mistral to get our embeddings.

With the embeddings (vector of length 4096) now in hand, we can move forward with the clustering process. Given that the ALFRED benchmark comprises seven distinct task types, we aim to create seven clusters. Armed with this information, we opt for the k-means algorithm, as it suits our predefined understanding of the number of clusters.

4.7.2 Generalized Method Creation

Once we have our clusters, we look at creating our generalized methods. These generalized methods allow us to accomplish any unseen instructions given to us from the ALFRED domain. We choose the method closest to the centroid to act as our base method. We then iteratively work to combine with the other methods in the cluster. We will be implementing a slightly modified version of the HTN merging algorithm from Mohseni-Kabir *et al.* [2014]. The full process consists of the following;

1. Parameterise,
2. Add preconditions,
3. Merge.

The initial phase of merging involves making the method more generic through parameterization. Rather than having a method designed for a specific case (like putting an apple in the microwave, see Listing 4.4), we transform it into a more flexible version that can handle any object and destination. For example, the method becomes capable of putting any item into any storage location. This generalization not only simplifies the method but also streamlines later stages in the process.

```
{
    "m_name": "put",
    "m_args": ["Apple","Microwave"],
    "subtasks": [
        {
            "m_name": "GotoLocation",
```

[1]The leading model on **M**assive **T**ext **E**mbedding **B**enchmark (MTEB) [Muennighoff *et al.* 2023] at the time

```
            "m_args": ["GarbageCan"],
            "subtasks": [ ....]
        },
        {
            "m_name": "PickupObject",
            "m_args": ["Apple"],
            "subtasks": [....]
        }
```

Listing 4.4: Example method before parameterising. Full method can be seen in Appendix 1.

This step is simple, we iterate through all the subtasks in the method and replace any arguments in the subtask that match the main method arguments. These arguments are replaced with `"obj"` and `"loc"`, respectively depending on their type. Resulting in a method that resembles Listing 4.5:

```
{
    "m_name": "put",
    "m_args": ["obj","loc"],
    "subtasks": [
        {
            "m_name": "GotoLocation",
            "m_args": ["GarbageCan"],
            "subtasks": [....]
        },
        {
            "m_name": "PickupObject",
            "m_args": ["obj"],
            "subtasks": [....]
        }
```

Listing 4.5: Example method after parameterising. Full method can be seen in Appendix 2.

The methods we have up until this point are standard HTN methods, however, they lack the necessary preconditions required to support our planning processes. The next step aims to address this issue.

In our implementation, we simplify HTN preconditions to just two essential types: `on` and `has`. The `on` precondition describes where the primary object is in the environment - for example, specifying that an Apple is located on a CounterTop. Meanwhile, the `has` precondition identifies which objects and locations are involved in executing a particular method. Further details are provided in Section 4.8.3, where planning is discussed.

We determine the `on` precondition by analyzing the sequence of subtasks in the method. Specifically, we look for the first occurrence where a `"GotoLocation"` subtask is immediately followed by a `"PickupObject"` subtask. This sequence reveals the spatial rela-

tionship between objects. For instance, if the method has us first go to the countertop and then pick up an apple, we can infer that the apple is located on the countertop. This inference allows us to establish the `on` precondition.

The `has` precondition simply lists all objects and locations involved in the method. the preconditions are returned in JSON format, e.g.;

```
"preconditions": [
            {
                "has": [
                    "GarbageCan",
                    "SinkBasin",
                    "Faucet"
                ]
            },
            {
                "on": [
                    "obj",
                    "GarbageCan"
                ]
            }
        ]
```

Listing 4.6: Example precondition

The final merge step uses the algorithm from Mohseni-Kabir *et al.* [2014]. First, we find the Longest Common Sequence (LCS) between the base method and the new method. If there is no LCS, we simply add the new method as an alternate method, which we call a recipe. If we find an LCS, we extract it from both methods to create what we call a common method. We then insert this common method back into both the base and new methods where the LCS was originally found. Finally, we add the new edited method as an alternate recipe.

After establishing the initial relationship between the base method and the first alternate method, the process becomes more comprehensive when handling multiple methods. The LCS comparison must be expanded to check against all existing recipes and common methods, not just the base method. When an LCS is found between the new method and one of the existing recipes, we follow the same extraction and replacement process described earlier. However, if the LCS matches an existing common method, the process is simplified – we only need to extract the matching sequence and replace it with a reference to the already established common method. Our generalized methods are stored in JSON format, as shown in the examples below.

```
"recipe0": {
        "m_name": "put",
        "m_args": ["obj","loc"],
        "subtasks": [
            {
                "m_name": "GotoLocation",
```

```
                "m_args": ["SinkBasin"],
                "subtasks": [....]
            },
            {
                "m_name": "CleanObject",
                "m_args": ["obj"],
                "subtasks": [....]
            },
            {
                "m_name": "GotoLocation",
                "m_args": ["Cabinet"],
                "subtasks": [....]
            },
            {
                "m_name": "PutObject",
                "m_args": ["SoapBar","Cabinet"],
                "subtasks": [....]
            }
        ]
    }....
```

Listing 4.7: Snippet of a recipe before LCS extracted.

```
"recipe0": {
        "m_name": "put",
        "m_args": ["obj", "loc"],
        "subtasks": [
            {
                "m_name": "GotoLocation",
                "m_args": ["SinkBasin"],
                "subtasks": [....]
            },
            {
                "m_name": "common40",
                "m_args": ["obj", "loc"]
            },....
```

Listing 4.8: Snippet of a recipe with common method

An example of a complete common method can be seen in Appendix 3.

4.7.3 Conversion to HDDL

As described earlier in Section 4.7, the method generalization process was constrained to a selected subset of 7 object classes to improve consistency in clustering. Once we have our generalized methods (`heat_and_place`, `cool_and_place`, `pickTwo _and_place`,`clean_and_pla

pick_and_place, examine_in_light, pick_and_place_with _movable_recep) we need to convert it from JSON to an HDDL Domain file for use by the PANDA planner.

Before constructing the HDDL representation, several simplifying assumptions were necessary to ensure compatibility between the abstracted task methods and the requirements of symbolic planners. These included: The manual definition of symbolic predicates such as `is-open`, `is-sliced`, and `is-clean`, to encode task-relevant object states that are not directly inferred from raw perception but are essential for plan reasoning, The use of a restricted set of object classes (seven in total), chosen to reduce clustering noise and ensure consistency in generalization; and the manual specification of action preconditions and effects, derived from analysis of demonstration data rather than learned, to ensure the feasibility of generated HDDL plans.

These assumptions reflect pragmatic design decisions made to support tractable plan generation and integration with the ALFRED benchmark. However, they also introduce handcrafted elements that limit the full autonomy and generality of the proposed approach. Their broader implications are discussed in Section 6.5.

Before we convert to HDDL, there are sections of the domain which we hardcode. These sections are built using prior knowledge obtained from the ALFRED domain. Future work should focus on automatically generating the domain file. The first is the predicates;

```
(:predicates
    (is-sliced ?x - object)
    (at-location ?loc - location)
    (in-obj ?obj - location)
    (is-on ?obj - object)
    (closed ?obj - object)
    (in-hand ?obj - object)
    (has ?loc - location)
    (on ?obj1 - object ?obj2 - object)
    (hand-empty)
)
```

These are the boolean statements we use to indicate the actions' effects. The next section is the task definitions for the simple methods. These methods are universal across multiple tasks, so they are generally useful despite being generated from prior knowledge.

```
(:task PutObjectIntoLocation :parameters(?obj1 - object ?loc - location) )
(:task PutObjectIntoObject :parameters(?obj1 - object ?obj2 - object) )
(:task PickupObject :parameters(?obj - object) )
(:task HeatObject :parameters(?obj1 - object ?microwave - location) )
(:task CoolObject :parameters(?obj1 - object ?fridge - location) )
(:task GotoLocation :parameters(?loc - location) )
(:task GotoObject  :parameters(?obj - object) )
(:task SliceObject :parameters(?obj - object) )
(:task CleanObject :parameters(?obj1 - object ?sinkbasin - location
```

```
                                ?faucet - object) )
(:task ToggleObject :parameters(?obj - object) )
(:task achieve-goals :parameters(?obj - object ?loc - location) )
```

Every method in an HDDL domain file needs to accomplish a task because our simple methods are not detailed in the JSON, have to specify the tasks, the methods and the basic actions.

```
(:method ToggleObject
    :parameters(?obj - object)
    :task (ToggleObject ?obj)
    :precondition ()
    :ordered-tasks (and
        (ToggleObjectOn ?obj)
    )
)

.....

(:action ToggleObjectOn
    :parameters (?obj - object)
    :precondition ()
    :effect (and (is-on ?obj))
)
```

Once we have all the hardcoded sections, we can move to convert the recipes in JSON to methods in HDDL.

1. the `:method` will be the method name, so either recipe<num> or common<num>, where <num> is the method number.

2. the `:paramters` is a superset of all the items listed in the arguments and preconditions.

3. `:task` is the goal the method needs to accomplish, to ease the problem definition all recipes accomplish the same task `(achieve-goals ?obj ?loc)`. The common methods are slightly different, to avoid them being considered valid standalone plans, they accomplish unique tasks corresponding to their name.

4. the `:precondition` consists of the same set of preconditions laid out in the JSON.

5. `:ordered-tasks` is the list of method names in subtasks. Due to the explicit nature of the PANDA and HDDL we needed to make adjustments to method names based on the type of item they are interacting with.

   Goto: if the method contained either the `loc` variable or any ALFRED receptacle (e.g. countertops, shelves, side tables...) the method was `GotoLocation` else it was `GotoObject`

PutObject: The same rules as above were used for the `PutObject`, with the names changing to `PutObjectIntoLocation` and `PutObjectIntoObject`, respectively.

This simple conversion gives us valid HDDL methods (and task definitions regarding the common methods), which can be seen in the below examples;

```
(:task common0 :parameters (?sinkbasin - location ))
....
....

(:method recipe0
    :parameters (?obj - object ?loc - location ?pot - location ?sinkbasin -
                 location  ?faucet - object )
    :task (achieve-goals ?obj ?loc)
    :precondition (and (has pot) (has sinkbasin) (has faucet) (on pot ?loc))
    :ordered-tasks (and
        (GotoLocation ?loc )
        (PickupObject pot )
        (common0 )
        (CleanObject pot )
        (GotoLocation ?loc )
        (PutObject pot ?loc )
    )
)

....

(:method common0
    :parameters (?sinkbasin - location )
    :task (common0 ?sinkbasin)
    :precondition (and (has sinkbasin) )
    :ordered-tasks (and
        (GotoLocation sinkbasin )
    )
)
```

Once we have created all the domain files, we are ready to run our tests on the ALFRED environment.

4.8 Runtime Pipeline

The runtime pipeline is our system for testing our generalized methods as well as the NL2FL module on unseen tasks. Our pipeline is based on **F**ollowing **I**nstructions in

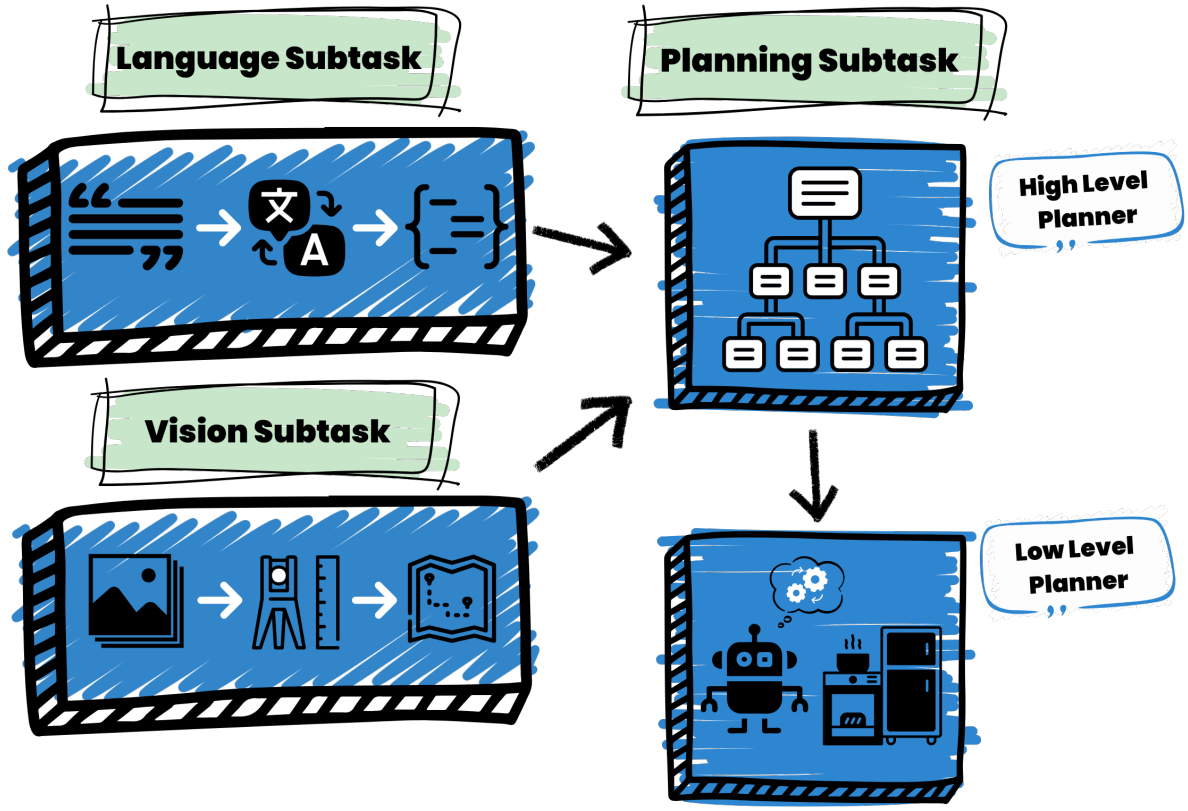

Figure 4.6: Our Runtime Pipeline evaluates performance on the ALFRED benchmark through three integrated subtasks. The Language subtask employs the NL2FL module (Section 4.5) to transform new instructions into formalized instructions. The Vision subtask generates a semantic map of the environment and maintains an inventory of all objects present. The Planning subtask combines the object list from the Vision subtask with arguments from the formalized instruction to search through the domain files to create a high-level plan, which is subsequently passed to a low-level deterministic planner for execution within ALFRED.

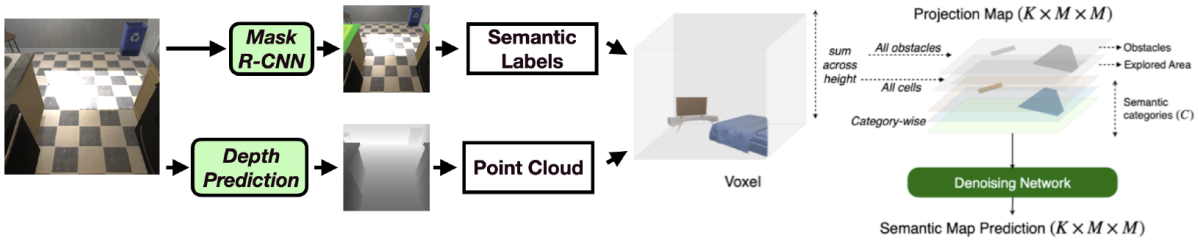

Figure 4.7: Semantic mapping Module [Min *et al.* 2021]. The semantic mapping module converts first-person perspective RGB input into a depth map and instance segmentation using fine-tuned MaskRCNN [He *et al.* 2017] and Blukis *et al.* [2022] depth prediction method. The depth map is processed into a point cloud with predicted semantic categories, and then transformed into a voxel representation. Summing voxel values along the vertical axis collapses scene height into a semantic map, which is locally refined and accumulated over time.

Language with **M**odular methods (FILM) proposed by Min *et al.* [2021][2]. The pipeline is separated into three subtasks; Language, Vision and Planning (Figure 4.6). **The Language subtask** uses the NL2FL module (Section 4.5) to convert natural language instructions into formal representations. **The Vision subtask** creates a semantic map of the environment and tracks all objects present in an inventory. **The Planning subtask** combines the object inventory from the Vision subtask with parameters from the formalized instructions to search domain files and generate a high-level plan, which is then handed off to a low-level deterministic planner for execution within ALFRED. We give details on each subtask in the sections that follow.

These subtasks correspond to the runtime execution layer of the system. While they differ from the pipeline's three-stage Generalized Method Creation process described earlier, they consume its output. For instance, the vision system interprets objects for the planner, which in turn uses the methods constructed during clustering and abstraction.

4.8.1 Vision Subtask

The vision subtask maps the environment, identifying objects and their positions. This information is crucial for planning and provides the necessary data for navigation.

We implement the semantic mapping module proposed by Min *et al.* [2021]. An overview can be seen in figure 4.7. We construct the map by using the first one hundred time steps to randomly move around the environment. The semantic mapping module process begins by converting egocentric RGB input into a depth map and instance segmentation using MaskRCNN [He *et al.* 2017] and the depth prediction method by Blukis *et al.* [2022]. These pre-trained models were fine-tuned on the training scenes of AL-

[2]Code available at https://github.com/soyeonm/FILM.

FRED. First, the depth map is processed, it is then restructured into a point cloud where each point is assigned predicted semantic categories. The point cloud is converted into a voxel representation, and the semantic map is obtained by summing the voxel values along the vertical axis, collapsing the height of the scene. The map undergoes local updates and is aggregated over time.

The semantic map is an allocentric $(C + 2) \times M \times M$ binary matrix, where the number of object categories is represented by C. Each $M \times M$ grid cell represents a 5cm $\times$ 5cm area. The C channels indicate detected objects, while the two additional channels show obstacles and explored areas. These $C + 2$ channels offer a spatial/semantic overview. M is set to 240 (12 meters in real terms), and C includes 28 receptacle objects, like "Table" or "Bathtub", plus extra subgoal objects for the task. An example of the semantic mapping process can be seen in Figure 4.8

4.8.2 Language Subtask

This subtask is concerned with transforming the natural language instruction into a structured representation via the NL2FL pipeline outlined in Section 4.5, and identifying the task type using an LLM.

This step makes the instruction more interpretable for the other process. We take the instruction and pass it through the NL2FL pipeline (template, formalize, parameterise, check arguments) so that we can get – for example, {"m_name": "Put", "m_args": ["Book", "Cabinet"]}. This is useful as it gives us our `obj` and `loc` variables for planning.

The next step is determining the task type. for this, we feed the following prompt into an LLM;

```
<s>[INST] <<SYS>>
You are a system that identifies tasks given an instruction. the tasks are
as follows:

1.HeatandPlace (tasks that involve a heated object or a microwave or
                a microwaved object)
2.PickandPlace
3.CoolandPlace (tasks that involve cooling an object or a fridge)
4.CleanandPlace
5.PickTwo
6.LookatObj
7.PickMovable

answer should be in the form: task_type = <<TASK_TYPE>> and nothing else
<</SYS>> [/INST]
```

We use the LLM's answer to choose one of the seven domain files for planning.

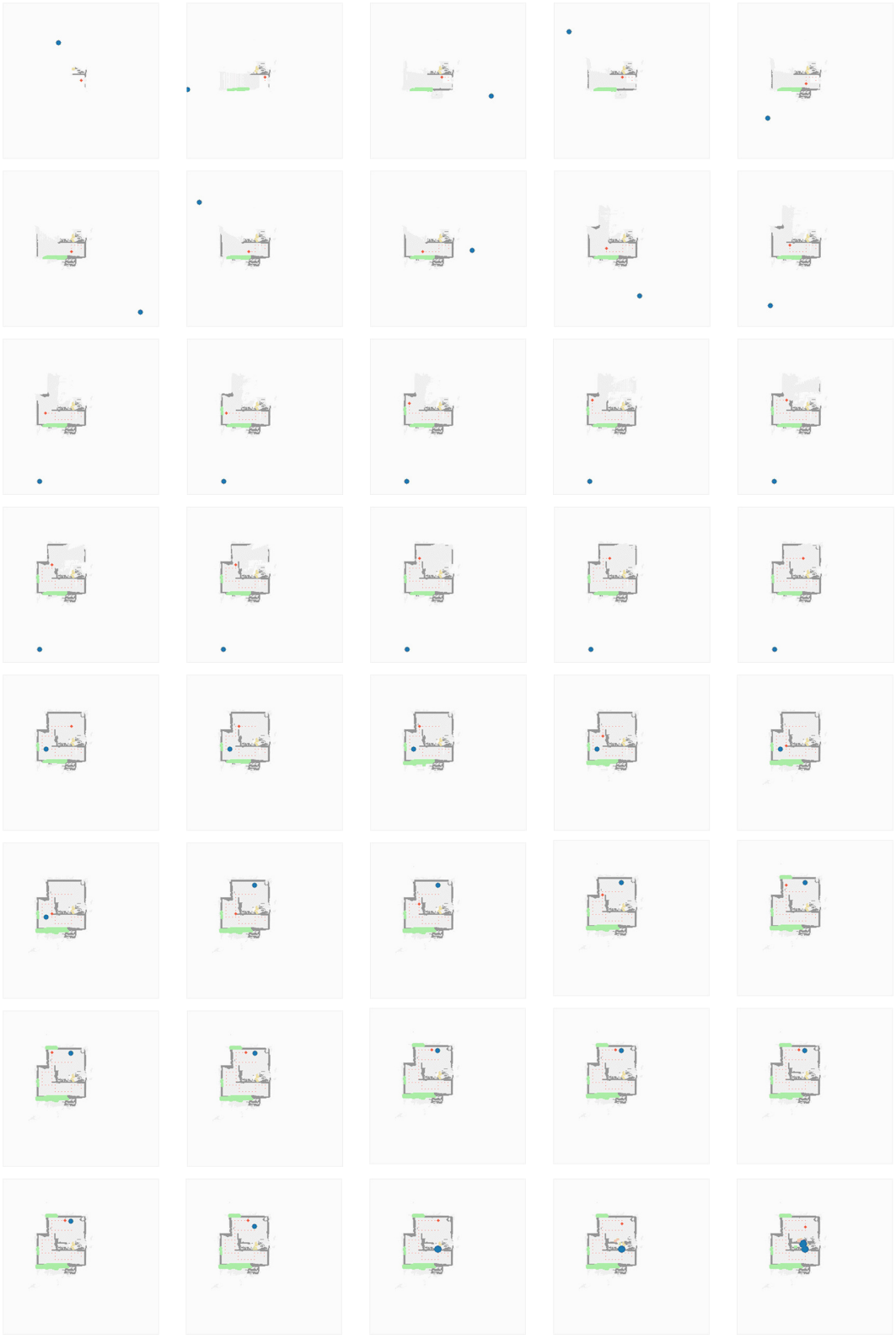

Figure 4.8: Map generation of a kitchen (every 13th frame used for example). The red dot is the agent, the dotted line is the path travelled, the blue dot is the randomly selected goal used for mapping, the grey represents walls, and the green shows cupboards/cabinets. The yellow and orange represent objects and receptacles, respectively.

4.8.3 Planning Subtask

This subtask handles both generating and executing the plan, it is broken into two levels; high-level planning and low-level planning. High-level planning is handled by the PANDA planner (Section 2.4) and the low-level uses the discrete planner proposed by Min *et al.* [2021].

High-level planning

The high-level planner is responsible for the abstract planning of the goal and decomposing of the HTN. For the PANDA planner to be able to find a plan, two files are necessary, the domain file and the problem file. We already have domain files, so we need to generate unique problem files for each test problem.

To create the problem file, we need to compile three key components: the object and location involved in the task, a comprehensive list of objects and receptacles in the scene, and a set of initial scene conditions.

The object and location are derived from the formalised sentence obtained through the language subtask (Section 4.8.2). The vision subtask provides a straightforward method for identifying scene objects, as the mapping process generates a list of all objects and receptacles currently visible to the agent.

The initial conditions are crucial for the planner to identify valid plans, and they should align with specific method preconditions—namely the `has` and `on` conditions. The `has` condition is simply a list of all objects in the scene. The `on` condition is determined using spatial coordinates from the mapping process: if an object's position falls within a predetermined threshold of a receptacle, we consider the object to be `on` that receptacle.

Due to the practical limitations of the ALFRED benchmark and current planners, several simplifications were introduced. These include the manual definition of predicates to represent object states and the use of seven predefined classes. While these violate the ideal of autonomous learning, they were necessary to complete the planning formalism and evaluate end-to-end performance.

In many ALFRED tasks, a common sub-goal involves slicing or cutting objects. We have developed a straightforward approach to handle this requirement: when the instruction contains words like "slice", "cut" or "knife" we add a specific goal condition `is-sliced` to the problem formulation.

By introducing the `is-sliced` goal, we ensure that the planner will prioritize and select plans that include the slicing action for the specified object. This method allows the planning system to explicitly address the slicing requirement as a critical part of task completion, rather than treating it as an optional or incidental action.

All this gives a problem file that resembles;

```
(define
 (problem p1)
```

```
(:domain ALFRED)

(:objects
       Mug - object
       Spoon - object
       Bowl - object
       ButterKnife - object
       PepperShaker - object
       Potato - object
       Knife - object
       Lettuce - object
       Pan - object
       Spatula - object
       Tomato - object
       Kettle - object
       Egg - object

       ....

)
   (:htn :tasks (and
            (achieve-goals Bread DiningTable)
 )

(:init
       (has Mug)
       (has Spoon)
       (has Bowl)
       (has ButterKnife)
       (has PepperShaker)
       (has Potato)
       (has Knife)
       (has Lettuce)
       (has Pan)
       (has Spatula)
       (has Tomato)
       (has Kettle)
       (has Egg)
       (has Plate)

       ....

       (on Egg SinkBasin)
       (on Spatula Cabinet)
       (on StoveKnob Cabinet)
       (on SaltShaker Coffeemachine)
```

```
        (on Pot Cabinet)
        (on Pan CounterTop)
        (on Mug Fridge)
        (on SoapBottle Cabinet)
        (on Bread Drawer)
        (on Kettle Drawer)
        (on Egg Microwave)
        (on Tomato Cabinet)
        (on Faucet DiningTable)
        (on Pan Cabinet)

        ....
 )
    (:goal (and
        (is-sliced Bread)
    ))
)
```

Subsequently, we provide PANDA with the problem and domain files, enabling it to search for the plan (the entire PANDA process was described in Chapter 2.4). If a plan is found, PANDA outputs a plan in the form;

```
Navigate[CounterTop]
PickObject[Mug]
Navigate[Microwave]
OpenObject[Microwave]
PutObject[Mug,Microwave]
CloseObject[Microwave]
ToggleObjectOn[Microwave]
ToggleObjectOff[Microwave]
OpenObject[Microwave]
PickObject[Mug]
CloseObject[Microwave]
Navigate[Mabinet]
OpenObject[Mabinet]
PutObject[Mug,Mabinet]
CloseObject[Cabinet]
```

or else it will output `proven unsolvable` and we will move on to the next test. If a plan is found we will pass the plan to the low-level planner, which will carry out the proposed plan in the ALFRED environment.

Low-Level Planning

The low-level controller is responsible for completing the actions and dealing with the stochasticity of the environment. The low-level planner by Min *et al.* [2021] uses a

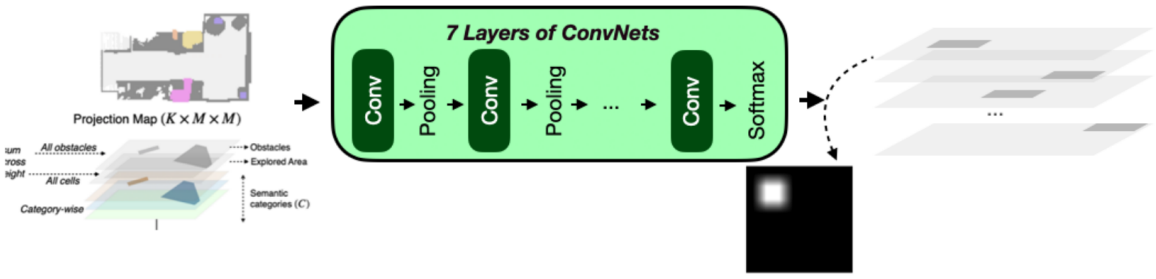

Figure 4.9: Semantic search policy [Min *et al.* 2021]

deterministic policy as research by Chaplot *et al.* [2020] showed it performs comparably to a learned local navigation policy.

The plan is given to the policy in the form [$(obj_1, action_1)$, ... , $(obj_k, action_k)$], where $(obj_i, action_i)$ is the task at hand. When an object (obj_i) is present in the current semantic map, the agent selects the nearest instance of that object as its navigation goal. If no such object exists in the map, the agent instead chooses a goal sampled from the semantic search policy.

The semantic search policy (Figure 4.9) generates a broad, 2D distribution that identifies potential locations for small target objects. This distribution is based on a semantic map containing only 28 types of receptacle objects, such as counter tops and shelves. The primary function of this policy is to predict probable search areas for small objects by analyzing the spatial arrangement of larger receptacle objects. It learns to infer where smaller objects are likely to be found based on the context and layout of larger objects in the environment. The policy is developed using supervised learning techniques.

To reach the goal, the agent uses the Fast Marching Method [Sethian 1996] for navigation. Upon arriving at the stop distance, the agent begins a systematic rotation, turning left 8 times at different camera horizons (0, 45, 90 degrees, etc.) until obj_i is detected in its first-person view. Once obj_i appears in the present frame, the agent decides to execute the corresponding action $(action_i)$ based on two key criteria:

1. Whether obj_i is situated near the "middle" of the frame.

2. Whether the smallest depth value to obj_i is within a visible range of 1.5 meters.

If these conditions are not met, the agent will either sidestep to maintain obj_i at the centre of the frame or keep rotating left at horizons 0 and 45 until obj_i becomes visible within the specified range.

If the agent attempts to execute $action_i$ but is unsuccessful, it will move backwards and update its map.

4.9 Conclusion

This chapter outlines our Generalized Method Construction and Runtime pipelines, detailing how the Natural Language to Formal Language (NL2FL) module uses an LLM to transform natural language instructions into formalized instructions. It explains the Method Construction module's approach of utilizing PDDL trajectories from ALFRED to build simple non-hierarchical methods before combining them into complex hierarchical methods with names derived from NL2FL's formalized instructions. The Generalized Method Construction section concludes by explaining how the Method Clustering and Generalization module aggregates all constructed methods, clusters similar ones together, and combines and parameterizes these clusters to create more comprehensive general methods.

We detail the three modules of the Runtime pipeline as follows; The Language Subtask employs the NL2FL module to transform natural language instructions into formal representations, while the Vision Subtask generates a semantic map of the environment and maintains an inventory of all present objects. The Planning Subtask integrates the object inventory from Vision with parameters from the formalized instructions to search domain files and develop a high-level plan, which is subsequently transferred to a low-level deterministic planner for execution within ALFRED.

Chapter 5

Results

To evaluate the feasibility of our approach, we perform a complete system and component-level evaluation. In the sections that follow, we break down how we evaluated each of the components.

5.1 Assessing clustering capability

Clustering our methods proved challenging as the sentence representation presented significant challenges for clustering. With sentences like: “examine GotoLocation Navigate PickupObject PickupObject GotoLocation Navigate ToggleObject ToggleObjectOn”. To determine the optimal approach, we conduct a comprehensive search exploring different clustering methods, embedding techniques, data augmentations, and similarity metrics.

5.1.1 Evaluation criteria

We use the following three metrics to evaluate how clustering across every experiment.

- **Silhouette Score** is a metric used to evaluate the quality of clustering in unsupervised machine learning. It measures how well each data point fits within its assigned cluster compared to other clusters. The score ranges from -1 to 1, with higher values indicating better clustering
- **Davies-Bouldin Index** measures how well a clustering solution separates data points into distinct groups, with lower values indicating better clustering.
- **Calinski-Harabasz Index** is a clustering evaluation metric that measures the ratio of between-cluster variance to within-cluster variance. Higher values indicate better-defined clusters.

Algorithms	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
DBScan	0.05	**1.08**	7.10
Kmeans	**0.13**	2.34	**213.77**

Table 5.1: Comparison of Clustering Algorithms. For the Silhouette Score, the higher value is better. The Davies-Bouldin Index measures better clustering with lower values. While the Calinski-Harabasz Index indicates a better-defined cluster with higher values. The bold values indicate the best results per row.

5.1.2 Comparison of Clustering Algorithms

Our initial tests are to see the best-suited clustering algorithm for our problem. We test three different unsupervised clustering algorithms, these are;

- **DBSCAN** – a density-based clustering algorithm that groups points based on how closely they are packed together.
- **K-means** – a popular unsupervised clustering algorithm that partitions data into K distinct clusters. As we know there are seven tasks in the ALFRED, we can use this as our k.

For our tests, we use SFR-Embedding-Mistral [Meng *et al.* 2024] as our encoder. We test using all of our 21 023 generated methods.

During testing, we use multiple DBScan versions with various epsilon values, with the best performance achieved at epsilon = 0.01. The results (seen in Table 5.1) here show that k-means is the best-performing algorithm, and based on that we select k-means as our primary algorithm for subsequent work.

5.1.3 Impact of Data Representation

Embedding Methods Comparison

We run multiple tests to find the best encoder for our algorithm;

- **Llama-2** [Touvron *et al.* 2023] – a family of open-source large language models developed by Meta AI.
- **SFR-Embedding-Mistral** [Meng *et al.* 2024] – a text-embedding model that builds upon the solid foundations of E5-mistral-7b-instruct [Wang *et al.* 2024] and Mistral-7B-v0.1 Jiang *et al.* [2023].
- **E5** [Wang *et al.* 2024] – a family of state-of-the-art text embeddings.
- **Mistral** [Jiang *et al.* 2023] – a 7B language model engineered for superior performance and efficiency.

Models	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
SFR-Embedding-Mistral	**0.13**	**2.34**	**213.77**
e5-mistral-7b-instruct	0.12	2.22	211.24
Mistral-7b	0.10	2.18	199.72
Llama-2-7b	0.10	2.10	194.53

Table 5.2: Table comparing different embedding models. The experiment uses all of our 21 023 generated methods. The table shows that SFR-Embedding-Mistral is the best-performing model while Llama-2-7b is the worst.

Metric	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
Cosine similarity	0.10	2.72	155.21
Euclidean distance	**0.13**	**2.34**	**213.77**

Table 5.3: Table comparing different similarity metrics, this experiment uses the same 21 023 methods as the other experiments as input.

As can be seen in Table 5.2, Llama-2 is the worst-performing model, unsurprisingly as it is one of the weaker models on the MTEB leaderboard[1]. As both e5-mistral-7b-instruct and SFR-Embedding-Mistral are models trained specifically for embeddings they outperform the general LLMs, with SFR-Embedding-Mistral having the best performance as it is a further fine-tuned e5-mistral-7b-instruct which was fine-tuned from Mistral-7b.

Similarity Metrics Analysis

We then look to compare different similarity metrics, these being;

- **Cosine similarity** – a measure used to determine how similar two vectors are, regardless of their magnitude. It calculates the cosine of the angle between two vectors, with values ranging from -1 to 1.
- **Euclidean distance** – helps determine how similar or different data points are from each other. It measures the straight-line distance between two points in Euclidean space.

We apply both of these metrics using the K-means algorithm with the SFR-Embedding-Mistral embedding model, obtaining the following results.

Here (Table 5.3), it can be seen that the Euclidean distance slightly outperforms the Cosine similarity. All following experiments will be using Euclidean distance as the similarity metric.

[1] https://huggingface.co/spaces/mteb/leaderboard

Variations	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
Full	0.13	2.34	213.77
No arguments	0.15	2.00	270.98
No subtasks	0.13	2.57	217.66
Neither	**0.31**	**1.46**	**668.11**

Table 5.4: Table comparing the effectiveness of different method variations

	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
Neither variation	0.31	1.46	668.11
Neither + KeyPhrase	**0.37**	**1.33**	**675.04**

Table 5.5: Table comparing different data augmentation techniques

5.1.4 Method Variations Analysis

When we speak about method variations we are referring to how much of the method we are going to use for clustering for example here is a full method;

"put rag metal_rack GotoLocation toilet Navigate toilet PickupObject cloth PickupObject Cloth GotoLocation cart Navigate cart PutObject cloth cart PutObject Cloth Cart"

This method contains all the subtasks and arguments of the put(rag,metal_rack) method. We evaluate variations that omit subtasks, arguments, or both components. Examples of those are;

- No arguments: "put GotoLocation Navigate PickupObject PickupObject GotoLocation Navigate PutObject PutObject"
- No subtasks: "put rag metal_rack GotoLocation toilet PickupObject cloth GotoLocation cart PutObject cloth cart"
- Neither:"put GotoLocation PickupObject GotoLocation PutObject"

As can be seen in Table 5.4 the difference in method variation has a great impact on the clustering. We theorize this is because as you strip away at the methods, they are left with the more unique aspects of the methods allowing for better similarity calculations. This can be seen from the fact that the best-performing variation is the variation with neither the subtasks nor arguments.

5.1.5 Data Augmentation Techniques

To further improve clustering we looked into LLM augmenting clustering, specifically the addition of keyphrases to further improve method uniqueness (Section 2.7.1). The results (Table 5.5) show that this augmentation further increased the performance of our clustering.

Model	Accuracy
Mistral-7B-Instruct-v0.2	35
Llama-2-7b-chat	**42**
Gemma-2-2b-it	29

Table 5.6: Comparing the accuracy of different LLM on template creation, consists of the results from the 1 529 unseen demonstrations available in the test set.

5.2 Analyzing template consistency

Evaluating the language subtask is a challenge (further discussions can be seen in Section 6.2.1, where we discuss limitations) as we cannot establish a ground truth for our templates. We set up a two-stage verification process; a manual code check and an LLM check.

5.2.1 Evaluation criteria

Our code check system leverages the spaCy framework to create structured templates through parts of speech tagging. At its core, the system examines the grammatical relationships between words to identify three key components: First, it recognizes Actions by detecting words tagged as "VERB" in the sentence. Second, it pinpoints Objects by looking for direct object dependencies ("dobj"), which typically represent the things being acted upon. Third, it determines Locations by analyzing prepositional phrases ("pobj"), often indicating where actions occur. We then compare the spaCY template to our template. The spaCy template will often not be perfect, which is why we didn't use it as our method, but it will help act as a guide and will often get the simple templates correct. Which is why we also implemented the LLM check.

LLMs are emerging as promising tools for evaluating text quality in natural language generation, potentially offering an alternative to traditional human evaluation methods [Chiang and Lee 2023; Gao *et al.* 2025]. Research indicates that LLMs can produce assessments that align well with expert human judgments. We use this as this is the next best thing for human evaluators, as we have no ground truths. We prompt GPT-4o-mini using the following;

```
Does the following template correctly identify the Action, Object & Loca-
tion  ("From"/"into/"to" can be used) of the sentence: "{sentence}"

{template}

Answer only yes or no, If no explain what is wrong.
```

When both validation checks yield matching results, we accept that as the definitive answer. In cases of disagreement, we pass it to human evaluation. This dual-check approach significantly reduces the manual review workload for this subtask.

Task	Accuracy (%)
Heat and Place	53
Pick and Place	57
Cool and Place	75
Clean and Place	63
Pick Two and Place	0
Examine in Light	50
Pick and Place with Movable recep	0

Table 5.7: Llama-2-7b-chat's accuracy across the seven different ALFRED tasks. The results here show that our pipeline is not suited for multi-object tasks. Also, the tasks may seem similar but the varying methods of description cause variance in the templating ability

5.2.2 LLM Analysis

During our testing, we evaluated Llama-3 models but found they performed significantly worse than their Llama-2 counterparts. Therefore, we present results for Llama-2-7b-chat instead. The results shown in Table 5.6 are collected from the 1 529 unseen demonstrations available in the test set. The relatively lower performance of Gemma-2-2b-it is expected given its substantially smaller size compared to the other models. We included this smaller model in our analysis to determine whether it could still successfully complete the task despite its reduced parameter count. Table 5.7 breakdowns Llama-2-7b-chat's accuracy across the seven different ALFRED tasks.

The results in Table 5.7 demonstrate that specific task instructions necessitate more flexible templates, especially for "Pick Two and Place" and "Pick and Place with Movable receptacle" tasks. This reveals constraints in our approach across multiple dimensions: our prompting methodology, template design, and the inherent limitations of using HDDL as a constraint language.

The primary challenge stems from our LLM prompting strategy and template framework being optimized for single-object manipulation, whereas these complex tasks demand simultaneous handling of multiple objects. This fundamental mismatch extends to our HDDL implementation, which lacks robust mechanisms for distinguishing between multiple instances of identical objects (such as differentiating between individual apples when multiple are present). These limitations underscore the inherent difficulty of creating generalizable solutions that can adapt across diverse task types.

5.3 Evaluating the accuracy of semantic mapping

While we directly adopt the vision implementation from Min *et al.* [2021], we need to assess its effectiveness since it plays a crucial role in overall performance. We evaluate the vision component in two ways, we checked whether it could:

1. Accurately identify the required objects and their locations for each task.

Figure 5.1: Scenes arranged in top-to-bottom rows: kitchens, living rooms, bedrooms, and bathrooms. Object placements are randomized according to available surface areas and class-specific constraints. (sourced from [Shridhar *et al.* 2020]).

2. Correctly match the object positions in our generated map to those specified in the ALFRED trajectory files.

These are not the most exhaustive evaluations, but they give us enough information to determine whether the point of failure came from the vision subtask.

5.3.1 Evaluation criteria

The first part of our analysis is simple, we parse through the generated problem file looking for mention of the required object and receptacle. If either is missing, we can say that the Vision subtask failed.

To verify the map's accuracy, we compare the coordinates of objects in our generated map with those in the ALFRED trajectory file. Specifically, we use the object centre coordinates for this comparison. We calculate the Euclidean distance between these points, and if the distance falls below a predetermined threshold, we consider the objects to be located in the same spatial position. If most objects are in the correct place, we declare that the vision subtasks succeeded.

Reasons	Percentage errors
Objects not located	80
Object in incorrect positions	20

Table 5.8: Breakdown of the distribution of the cause of the mapping errors. Collected from the 1 529 unseen demonstrations available in the test set. The table shows that the semantic mapping module often is incapable of finding the main object or receptacle needed for the task.

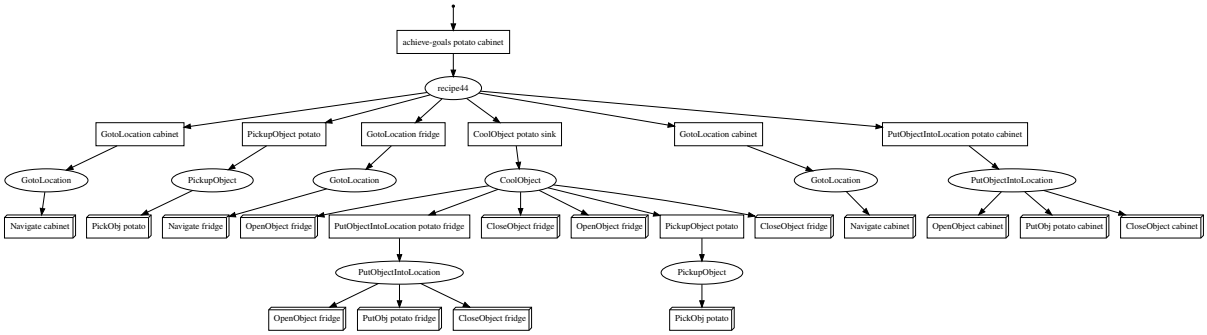

Figure 5.2: Graph visualization of an HTN plan using IPC output format. The example showcases how PANDA breakdowns of the task `achieve-goals Potato Cabinet` using recipe 44 from the cool_and_place domain file.

5.3.2 Analysis

The accuracy of the semantic mapping module is **30%**. The mapping process is a large hindrance to the performance of the entire pipeline. The process often struggles to find the required objects in the scene (see Table 5.8). There are many reasons for this;

- Objects are not very large, so can be easily missed.
- Objects are not always in the open, especially in the kitchen setting, many objects are located inside drawers, cupboards, etc. This makes it extremely difficult to locate.
- The scenes are very diverse (as can be seen in Figure 5.1) so it is very difficult to generalize to all the scenes.

There is much room for improvement in this area, but it is not the focus of our research. Much future work can look to improving this aspect of the problem.

5.4 Verifying plan integrity

Like the Language Subtask, the Planning Subtask lacks ground truths, so determining whether a plan is correct is challenging. Again, we look toward a mixture of LLM and algorithmic code checks.

5.4.1 Evaluation criteria

Plan accuracy in this work refers to whether a generated HDDL plan successfully satisfies the high-level task goal defined by the natural language instruction. Due to the lack of a standard symbolic ground-truth plan representation for each task, accuracy is computed via a heuristic evaluation pipeline, described as follows:

- The generated plan is converted into a natural language summary using a fixed template.
- This summary, along with the original instruction, is passed to an LLM (GPT-4o-mini) with a yes/no prompt asking if the plan achieves the goal.
- The LLM's binary judgment is used as the accuracy signal.

This method relies on semantic interpretation rather than symbolic matching but introduces potential noise or inconsistency from the LLM. The limitations and problems with this are further discussed in Section 6.2.2. We start by prompting GPT-4o-mini with;

```
Does the following plan:
{plan}
accomplish the high-level instructions:
{steps}
Only say whether they accomplish the instructions and how many
instructions were accomplished.
```

We prompt the model with multiple examples of plan-instruction pairs from the same task, then give this follow-up prompt;

```
From all the plans what were the common mistakes that occurred?
```

Through repeated use of this prompt structure for various tasks, we can identify patterns in plan errors, which we use to guide our algorithmic checks.

5.4.2 Analysis

The Accuracy of the PANDA planner is **42%**. The performance of the planner leaves a lot to be desired but it establishes proof of concept, as this is the first step in a new direction that bridges classical methods with modern AI. A successful plan can be seen in Figure 5.2.

The complexity of the ALFRED environment leaves many challenges when it comes to planning. Table 5.9 shows the accuracy of the planner on the different tasks. The poor results on the "Pick Two and Place" and "Pick and Place with Movable recep" tasks stem from the inability of the language subtask to create correct templates for multi-object problems. The PANDA framework also faces scalability challenges when handling domain files containing thousands of methods. When combined with large problem files (representing complex scenes), the planner's search time increases exponentially, often preventing it from finding solutions within time constraints. This significantly impacts performance.

Task	Accuracy
Heat and Place	51
Pick and Place	76
Cool and Place	54
Clean and Place	62
Pick Two and Place	0
Examine in Light	48
Pick and Place with Movable recep	0

Table 5.9: The accuracy generated plan across the seven different ALFRED tasks distributed in the 1 529 unseen demonstrations

5.5 ALFRED benchmark performance analysis

The above results test each component in our Runtime pipeline individually. The ALFRED benchmark is where we test our complete pipeline against other baselines.

5.5.1 Metrics

ALFRED has metrics to evaluate full task and task goal-condition completion. Those are:

- **Task Success**. Determined by whether the object positions and state changes accurately align with the task's desired goal conditions at the action sequence's conclusion. If the correspondence is correct, Task Success is assigned a value of 1; otherwise, it is assigned a value of 0.
- **Goal-Condition Success**. The completion ratio of a task at the end of an episode is calculated by dividing the number of goal-conditions successfully achieved by the total number of goal-conditions required to complete the task.

5.5.2 Baselines

We will be comparing our method to the following baselines;

- FILM [Min *et al.* 2021] presented a modular approach that utilizes structured representations to achieve a natural language goal. The method consists of two key components: (1) constructing a semantic map of the scene, and (2) conducting exploration using a semantic search policy. Together, these components enable the attainment of the desired goal expressed in natural language.
- LGS-RPA [Murray and Cakmak 2022] presented a novel approach that leverages visible landmarks to enhance the efficiency of environment exploration in pursuit of objects specified in natural language instructions. Moreover, we introduce the utilization of a pose adjustment policy during manipulation planning, enabling the robot to overcome challenges arising from noisy visual observations and improve overall performance.

	Test Seen		Test Unseen	
Method	TS	GC	TS	GC
FILM [Min *et al.* 2021]	25.77	36.15	27.8	38.52
LGS-RPA [Murray and Cakmak 2022]	33.01	41.71	35.41	45.24
Prompter [Inoue and Ohashi 2022]	49.38	55.90	45.72	58.76
EPO [Zhao *et al.* 2024][2]	**64.79**	**72.30**	**62.35**	**67.52**
Ours	20.23	30.15	20.04	30.48

Table 5.10: Results on the test splits of the ALFRED benchmark. It can be seen that the EPO method is the best performing with our approach being the worst.

- Prompter [Inoue and Ohashi 2022] introduced FILM++ as an extension to the existing work FILM. The modifications introduced in FILM++ offer enhanced capabilities without the need for additional data. They also introduced a replacement for FILM++'s semantic search module, Prompter, a language model prompting approach.
- EPO [Zhao *et al.* 2024] proposes a hierarchical framework that uses separate LLMs for subgoal prediction and action generation to break complex tasks into manageable steps. Their environment Preference Optimization (EPO) method generates training signals from multimodal environment feedback, creating preference signals to train LLM agents without requiring annotated datasets.

5.5.3 Evaluations

The results (Table 5.10) on the ALFRED benchmark may not be competitive with the top models, this is due to many problems such as mapping and low-level planner from FILM. The nature of the pipeline means that if a single component fails, the run will be a failure. As we introduced many novel components there is still much room for improvement on all of them.

5.6 Conclusion

After reviewing our results, we fall short of the state-of-the-art performance on the ALFRED benchmark. While we propose a novel pipeline, its complexity has become a significant limitation. By introducing multiple new components, we've created numerous potential failure points throughout the system.

The language subtask suffers as the templating procedure does not fully support multi-object tasks. The general performance is also not very high, a simple remedy to this is to use larger LLMs such as ChatGPT-4 [Achiam *et al.* 2023] or the 70b-parameter Llama-2-chat model [Touvron *et al.* 2023], as these would produce more consistent results. However, running these models locally is impossible (ChatGPT-4 is only accessible through API) or is at least too expensive (in the case of running any large model of 70b parameters or more). Only having API access creates its own set of issues such

as latency, and connectivity issues and when dealing with robots in the home there is ethical consideration as well.

The process of converting to HDDL and searching for a plan using PANDA is still cumbersome. Due to the inconsistency of LLMs, there is still much processing that must be done for the translation to occur. There are also scale problems when dealing with PANDA, as it was not designed to have domain files with thousands of methods. This leads to issues when the problem file also gets large (the scene is very busy) where the search time of the planner scales exponentially, which results in the planner not being able to find a plan within the allotted time. This is a major hindrance to performance.

The combinatorial effect of these vulnerabilities means that achieving a completely successful run on any ALFRED task requires all components to function correctly simultaneously – a challenging requirement that our current implementation struggles to meet. As this is the first attempt at the pipeline, there is room for much future work to improve the results.

Chapter 6

Conclusion and Future Directions

While this study provides valuable insights into the effectiveness of combining HTNs and LLMs to create generalized methods, it is important to acknowledge certain limitations that contextualize the findings. These limitations not only frame the scope of the current research but also highlight promising directions for future investigation.

6.1 Problems with Natural Language Understanding

Recent research highlights challenges in evaluating LLMs for **N**atural **L**anguage **U**nderstanding (NLU) tasks. LLMs often rely on shortcuts, creating an illusion of enhanced performance while lacking generalizability [Bihani and Rayz 2025]. Fine-tuning open-source LLMs for high-precision NLU in goal-driven dialog systems has shown promise, particularly for small-medium enterprises seeking cost-effective solutions [Padró and Saurí 2024]. However, challenges remain in ensuring LLMs adhere to strict format and consistency requirements for real-world applications. These studies emphasize the need for more robust evaluation methods and the development of LLMs that can generalize effectively across languages and domains while maintaining factual accuracy and reasoning capabilities.

A recurring example of this can be seen in the word “move” which creates ambiguity when templating the instructions, as demonstrated in our testing. Consider these two types of sentences:

1. “Move from the countertop to the microwave”
2. “Move the apple from the countertop to the microwave”

While the first sentence is a navigation task, the second is an action task. Despite this distinction, the LLM consistently interprets both as navigation tasks, failing to understand when an object needs to be manipulated. It overlooks the presence of the object (like “the apple”) and defaults to treating all “move” commands as instructions for navigation.

This serves as an effective demonstration that while LLMs have made tremendous progress in understanding and generating human language, they are not foolproof. Context parsing, especially in scenarios with slight syntactic variations, remains a complex challenge in NLP. Such examples highlight the need for future work to refine NLU techniques that can more accurately capture the semantic nuances of human communication.

6.2 Difficulty Measuring Correctness

Without ground truth data, there's no clear standard to measure performance against. This makes it difficult to definitively say whether predictions or outputs are correct or determine the actual error rate.

The structure of our pipeline created gaps where ground truth data was unavailable. This forced us to rely on more subjective evaluation methods, particularly when assessing the quality of templates created by LLMs and plans generated by PANDA.

6.2.1 LLM templates

"What does it mean for a template to be correct?" is a question that does not have a single objective answer or metric/baseline to validate against. The challenge of defining "correctness" for templates stems from several factors:

The complexity of some instructions allows for multiple valid approaches. For example, "Take a potato, heat it, place it on the table", is phrased in a way where the template {"Action": "Heat", "Object": "Potato", "On": "Table"} and {"Action": "Place", "Object": "heated_Potato", "On": "Table"} may seem correct. It can be argued that in this domain the "Heat" action implies the high-level task of heat_and_place, which means heat the object in the microwave and then place it elsewhere. There could also be arguments that neither is an accurate representation of the sentence as it does not contain all the actions.

Edge cases & variability is an issue. Sentences with different word orders, omitted parts, or synonyms (e.g., "Pick soap up from tub, wash soap in sink, put soap on box in front of sink.") may not be captured correctly. Algorithmically it is difficult to detect missing information — if a template is incomplete or missing a required element, this simple check may not flag the issue.

The loss of information is another point of contention. If the sentence is "Pick up the mug from the sink and put it on the shelf", the template of {"Action": "put", "Object": "Mug", "On": "Shelf} loses the initial position of the mug. Though the template still accurately describes the action that needs to take place, but relies on the planning procedure to account for the correct mug, using the context of the objects in the scene.

This ambiguity means it is difficult to establish a single "correct" template for each task. Although we established evaluation criteria using LLMs, code and human evaluation it is still not a replacement for ground truth labels, as our criteria will contain unintended biases.

Our evaluation of the Language Subtasks acts as more of an analysis, providing insight into where the LLM goes wrong instead of a strict assessment of the modules' performance.

Another area of our work that faced a similar problem is evaluating the plans produced by PANDA, as, like the templates, there is no ground truth available to validate against.

6.2.2 Generated Plans

In the ALFRED environment, evaluating plan correctness is challenging due to the absence of clear ground truth criteria. While one might define a correct plan as one that successfully completes an episode, the complexity of executing plans makes this definition problematic. In this context, a plan could be theoretically correct but still fail during low-level execution, rendering the episode unsuccessful despite having a structurally sound plan.

So we find ourselves asking "What does it mean for a plan to be correct?", This answer is also determined by several factors.

Goal Achievement means a correct plan must effectively address the intended objective. It should provide a clear pathway to accomplish the specific goal or solve the problem at hand. Each step in the plan must contribute directly or indirectly to the desired outcome through clear cause-and-effect relationships. When doing algorithmic checks, it is very difficult to check for every possible subgoal in a task.

The plan must be realistically implementable given the available resources, constraints, and circumstances. To determine the feasibility of a plan algorithmically is very challenging as you have to take into account the environment, all the objects in the environment and whether the interactions are possible given that information.

A correct plan follows a coherent and logical sequence of steps. The plan should be comprehensive, covering all essential steps required to achieve the goal. This means anticipating potential challenges and including strategies to address them.

The effectiveness of a plan can only be meaningfully assessed within its specific context – taking into account the intended objectives, circumstances, and limitations involved. This makes it challenging to develop a comprehensive evaluation framework that could account for all these nuanced factors. As such, our evaluation methods are better suited for identifying common mistakes than definitively determining whether a plan is "correct" or not.

6.3 PANDA scaling issues

The PANDA planner offers fast and effective planning capabilities for standard HTN challenges. However, our system's unique architecture often exceeds conventional HTN problem boundaries, particularly in terms of domain and problem file dimensions and overall computational scale.

Given the combinatorial nature of PANDA's search space, the planner can encounter computational challenges that may result in prolonged search times. Our experimental findings reveal that in some scenarios, the search process can extend to over an hour, which contrasts the millisecond-level performance for most typical runs. In extreme cases, we encountered a halting problem where the planner continued executing for more than an hour and a half, forcing us to terminate the process due to uncertainty about whether it would ever conclude.

Classical methodologies struggle to address modern challenges effectively. One key limitation emerges when integrating classical approaches with LLMs: classical methods rely on precise, formal languages that require complete accuracy, whereas LLMs, despite their impressive consistency, cannot guarantee perfect outputs.

6.4 LLM consistency

LLMs show exceptional abilities in in producing code and structured formats like JSON, though their output isn't flawless. Recent research by Lee *et al.* [2024] analyzed consistency patterns in LLM evaluators, finding that even advanced commercial models can struggle to maintain consistent output. The errors can be minor, such as incorrectly placed or missing commas in JSON, or more significant, such as improper bracket matching.

The output of an LLM is also not the same every time, these variations in LLM outputs pose particular challenges when working with formal domain-specific languages like HDDL. This non-deterministic nature means that even if an LLM generates a valid output once, there's no guarantee it will do so in subsequent attempts with the same prompt.

Future research directions could explore integrating methods similar to those proposed by Yang *et al.* [2025] and Raj *et al.* [2023], which aim to enhance output consistency, as well as investigate novel approaches to this challenge.

6.5 Problem generating entire HDDL domain files

While we can generate most parts of the HDDL domain files (see Section 4.7.3). There are still some parts which we have to hard code in, these being the predicates along with the task and method definitions of the simple methods.

The predicates are built upon prior knowledge of the ALFRED domain and aren't explicitly defined since they're only required for HDDL, which isn't ALFRED's primary focus as it's designed for broader applications. This makes the algorithmic generation of these predicates particularly challenging. Since predicates are derived from action effects, future research could explore methods to infer predicates from the provided demonstrations.

The simple methods aren't described in the trajectory file since they represent basic actions within the ALFRED domain. Consequently, we lack a way to describe them in

JSON format and, by extension, in HDDL. This limitation forces us to hard-code these methods rather than generate them algorithmically. Future research could explore potential approaches to address this challenge.

6.6 Contributions

The ability of robots to adapt and be general agents is still a grand challenge in the world of robotics [Brohan *et al.* 2023]. There are many difficulties in creating a general-purpose robot.

To attempt to overcome these difficulties, we propose a pipeline to learn generalized HTN methods. This pipeline included three modules; Natural Language to Formal Language, Method Construction and Clustering and Paramertising. Our original hypothesis states that merging and parameterising all similar methods would create generalised methods that can accomplish unseen tasks in seen and unseen environments.

Based on the results presented in Chapter 5, we conclude that the hypothesis proposed in Section 4.2 is partially supported.

Specifically, the hypothesis stated that: “Hierarchical task methods abstracted from demonstration trajectories and structured via large language model prompting and clustering can generalize across unseen tasks and be used to generate effective plans.”

The system succeeded in producing generalized methods that were reusable across similar tasks, and it demonstrated the ability to generate valid HDDL plans from unseen instructions. However, the overall execution success was limited, particularly in tasks with complex object interactions or requiring nuanced semantic understanding (e.g., ”Pick Two and Place”).

Therefore, while the core mechanism of abstracting and generalizing methods proved viable, the performance was hindered by limitations in semantic grounding, plan evaluation accuracy, and environmental assumptions. These factors constrained the pipeline’s practical effectiveness and point to the need for a more robust integration of perception and planning in future work.

References

[Achiam *et al.* 2023] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[Ada *et al.* 2019] Suzan Ece Ada, Emre Ugur, and H Levent Akin. Generalization in transfer learning. *arXiv preprint arXiv:1909.01331*, 2019.

[Aeronautiques *et al.* 1998] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[Agarwal *et al.* 2021] Rishabh Agarwal, Marlos C Machado, Pablo Samuel Castro, and Marc G Bellemare. Contrastive behavioral similarity embeddings for generalization in reinforcement learning. *arXiv preprint arXiv:2101.05265*, 2021.

[Alford *et al.* 2015] Ron Alford, Pascal Bercher, and David Aha. Tight bounds for htn planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 7–15, 2015.

[Bae *et al.* 2020] Juhee Bae, Tove Helldin, Maria Riveiro, Sławomir Nowaczyk, Mohamed-Rafik Bouguelia, and Göran Falkman. Interactive clustering: A comprehensive review. *ACM Computing Surveys (CSUR)*, 53(1):1–39, 2020.

[Behnke *et al.* 2018a] Gregor Behnke, Daniel Höller, and Susanne Biundo. totsat-totally-ordered hierarchical planning through sat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[Behnke *et al.* 2018b] Gregor Behnke, Daniel Höller, and Susanne Biundo. Tracking branches in trees-a propositional encoding for solving partially-ordered htn planning problems. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 73–80. IEEE, 2018.

[Behnke *et al.* 2019] Gregor Behnke, Daniel Höller, and Susanne Biundo. Finding optimal solutions in htn planning-a sat-based approach. In *IJCAI*, pages 5500–5508, 2019.

[Behnke *et al.* 2020] Gregor Behnke, Daniel Höller, Alexander Schmid, Pascal Bercher, and Susanne Biundo. On succinct groundings of htn planning problems. In *Pro-*

ceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 9775–9784, 2020.

[Bercher *et al.* 2014a] Pascal Bercher, Susanne Biundo, Thomas Geier, Thilo Hoernle, Florian Nothdurft, Felix Richter, and Bernd Schattenberg. Plan, repair, execute, explain—how planning helps to assemble your home theater. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, pages 386–394, 2014.

[Bercher *et al.* 2014b] Pascal Bercher, Shawn Keen, and Susanne Biundo. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the International Symposium on Combinatorial Search*, volume 5, pages 35–43, 2014.

[Bercher *et al.* 2017] Pascal Bercher, Gregor Behnke, Daniel Höller, and Susanne Biundo. An admissible htn planning heuristic. In *IJCAI*, pages 480–488, 2017.

[Bihani and Rayz 2025] Geetanjali Bihani and Julia Rayz. Learning shortcuts: On the misleading promise of nlu in language models. In *Artificial Intelligence for Design and Process Science*, pages 147–158. Springer, 2025.

[Blukis *et al.* 2022] Valts Blukis, Chris Paxton, Dieter Fox, Animesh Garg, and Yoav Artzi. A persistent spatial semantic representation for high-level natural language instruction execution. In *Conference on Robot Learning*, pages 706–717. PMLR, 2022.

[Brohan *et al.* 2023] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, pages 287–318. PMLR, 2023.

[Brown *et al.* 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[Cao 2024] Hongliu Cao. Recent advances in text embedding: A comprehensive review of top-performing methods on the mteb benchmark. *arXiv preprint arXiv:2406.01607*, 2024.

[Carvallo *et al.* 2020] Andres Carvallo, Denis Parra, Hans Lobel, and Alvaro Soto. Automatic document screening of medical literature using word and text embeddings in an active learning setting. *Scientometrics*, 125:3047–3084, 2020.

[Chaplot *et al.* 2020] Devendra Singh Chaplot, Dhiraj Prakashchand Gandhi, Abhinav Gupta, and Russ R Salakhutdinov. Object goal navigation using goal-oriented se-

mantic exploration. *Advances in Neural Information Processing Systems*, 33:4247–4258, 2020.

[Chiang and Lee 2023] Cheng-Han Chiang and Hung-yi Lee. A closer look into automatic evaluation using large language models. *arXiv preprint arXiv:2310.05657*, 2023.

[Chowdhery *et al.* 2023] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[Chung *et al.* 2024] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

[Cresswell and Gregory 2011] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *Proceedings of the international conference on automated planning and scheduling*, volume 21, pages 42–49, 2011.

[Dehghani *et al.* 2023] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pages 7480–7512. PMLR, 2023.

[Devlin *et al.* 2017] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[Devlin *et al.* 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[Driess *et al.* 2023] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[Dubey *et al.* 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[Erol *et al.* 1994] Kutluhan Erol, Jim Hendler, and Dana Nau. Htn planning: Complexity and expressivity. 1994.

[Ethayarajh 2019] Kawin Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512*, 2019.

[Fang *et al.* 2020] Hongchao Fang, Sicheng Wang, Meng Zhou, Jiayuan Ding, and Pengtao Xie. Cert: Contrastive self-supervised learning for language understanding. *arXiv preprint arXiv:2005.12766*, 2020.

[Gao *et al.* 2021] Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

[Gao *et al.* 2025] Mingqi Gao, Xinyu Hu, Xunjian Yin, Jie Ruan, Xiao Pu, and Xiaojun Wan. Llm-based nlg evaluation: Current status and challenges. *Computational Linguistics*, pages 1–28, 2025.

[Gates 2007] Bill Gates. A robot in every home. *Scientific American*, 296(1):58–65, 2007.

[Gogoll *et al.* 2020] Dario Gogoll, Philipp Lottes, Jan Weyler, Nik Petrinic, and Cyrill Stachniss. Unsupervised domain adaptation for transferring plant classification systems to new field environments, crops, and robots. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2636–2642. IEEE, 2020.

[Graves *et al.* 2014] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[Harris *et al.* 2024] Nicholas Harris, Anand Butani, and Syed Hashmy. Enhancing embedding performance through large language model-based text enrichment and rewriting. *arXiv preprint arXiv:2404.12283*, 2024.

[Hayashi *et al.* 2005] Hisashi Hayashi, Kenta Cho, and Akihiko Ohsuga. A new htn planning framework for agents in dynamic environments. In *Computational Logic in Multi-Agent Systems: 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6-7, 2004, Revised Selected and Invited Papers 4*, pages 108–133. Springer, 2005.

[He *et al.* 2015] Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep reinforcement learning with a natural language action space. *arXiv preprint arXiv:1511.04636*, 2015.

[He *et al.* 2017] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[Höller *et al.* 2018] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. A generic method to guide htn progression search with classical heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 114–122, 2018.

[Höller *et al.* 2019] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. On guiding search in htn planning with classical planning heuristics. In *IJCAI*, pages 6171–6175, 2019.

[Höller *et al.* 2020a] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. Hddl: An extension to pddl for expressing hierarchical planning problems. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 9883–9891, 2020.

[Höller *et al.* 2020b] Daniel Höller, Pascal Bercher, and Gregor Behnke. Delete- and ordering-relaxation heuristics for htn planning. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4076–4083. IJCAI organization, 2020.

[Höller *et al.* 2021] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. The panda framework for hierarchical planning. *KI-Künstliche Intelligenz*, 35(3):391–396, 2021.

[Huang *et al.* 2022] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[Inoue and Ohashi 2022] Yuki Inoue and Hiroki Ohashi. Prompter: Utilizing large language model prompting for a data efficient embodied instruction following. *arXiv preprint arXiv:2211.03267*, 2022.

[Jiang *et al.* 2023] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[Jin and Zhuo 2022] Kebing Jin and Hankz Hankui Zhuo. Integrating ai planning with natural language processing: a combination of explicit and tacit knowledge. *ACM Transactions on Intelligent Systems and Technology*, 2022.

[Kelly *et al.* 2008] John-Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 60–65, 2008.

[Kidd and Breazeal 2008] Cory D Kidd and Cynthia Breazeal. Robots at home: Understanding long-term human-robot interaction. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3230–3235. IEEE, 2008.

[Kojima *et al.* 2022] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[Kolve *et al.* 2017] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu,

et al. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017.

[Lallement *et al.* 2014] Raphaël Lallement, Lavindra De Silva, and Rachid Alami. Hatp: An htn planner for robotics. *arXiv preprint arXiv:1405.5345*, 2014.

[Lee *et al.* 2024] Noah Lee, Jiwoo Hong, and James Thorne. Evaluating the consistency of llm evaluators. *arXiv preprint arXiv:2412.00543*, 2024.

[Li *et al.* 2018] Da Li, Yongxin Yang, Yi-Zhe Song, and Timothy Hospedales. Learning to generalize: Meta-learning for domain generalization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[Li *et al.* 2020] Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. On the sentence embeddings from pre-trained language models. *arXiv preprint arXiv:2011.05864*, 2020.

[Lindsay *et al.* 2017] Alan Lindsay, Jonathon Read, Joao Ferreira, Thomas Hayton, Julie Porteous, and Peter Gregory. Framer: Planning models from natural language action descriptions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, pages 434–442, 2017.

[Liu *et al.* 2023] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

[McDermott *et al.* 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *Pddl–the planning domain definition language–version 1.2*. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational . . . , 1998.

[Meng *et al.* 2024] Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Sfrembedding-mistral: enhance text retrieval with transfer learning. *Salesforce AI Research Blog*, 3:6, 2024.

[Min *et al.* 2021] So Yeon Min, Devendra Singh Chaplot, Pradeep Ravikumar, Yonatan Bisk, and Ruslan Salakhutdinov. Film: Following instructions in language with modular methods. *arXiv preprint arXiv:2110.07342*, 2021.

[Mohseni-Kabir *et al.* 2014] Anahita Mohseni-Kabir, Sonia Chernova, and Charles Rich. Collaborative learning of hierarchical task networks from demonstration and instruction. In *2014 AAAI Fall Symposium Series*, 2014.

[Muennighoff *et al.* 2023] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. *MTEB: Massive Text Embedding Benchmark*, 2023.

[Murray and Cakmak 2022] Michael Murray and Maya Cakmak. Following natural language instructions for household tasks with landmark guided search and reinforced pose adjustment. *IEEE Robotics and Automation Letters*, 7(3):6870–6877, 2022.

[Nau *et al.* 2003] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of artificial intelligence research*, 20:379–404, 2003.

[Nie *et al.* 2024] Zhijie Nie, Zhangchi Feng, Mingxin Li, Cunwang Zhang, Yanzhao Zhang, Dingkun Long, and Richong Zhang. When text embedding meets large language model: A comprehensive survey. *arXiv preprint arXiv:2412.09165*, 2024.

[Oord *et al.* 2018] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[Ouyang *et al.* 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[Padmakumar *et al.* 2022] Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. Teach: Task-driven embodied agents that chat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 2017–2025, 2022.

[Padró and Saurí 2024] Lluís Padró and Roser Saurí. Fine-tuning open access llms for high-precision nlu in goal-driven dialog systems. In *Proceedings of the Second International Workshop Towards Digital Language Equality (TDLE): Focusing on Sustainability@ LREC-COLING 2024*, pages 33–42, 2024.

[Pashevich *et al.* 2021] Alexander Pashevich, Cordelia Schmid, and Chen Sun. Episodic transformer for vision-and-language navigation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15942–15952, 2021.

[Patil and Gudivada 2024] Rajvardhan Patil and Venkat Gudivada. A review of current trends, techniques, and challenges in large language models (llms). *Applied Sciences*, 14(5):2074, 2024.

[Pennington *et al.* 2014] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[Petukhova *et al.* 2025] Alina Petukhova, João P Matos-Carvalho, and Nuno Fachada. Text clustering with large language model embeddings. *International Journal of Cognitive Computing in Engineering*, 6:100–108, 2025.

[Raj *et al.* 2023] Harsh Raj, Vipul Gupta, Domenic Rosati, and Subhabrata Majumdar. Semantic consistency for assuring reliability of large language models. *arXiv preprint arXiv:2308.09138*, 2023.

[Ren 2024] Mengchao Ren. Advancements and applications of large language models in natural language processing: A comprehensive review. *Applied and Computational Engineering*, 97:55–63, 2024.

[Savva *et al.* 2019] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9339–9347, 2019.

[Schaal 1996] Stefan Schaal. Learning from demonstration. *Advances in neural information processing systems*, 9, 1996.

[Sethian 1996] James A Sethian. A fast marching level set method for monotonically advancing fronts. *proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[Shridhar *et al.* 2020] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749, 2020.

[Silva *et al.* 2021] Andrew Silva, Nina Moorman, William Silva, Zulfiqar Zaidi, Nakul Gopalan, and Matthew Gombolay. Lancon-learn: Learning with language to enable generalization in multi-task manipulation. *IEEE Robotics and Automation Letters*, 7(2):1635–1642, 2021.

[Stepputtis *et al.* 2020] Simon Stepputtis, Joseph Campbell, Mariano Phielipp, Stefan Lee, Chitta Baral, and Heni Ben Amor. Language-conditioned imitation learning for robot manipulation tasks. *Advances in Neural Information Processing Systems*, 33:13139–13150, 2020.

[Suddrey *et al.* 2016] Gavin Suddrey, Christopher Lehnert, Markus Eich, Frederic Maire, and Jonathan Roberts. Teaching robots generalizable hierarchical tasks through natural language instruction. *IEEE Robotics and Automation Letters*, 2(1):201–208, 2016.

[Suglia *et al.* 2021] Alessandro Suglia, Qiaozi Gao, Jesse Thomason, Govind Thattai, and Gaurav Sukhatme. Embodied bert: A transformer model for embodied, language-guided visual task completion. *arXiv preprint arXiv:2108.04927*, 2021.

[Thakur *et al.* 2021] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. Beir: A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*, 2021.

[Thoppilan *et al.* 2022] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[Tipirneni *et al.* 2024] Sindhu Tipirneni, Ravinarayana Adkathimar, Nurendra Choudhary, Gaurush Hiranandani, Rana Ali Amjad, Vassilis N Ioannidis, Changhe Yuan, and Chandan K Reddy. Context-aware clustering using large language models. *arXiv preprint arXiv:2405.00988*, 2024.

[Touvron *et al.* 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[Truong *et al.* 2021] Joanne Truong, Sonia Chernova, and Dhruv Batra. Bi-directional domain adaptation for sim2real transfer of embodied navigation agents. *IEEE Robotics and Automation Letters*, 6(2):2634–2641, 2021.

[Valmeekam *et al.* 2022] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[Vaswani *et al.* 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Viswanathan *et al.* 2024] Vijay Viswanathan, Kiril Gashteovski, Kiril Gashteovski, Carolin Lawrence, Tongshuang Wu, and Graham Neubig. Large language models enable few-shot clustering. *Transactions of the Association for Computational Linguistics*, 12:321–333, 2024.

[Wang *et al.* 2022] Xingyao Wang, Sha Li, and Heng Ji. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*, 2022.

[Wang *et al.* 2023] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*, 2023.

[Wang *et al.* 2024] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. *Text Embeddings by Weakly-Supervised Contrastive Pre-training*, 2024.

[Xu *et al.* 2018] Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3795–3802. IEEE, 2018.

[Yang *et al.* 2025] Jingyuan Yang, Dapeng Chen, Yajing Sun, Rongjun Li, Zhiyong Feng, and Wei Peng. Enhancing semantic consistency of large language models through model editing: An interpretability-oriented approach. *arXiv preprint arXiv:2501.11041*, 2025.

[Zellers *et al.* 2021] Rowan Zellers, Ari Holtzman, Matthew Peters, Roozbeh Mottaghi, Aniruddha Kembhavi, Ali Farhadi, and Yejin Choi. Piglet: Language grounding through neuro-symbolic interaction in a 3d world. *arXiv preprint arXiv:2106.00188*, 2021.

[Zhang and Chai 2021] Yichi Zhang and Joyce Chai. Hierarchical task learning from language instructions with unified transformers and self-monitoring. *arXiv preprint arXiv:2106.03427*, 2021.

[Zhang *et al.* 2022] Yichi Zhang, Jianing Yang, Jiayi Pan, Shane Storks, Nikhil Devraj, Ziqiao Ma, Keunwoo Peter Yu, Yuwei Bao, and Joyce Chai. Danli: Deliberative agent for following natural language instructions. *arXiv preprint arXiv:2210.12485*, 2022.

[Zhang *et al.* 2023] Yuwei Zhang, Zihan Wang, and Jingbo Shang. Clusterllm: Large language models as a guide for text clustering. *arXiv preprint arXiv:2305.14871*, 2023.

[Zhao *et al.* 2024] Qi Zhao, Haotian Fu, Chen Sun, and George Konidaris. *EPO: Hierarchical LLM Agents with Environment Preference Optimization*, 2024.

Appendices

```
{
    "m_name": "put",
    "m_args": [
        "Apple",
        "Microwave"
    ],
    "subtasks": [
        {
            "m_name": "GotoLocation",
            "m_args": [
                "GarbageCan"
            ],
            "subtasks": [
                {
                    "s_name": "Navigate",
                    "s_args": "GarbageCan"
                }
            ]
        },
        {
            "m_name": "PickupObject",
            "m_args": [
                "Apple"
            ],
            "subtasks": [
                {
                    "s_name": "PickupObject",
                    "s_args": "Apple"
                }
            ]
        }

        .....

        {
            "m_name": "GotoLocation",
            "m_args": [
                "Microwave"
            ],
            "subtasks": [
                {
                    "s_name": "Navigate",
                    "s_args": "Microwave"
                }
            ]
```

```
        },
        {
            "m_name": "PutObject",
            "m_args": [
                "Apple",
                "Microwave"
            ],
            "subtasks": [
                {
                    "s_name": "OpenObject",
                    "s_args": "Microwave"
                },
                {
                    "s_name": "PutObject",
                    "s_args": [
                        "Apple",
                        "Microwave"
                    ]
                },
                {
                    "s_name": "CloseObject",
                    "s_args": "Microwave"
                }
            ]
        }
```

Listing 1: Example method before parameterising

```
{
    "m_name": "put",
    "m_args": [
        "Apple",
        "Microwave"
    ],
    "subtasks": [
        {
            "m_name": "GotoLocation",
            "m_args": [
                "GarbageCan"
            ],
            "subtasks": [
                {
                    "s_name": "Navigate",
                    "s_args": "GarbageCan"
                }
            ]
        },
```

```
        {
            "m_name": "PickupObject",
            "m_args": [
                "obj"
            ],
            "subtasks": [
                {
                    "s_name": "PickupObject",
                    "s_args": "obj"
                }
            ]
        }

        .....

        {
            "m_name": "GotoLocation",
            "m_args": [
                "loc"
            ],
            "subtasks": [
                {
                    "s_name": "Navigate",
                    "s_args": "loc"
                }
            ]
        },
        {
            "m_name": "PutObject",
            "m_args": [
                "obj",
                "loc"
            ],
            "subtasks": [
                {
                    "s_name": "OpenObject",
                    "s_args": "loc"
                },
                {
                    "s_name": "PutObject",
                    "s_args": [
                        "obj",
                        "microwave"
                    ]
                },
                {
```

```
                    "s_name": "CloseObject",
                    "s_args": "loc"
                }
            ]
        }
```

Listing 2: Example method after parameterising

```
"common1": {
        "m_args": [
            "obj",
            "loc"
        ],
        "preconditions": [
            {
                "has": [
                    "fridge"
                ]
            }
        ],
        "subtasks": [
            {
                "m_name": "CoolObject",
                "m_args": "obj",
                "subtasks": [
                    {
                        "s_name": "OpenObject",
                        "s_args": "fridge"
                    },
                    {
                        "s_name": "PutObject",
                        "s_args": [
                            "obj",
                            "fridge"
                        ]
                    },
                    {
                        "s_name": "CloseObject",
                        "s_args": "fridge"
                    },
                    {
                        "s_name": "OpenObject",
                        "s_args": "fridge"
                    },
                    {
                        "s_name": "PickupObject",
                        "s_args": "obj"
```

```
                    },
                    {
                        "s_name": "CloseObject",
                        "s_args": "fridge"
                    }
                ]
            },
            {
                "m_name": "GotoLocation",
                "m_args": "loc",
                "subtasks": [
                    {
                        "s_name": "Navigate",
                        "s_args": "loc"
                    }
                ]
            },
            {
                "m_name": "PutObject",
                "m_args": [
                    "obj",
                    "loc"
                ],
                "subtasks": [
                    {
                        "s_name": "OpenObject",
                        "s_args": "loc"
                    },
                    {
                        "s_name": "PutObject",
                        "s_args": [
                            "obj",
                            "loc"
                        ]
                    },
                    {
                        "s_name": "CloseObject",
                        "s_args": "loc"
                    }
                ]
            }
        ]
    }
```

Listing 3: Example common method