Creating PDDL Models from Javascript using LLMs: Preliminary Results

Kelsey Sikes[1], **Morgan Fine-Morris**[2,3], **Sarath Sreedharan**[1], **Leslie Smith**[3], **Mark Roberts**[3]

[1]Colorado State University, Fort Collins, Colorado, USA
[2]NRC Postdoc, [3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
kelsey.sikes@colostate.edu, morgan.f.fine-morris.ctr@us.navy.mil, sarath.sreedharan@colostate.edu,
leslie.n.smith20.civ@us.navy.mil, mark.c.roberts20.civ@us.navy.mil

Abstract

Systems that act within an environment (e.g., robots, game agents) often have at least two layers of planning: a descriptive task-planning layer that describes *what* to do and an operational acting layer that states *how* to accomplish a task or action. Because they provide platform-specific details, operational models present a rich source for constructing task planning models. While LLMs have been used for generating task models from natural language, less work has examined how to incorporate operational models. We develop a pipeline that produces PDDL task models by combining natural language and operational JavaScript models for a Minecraft agent. We evaluate the pipeline's ability to produce correct actions under varying conditions. Our results show that the pipeline sometimes generates valid actions, although their correctness fluctuates depending on the input and parameters. We provide six points for consideration of future work in this area.

1 Introduction

Planning with an LLM is often infeasible due to the tendency of LLMs to hallucinate, and their general inability to engage in logical thought (Pallagani et al. 2024). The field of Automated Planning already provides many planning algorithms capable of generating logically-sound plans for domains with a variety of characteristics. Unlike LLMs, automated planners require formal planning-domain models to generate correct plans. These models are a major engineering effort, which is usually undertaken by an expert in the target domain. As such, the automation of domain-model authoring is a common area of research.

Robotics systems can utilize plans generated by automated planners to accomplish their goals, but this requires that the programmer provide not just the implementation for the low-level actions, but also the planning model. To reduce the human-effort cost of setting up such a system, we seek to create planning models based on pre-existing operational models for implemented robotic controllers.

Previous work has investigated the possibility of constructing formal planning domain definitions from natural language domain descriptions via LLMs (Oswald et al. 2024; Oates et al. 2024). Our work extend this research to the construction of planning models from operational models (i.e., low-level agent controller code). Specifically, we use a LLM to translate the controller code in the context of provided natural language information about the domain and extrapolate its preconditions and effects into a PDDL action.

We construct a two-stage pipeline for generating actions in PDDL, pictured in Figure 1. It takes as input (1) controller functions corresponding to some action the bot can undertake, (2) a partial domain model comprising a set of allowed predicates and object types, and (3) natural-language domain information pertinent to the action. It outputs PDDL action definitions for each provided controller function. The first stage of the pipeline processes and summarizes controller functions and the second uses these summaries to select the appropriate set of types and predicates from the allowed set and then constructs an action using them.

We present a case-study on the functions implemented for the Minecraft bot used in Voyager (Wang et al. 2023). Specifically, we test our pipeline's ability to generate valid PDDL actions for crafting a craft table, planks, sticks and a wooden sword. To see the effect of different temperature settings on the actions that are made, we modulate this hyperparameter over four experimental runs. We show that our pipeline is capable of producing valid actions and that lower temperature settings predominantly result in the LLM generating correct actions for crafting most Minecraft objects.

2 Related Work

Work on automated action-model creation has mainly focused on extracting models (1) from plan traces, (2) from natural language using more traditional NLP techniques– both instructional and non-instructional text, or (3) from natural language with LLMs.

Automated action-model creation has long been accomplished by analyzing plan traces to extract action definitions, as with the many works consolidated under MACQ (Callanan et al. 2022). Our approach takes natural language inputs instead of plan traces in formal language and is significantly different than those for learning from plan traces, so we will not make an in-depth discussion here.

Other works have extracted action models from natural-language domain knowledge and natural-language instructions using traditional NLP techniques. The PrePost system (Sil and Yates 2011) constructs STRIPS actions using

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

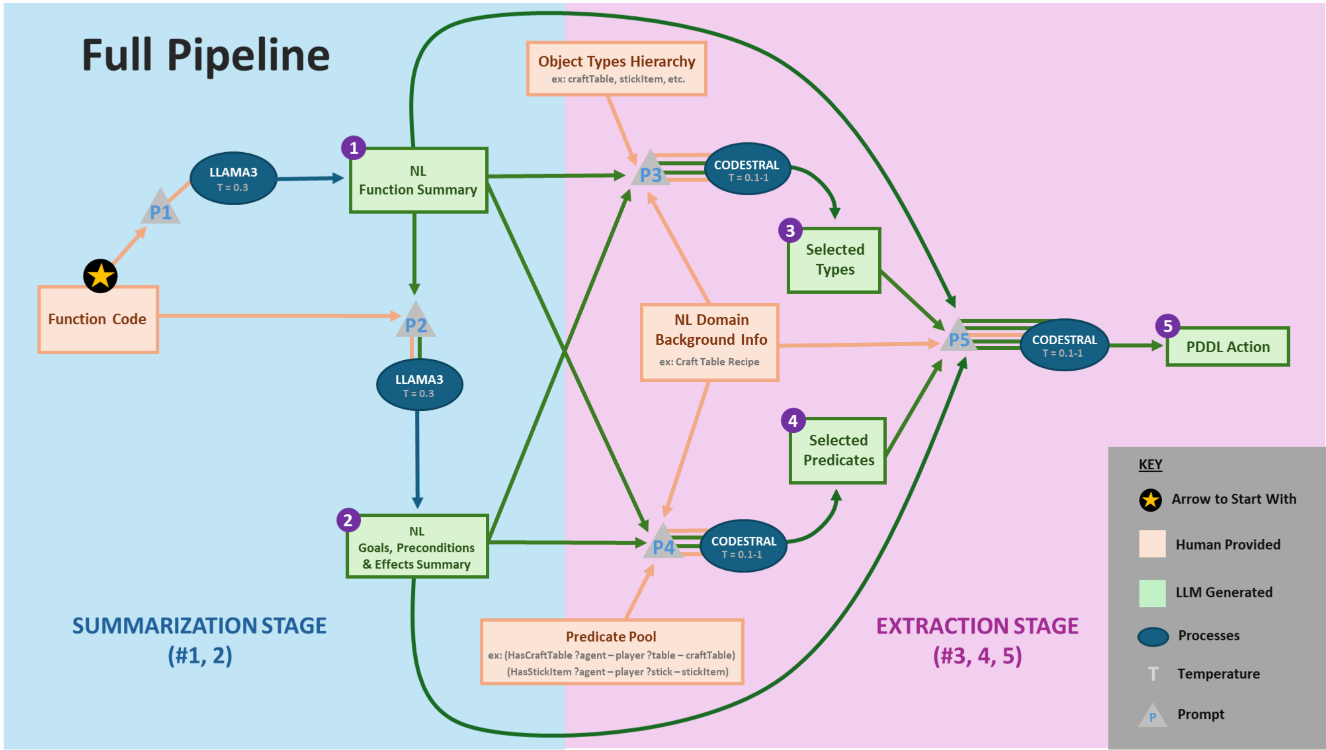


Figure 1: A visualization of the full pipeline process. The process starts at the starred box, and each subsequent intermediate output has a number showing the order in which it is generated. Orange boxes show user-provided inputs and green boxes show LLM outputs. Grey triangles represent prompts into which these components may be fed, and which are then fed into an LLM represented by blue ovals.

a combination of NLP and statistical analysis on webtext. It identifies actions as verbs in gerund-form, and then extracts the preconditions and effects using an SVM trained on a small labeled set of training data to detect if candidate words in a text are likely related to an action word based on proximity. Ge et al. (2012) create an incremental learner for extracting action preconditions, effects, and temporal relations from webtext using NLP, bootstrapping, and knowledge reasoning. Yordanova and Kirste (2016) process instructional text to create PDDL actions. Their technique transforms instruction sequences into timeseries which can be analyzed statistically to identify causal relations between actions and between partial states and actions. They infer preconditions and effects from the causal relations. Huo et al. (2020) used BERT, word2vec, and BiLSTM-CRF to construct PDDL models based on natural language text. They leverage extensive user interaction for refining the model including: classifying objects, removing duplicates, optionally correcting action preconditions and effects. Framer (Lindsay et al. 2017) analyzes plans comprising simple natural language imperative sentences to generate action templates and merges similar templates, and transforms the natural language plans into PDDL-based plans. It provides these to a traditional plan-trace-based operator learner to extract the PDDL model. Jin et al. (2022) use LLMs to extract plan traces from text and then use constraint-based techniques on the extracted plan traces to construct action models.

Several works have leveraged LLMs to construct PDDL-based action models. Oswald et al. (2024) look at creating PDDL actions based on natural-language descriptions and context examples, while restricting the LLM to using only a provided set of PDDL predicates and types. The second stage of our pipeline is similar to this, although we are not using context examples. Oates et al. (2024) use LLMs with in-context learning to generate PDDL actions from CVEs, natural-language technical descriptions of cybersecurity vulnerabilities. They including a human-in-the-loop to inspect the generated operators, refine the prompt, and provide further in-context examples to the LLM when necessary. Smirnov et al. (2024) use LLMs to generate a domain and problem definition in one pass, unlike the previous works which prompt on an action-by-action basis. They use external checking and goal-reachability analysis to identify errors in the domain and problem definitions and an error-correction loop to prompt the LLM to correct the errors. Ding et al. (2023) use LLMs to augment classical planning knowledge with common sense knowledge from the LLM to refine and improve user-provided PDDL action models. Liu et al. (2023) create a planning system which converts natural language domain and problem descriptions into PDDL, constructs a plan using a classical planner, and then coverts the solution back into natural language. Guan et al. (2024)

use LLMs to generate actions, suggest new predicates, and revise previously generated actions to use new predicates. They use both validation tools and human-in-the-loop to prompt LLMs for refinements of the generated models. The CaStL framework (Guo et al., 2024) uses LLMs to generate a PDDL problem description from a PDDL domain description, a scene graph describing the environment, and a natural-language goal description. To the best of our knowledge no existing works use action implementations as their initial input to generate PDDL domain definitions.

Several works use LLMs as planning aids or model-creation aids for the Minecraft domain. However, none of these are concerned with formalizing actions models, but with learning how to perform and combine complex tasks. Voyager is a Minecraft agent which uses LLMs to generate a curriculum for learning and to combine low-level skill function code to make more complex skill functions (Wang et al. 2023). Plan4MC uses LLMs to construct a graph of Minecraft skills, and then searches over the graph to find a path to accomplish complex tasks (Yuan et al. 2023). Ghost in the Minecraft uses LLMs to decompose complex tasks described in natural language into easier subtask also described in natural language and to construct natural language plans, both of which it stores as text for re-use on similar problems (Zhu et al. 2023). We use the low-level JavaScript functions of the Voyager work as one of our inputs.

3 Preliminaries

PDDL is one of the earliest and most-commonly used formalisms in classical planning, which allows the modeling of agent actions as unitary black boxes, abstracting the complexity of performing an action away behind a set of preconditions and effects that encode a transition function. We use it as the formalism to translate our functions.

In PDDL, a planning model is represented using two separate components: the planning domain and the problem. The domain usually contains the types of objects, type relationships, lifted predicates used to describe potential states, and a set of lifted operators (also referred to as action schemas). An operator is further defined using a set of preconditions, add effects, and deletes. Preconditions specify when a possible instance of the operator (i.e., an action) can be executed, and the add and delete effects describe the effect of executing that action. Here each of these components are expressed using a set of lifted predicates. Both the predicates and operators in the domain definitions are usually expressed using variables that stand in for possible objects. The problem, on the other hand, lists the possible objects that can be used, along with the initial state and goal specification.

One obtains the actions that will be executed in the environment by replacing each of the variables with a compatible object from the problem through a process called *grounding*. Grounding also provides us with grounded predicates obtained by replacing the variables in the lifted predicates with objects. Each state possible for a given problem is described using a subset of grounded predicates. A solution for a given planning model is a sequence of actions whose execution in the initial state will result in a state that satisfies the goal specification.

Coming to Minecraft, an example lifted predicate for the fact "object located at a grid square" could be represented by a predicate of the form `object_location(?item - object ?place - gridSquare)`. Here, `object_location` is the predicate name, `?item` and `?place` the variables. Finally, `object` and `place` give the corresponding object types of the variables. Here is an example of a PDDL operator from the domain, where an agent removes an item from its inventory to place it in the world.

```
(:action place
  :parameters (?player ?object ?place)
  :precondition (and
    (inventory_object ?player ?object)
    (not (object_location ?object ?place))
    (agent_location ?player ?place))
  :effect (and
    (not(inventory_object ?player ?object))
    (object_location ?object ?place)))
```

The action describes this process via a state represented by the variables `?player`, `?object`, and `?place`. Its preconditions are that (1) the `?object` is in the agent's inventory, (2) the agent is located where they want to place the object at, and (3) and no other object is currently located in the same place. The add-list effects comprise `(object_location ?object ?place)` which means the `?object` has now been set some `?place`. The delete-list effects are `(not (inventory_object ?player ?object))`, with the "not" keyword indicating that the predicates it prefixes should be removed from the state. The result of this is that the effects negate all predicates in the preconditions.

3.1 Test Domain: Minecraft

Minecraft is a 3D video game which allows a player to control a character to gather resources and craft items. Essentially, resources are combined to create items using recipes. For example, the player gathers resources like `wood` from trees, and then uses these to make `sticks` and `wood planks`, which are then combined to craft an item called a `wooden-sword`. While simple crafting recipes (e.g., planks, sticks, crafting table) can be executed without additional equipment, more complex recipes (e.g., swords) must be crafted on a crafting table, which the player usually must craft themselves and place somewhere in the Minecraft world. Minecraft has been used in a number of works (e.g., (Wang et al. 2023), (Zhu et al. 2023)) and has two attributes that make it an attractive domain for this work. First, many automated bots exist to automate gameplay agents, so there is lots of code to draw from. The Minecraft world is a large and complex environment with varied tasks, so some elements of the game may not be completely supported by existing bots, but many tasks are implemented sufficiently for an initial case study. Second, a set of wiki documents is available in the form of written natural language that we can use as input. For a more detailed description of the domain see the appendix.

4 Pipeline for Generating PDDL Actions

Figure 1 shows the two-stage pipeline for translating action implementations into PDDL operators comprising: (1) the Summarization Stage and (2) the Extraction Stage. In each stage, inputs are injected into prompts for the LLM. Some of these inputs are user-provided and some are generated by an LLM in previous stages. The pipeline converts a JavaScript function implementing an action for an existing robot controller into a PDDL action. This process unfolds iteratively, ending with all desired actions being constructed following a series of successive pipeline cycles. Some prompts can be found in the appendix, as well as a discussion of the generalizability of the prompts, for use with other domains.

4.1 The Summarization Stage

The Summarization Stage summarizes a provided function implementing an action. The input of the Summarization Stage is a JavaScript function which implements an action. Its outputs are two natural-language summaries describing the provided function, one focused on giving a general overview, the other on elucidating that action's goals, preconditions, and effects in natural language. The Summarization Stage queries Llama3 to generate two different summaries of a function based on its implementation. The first is a concise, high-level 3-5 line summary of the function. The second is a more targeted 1-sentence summary of 25 words or less which describes the goals, preconditions and effects of the function. In Figure 1, these summaries are the green boxes numbered (1) and (2). To produce summary 1, Codestral is provided with a prompt containing its instructions and the function implementation. To generate summary 2, Codestral is provided a prompt containing the function implementation, and summary 1.

4.2 The Extraction Stage

The Extraction Stage has two sub-stages. The first substage selects the appropriate types and predicates for the target action from the provided pool via Codestral (Codestral-22B-v0.1) and prompts P3 and P4. Here, Prompts P3 and P4 are constructed by inserting the following inputs into a prompt template: (i) NL Function Summary (output 1 in Figure 1), (ii) NL Goals, Preconditions, & Effects Summary (output 2 in Figure 1), (iii) NL Domain Background Info, and (iv) either (for prompt P3) the Object Types Hierarchy or (for prompt P4) the Predicate Pool. Prompt P3 instructs Codestral to select the subset of types necessary to construct the specific PDDL action, without changing them in any way. The types selected by Codestral correspond to the green, #3 box in Figure 1. Similarly to Prompt P3, Prompt P4 instructs Codestral to select the subset of predicates necessary to construct the specificed PDDL action without duplicating or changing them in any way. The predicates selected by Codestral correspond to the green, #4 box in Figure 1.

In the second phase of the Extraction Stage, a PDDL action is generated based on the function summaries from the Summarization Stage via Codestral and prompt P5. Here, Prompt P5 is prepared by inserting the following inputs into the P5 template: (i) NL Function Summary (output 1 in Figure 1), (ii) NL Goals, Preconditions, & Effects Summary (output 2 in Figure 1), (iii) NL Domain Background Info, (iv) Selected Types (output 3 in Figure 1, generated with Prompt 3) and (v) Selected Predicates (output 4 in Figure 1, generated with Prompt 4). Note that this is essentially the same inputs used to create Prompts P3 and P4, except that the full predicate pool and type hierarchy are replaced by Codestral-selected subset of predicates and types, and P5 is given both the predicates and types, instead of one or the other.

Using this information, Codestral is prompted to write a single PDDL action using only the types and predicates it has been given, and any not-equal constraints needed to ensure variable uniqueness. Although we could attempt to generate the predicates in addition to the actions, for now we leave that to future work. Under optimal circumstances, Codestral will use the list of selected types when generating parameters for the action, and the list of selected predicates without any types included as fluents for the action. The human-provided background information and function summaries should guide the action's overall construction. The PDDL action generated by Codestral corresponds to the green, #5 box in Figure 1.

5 Evaluation

We evaluate how well the pipeline produces valid PDDL actions. The goal is to create a complete planning model and to probe the impacts of temperature and predicate specificity on that task. For the Summarization Stage of the pipeline we used Llama3 (8b-instruct-fp16, from Ollama) due to the strength of its general knowledge and language understanding capabilities. For the Extraction Stage of the pipeline, we used Codestral due to its superior ability to understand and generate code.

We ran experiments on a Lambda Laboratories quad AMD 1.5GHz EPYC 64 core system with 2TB of RAM, 8 NVIDIA A-100 80GB GPUs, and 32TB of SSD storage.

5.1 Ground-Truth Domains

To evaluate their accuracy, all actions generated by the pipeline were compared to several ground truth domains that were authored based on the Voyager JavaScript Functions, detailed below. These included domain and problem files for crafting a craft table, crafting a craft table and sticks and/or planks, and crafting all previously mentioned items and a wooden sword. These actions were chosen because generic Voyager functions for them existed, and because of their simplicity, as none require the player to interact with other Minecraft entities, consider time constraints, or implement overly complicated recipes to be made. This was important because compared with standard programming languages, PDDL has limited expressiveness, making it more difficult to use when modeling more advanced problem-solving scenarios. Such domains represent one way to craft these objects in Minecraft. In the open-world game, more options for crafting them exist.

All domain problem files had the player start in a designated location, without any items in their inventory or crafting tables in their environment. All items the player could

collect or craft were treated as distinct objects that could be passed into a given predicate as the player achieved their goals (i.e. `stick1`, `stick2`, `stick3`).

5.2 Pipeline Inputs

Function Code We use a subset of the JavaScript functions mostly taken from the Voyager project as our input functions. See the appendix for more information.

Natural Language (NL) Domain Background Info The NL background information provided as input to the Extraction Stage of the pipeline comprised 2-4 hand-written sentences explaining the expected preconditions and effects for the target action, and other pertinent facts related to it, like the specific ingredients needed to craft an item and whether it requires a crafting table. Details are in the appendix.

Objects Type Hierarchy This hierarchy included the types: `player`, `craftTable`, `plankBlock`, `stickItem`, `woodSword`, `woodPickaxe`[1], `gridSquare`, and `some_object`

Here, the `craftTable`, `plankBlock`, `woodBlock`, `stickItem`, `woodenSword`, and `woodenPickaxe` types represent specific objects in Minecraft, and `some_object` is their parent type.

Predicate Pool The predicate pool used for these experiments contained:

- `(agent_located_at ?agent - player ?square - gridSquare)`
- `(object_located_at ?someobject - some_object ?square - gridSquare)`
- `(connected ?current_position - gridSquare ?next_position - gridSquare)`
- `(HasWoodBlock ?agent - player ?wood - woodBlock)`
- `(HasPlankBlock ?agent - player ?plank - plankBlock)`
- `(HasStickItem ?agent - player ?stick - stickItem)`
- `(HasWoodenSword ?agent - player ?sword - woodSword)`
- `(HasWoodenPickaxe ?agent - player ?pickaxe - woodPickaxe)`
- `(HasCraftTable ?agent - player ?table - craftTable)`

5.3 Main Experiment

In the main experiment, we evaluated the quality of the action model generated under different values of the temperature parameter, given the same predicate pool We generated crafting actions for four items: a table, planks, sticks and a wooden sword. Temperature is a hyperparameter which controls the LLM's creativity, or ability to choose a less probable next word. Increasing the temperature enables the LLM to produce more varied responses. Here, Llama3 was set to a temperature of 0.3 to allow for a controlled level of creativity that wasn't excessively unpredictable, while the Codestral temperature was varied between 0.1, 0.5, 0.75 and 1. We generated 5 actions for each item and for each Codestral temperature setting. This created 20 PDDL actions per inputted JavaScript function, for a total of 80 PDDL actions.

Results All generated actions were categorized into correct, debatable or wrong, results shown in Figure 2. All entirely accurate actions were labeled as correct, while any actions with one or more errors were labeled as wrong. An action was only labeled as debatable if it would execute in some appropriate circumstances but not others. For example, if a predicate was added to an action's precondition allowing it to only be executed once or requiring that a player have access to a crafting table when it's not necessary.

Figure 2a shows craft-table-specific results. At the lowest Codestral temperature considered, the pipeline was consistently able to construct actions matching the ground truth. As higher temperatures were tested, this accuracy slowly diminished over time until 60% of the actions constructed by the pipeline were classified as incorrect.

Figure 2b shows specific results for crafting sticks in Minecraft. Again, at the lowest Codestral temperature considered, the pipeline was able to construct actions perfectly matched to the ground truth. As higher temperatures were tested, the pipeline outputs became less accurate. This was especially notable at a Codestral temperature settings of 0.5 and 1, where 80% of the actions constructed by the pipeline were classified as incorrect.

Figure 2c shows results for crafting planks in Minecraft. Here, unlike the other experiments run, lower temperature settings resulted in the pipeline repeatedly failing to construct any actions that matched the ground truth. Higher temperatures showed a slight improvement, with 20% of the actions created being classified as correct for the final two highest temperature settings.

Figure 2d shows results for crafting wooden swords in Minecraft. Aside from the results for crafting a craft table, this set of experiments achieved the second highest accuracy. Here, at the 0.1, 0.5 and 0.75 temperature settings, the pipeline was able to construct actions matching the ground truth 40% of the time, with the remaining 60% of the actions it created being executable under most but not all circumstances. This accuracy declined to 20% once the highest temperature setting was applied.

These experimental results show the craftCraftTable action was most often created correctly, followed by craftWoodenSword, craftSticks and craftPlanks. Additionally, they show lower temperature settings most often resulted in the pipeline correctly producing PDDL actions for crafting a table, sticks and a wooden sword but not for crafting planks. However, while at higher temperature settings the pipeline's ability to create correct PDDL actions declined for all crafting actions, it slightly improved for the craftPlank action.

To better understand this, we analyzed all incorrect action instances to determine consistent points of failure, shown in Figure 3. In all, 8 main failure points were identified

[1] This object type is included as part of this hierarchy and as part of the predicate pool via `HasWoodenPickaxe` because initially experiments for creating woodPickaxe actions were also conducted. However, we later narrowed our focus to the main actions presented in this paper, abandoning the initial, incomplete experiments related to crafting a woodPickaxe.

(a) Craft Table Results

(b) Sticks Results

(c) Planks Results

(d) Wooden Sword Results

Key

Correct Debateable Wrong

Figure 2: Experiment results showing the number of correct, debatable, and incorrect PDDL actions generated by the pipeline for crafting a craft table, planks, sticks, and a wooden sword in Minecraft. Each chart corresponds to a specific craft action, for which 5 runs were performed for each temperature setting combination where Llama3 was set to 0.3, and Codestral to 0.1, 0.5, 0.75, or 1. For the PDDL domains considered, correct actions were completely error-free, debatable actions were executable under certain circumstances, and incorrect actions were completely inexecutable.

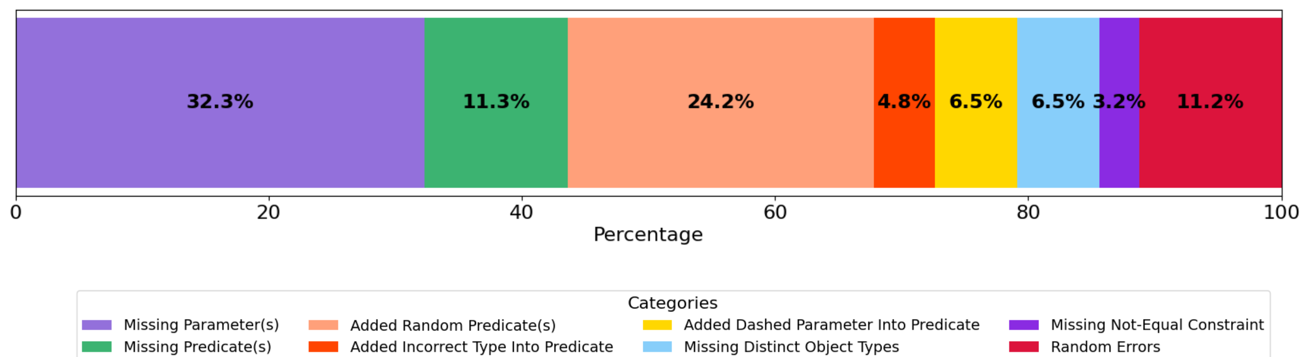


Figure 3: The frequency and distribution of errors among all debateable and incorrect crafting-specific actions.

with Codestral's most common mistake being leaving out required action parameters, followed by adding into actions unnecessary or potentially error-producing predicates. One-off random errors aside which included things like including extra parentheses, the next most common mistake was forgetting or modifying certain required action predicates.

Outside of these, Codestral's errors tended to be more granular. Sometimes it would incorrectly add dashed parameters into predicates, for example, adding - *woodenPlank* into the predicate `(HasPlank ?agent ?plank - woodenPlank)`. Or, it would misunderstand what predicates and variables belonged together, creating predicates like `(HasWood ?agent ?plank)` where `?plank` should have been the variable `?wood`. Lastly, with actions that included many, often similar objects, Codestral would sometimes fail to use distinct object types (e.g. using `Plank` vs. `Plank1`, `Plank2`, etc.) or include not-equal constraints to differentiate between objects. Because a large part of Minecraft is a player crafting new objects using different quantities of items in their inventory, this was especially problematic though a less commonly made mistake.

Examining this data in its entirety, we see across all experiments, every action was incorrect at least once. However, the errors leading to this seemed to be specific to each action. For example, whereas the craftPlank action was routinely wrong because Codestral forgot to pass in all required parameters, the craftSticks action was often wrong when Codestral failed to differentiate between its different stick objects (i.e. using `Stick` for all objects vs. `Stick1`, `Stick2`, etc.). Similarly, most of the craftWoodenSword actions were classified as debateable because Codestral designed them in such a way that crafting a sword was only possible if the player did not already possess one. As a result, the action was executable, but at most once. In Figure 4, we show a more in-depth breakdown of errors by specific crafting action type.

6 Discussion

Prior work has demonstrated that LLM's are poor at numeric and spatial reasoning (Xie et al. 2023). To address this challenge, the STRIPS planning formalism was adopted for this work to reduce the complexity and numeric reasoning needed to generate correct actions. However, during the initial prompt construction phase, Codestral's sensitivity to the language used in the prompts given to it seemed to result in errors of a comparable kind, some of which carried over to our experimental results. Thus, while the pipeline was often able to produce correct PDDL actions, our work also illustrates the difficulty in using LLM's to construct STRIPS PDDL planning models.

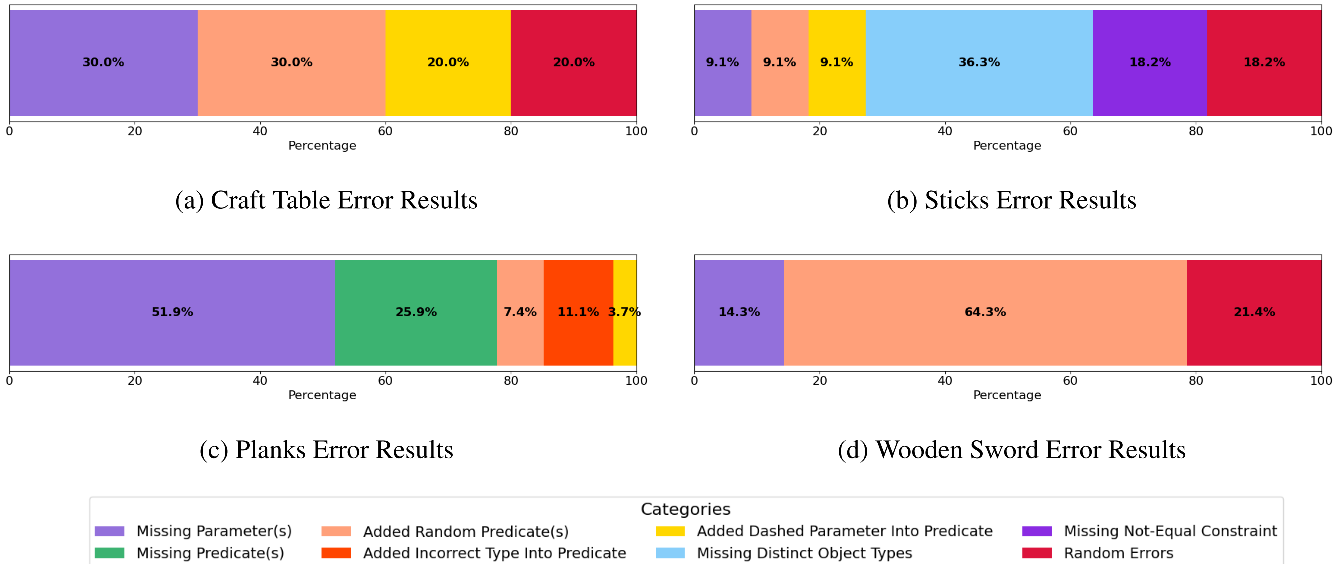
When designing our experimental prompts, we found that when specific domain background information indicating what an action's goals, preconditions and effects should be were given to Codestral, its accuracy greatly increased. But when the same information with no context was given (i.e. "two planks of any kind and one stick."), Codestral's ability to translate this information into valid actions dropped significantly. Similarly, telling Codestral when a crafting table was needed to execute a crafting action would at times result in erroneous outputs (i.e. including "A crafting table is not needed to perform this action" diminished the LLM's performance). This was most evident for the craftTable action, where to make a crafting table, Codestral would often illogically require the player to already possess one. However, when such information was omitted from its input, in some cases, Codestral's ability to construct this action rose to 100% accuracy during our experiments (see Figure 2).

In the initial testing phase, Codestral would also often ignore prompt directions to modify different predicates or include extraneous text along with its answer. For example, when `player` was omitted as a type given to Codestral, it would frequently add it to the predicates it used anyways, making the model more error-prone overall. Because of this, in the main experiments, `player` was always included in the list of types fed into the pipeline, and Codestral was instructed to provide its answers inside triple single quotes to differentiate them from excess words. While incorporating these things into the final prompts and supplied list of types/predicates given to the pipeline improved Codestral's accuracy, they inevitably raise further questions about whether other, less apparent linguistic elements present in each might still be influencing its outputs in unforeseen ways. It is not clear how to force the LLM to obey instructions on the first pass, but it may be possible to force the LLM to identify and/or revise its answer from external feedback as demonstrated by Chen et al. (2023); Smirnov et al. (2024).

With this, Minecraft is a complicated game for which writing an accurate domain that captures all facets of the game is difficult. When initially testing Codestral's response to the types and predicates from our ground-truth model, including the predicate `(object_in_inventory ?agent - player ?item - some_object)` instead of `(HasInInventory ?agent - player ?item - some_object)` was shown to more positively effect its output. Making this connection may be difficult for much more complicated domains.

In the main experiments, Codestral was consistently able to select the correct types for a given action with a very high accuracy. While sometimes it would select more types then it needed to make a given action, this overestimation never appeared to negatively effect its final outputted actions. When selecting predicates, Codestral fluctuated between exactly reproducing the selected predicates, and expanding the predicate using a set of parameter names, i.e., `(HasPlankBlock ?player ?plank)` vs `(HasPlankBlock ?player ?plank1)`, `(HasPlankBlock ?player ?plank2)`, `(HasPlankBlock ?player ?plank3)`. With certain actions, such as craftWoodenSword, Codestral would consistently omit certain predicates. Sometimes these missing predicates would be added to the final action it created. Other times, Codestral would entirely forget them or incorrectly change another predicate it had selected in an attempt to remedy this.

Despite this, such experiments revealed that the majority of the actions generated by the pipeline were either correct or required only minimal modifications—typically no more than a few adjustments—to achieve correctness. In fu-



(a) Craft Table Error Results

(b) Sticks Error Results

(c) Planks Error Results

(d) Wooden Sword Error Results

Figure 4: The frequency and distribution of errors among all debateable and incorrect actions by specific action type.

ture iterations of this work, incorporating an error detection loop could significantly enhance the accuracy of such actions, mitigating the occurrence of errors in them that are often minor or recurrent in nature.

7 Future Work

We see seven major areas for future work. The first area is an ablation study to evaluate the effectiveness of the LLM-generated function summaries and user-provided natural language domain information. The second area is to increase the amount of code we provide to the function summarizer. Currently we provide only the function definitions themselves, but not the code for any of the functions they call, forcing the LLM to "guess" based on the function names. Provided definitions for the sub-functions to the LLM, or providing the LLM with information of the expected call-hierarchy might increase the accuraccy of its summaries. The third area is to prompt the LLM to suggest predicates and types based on the function summaries and additional natural language domain knowledge. The fourth is to leverage a library like Outlines (Willard and Louf 2023) for structured generation with LLMs to reduce the number of syntax errors. The fifth area is to add an error-correction component for prompting the model to refine either summaries or actions. Previous works have demonstrated the utility of an error-correction loop, where some combination of parser and logic-checker analyze the LLM output and create error messages that can be used to query the LLM for corrections. Because we are learning actions for an existing bot, we could also detect errors via gameplay by tracking when plans constructed with our planning model fail. The sixth area is identifying complex functions and when they need to be broken into multiple PDDL actions. Controllers are typically written in Turing-complete languages. PDDL and other formal languages used by planners to express operators are typically not Turing-complete. This means that some functions will require being broken into multiple actions. When such actions appear in a plan, they will need to be translated back into a single function-call for the controller to execute. The seventh area is to test the performance of other LLMs in each stage of the pipeline.

8 Conclusion

We have created a two-stage pipeline for generating a PDDL planning model tailored to an existing operational model–i.e., a controller implementation for a robot. The pipeline uses an LLM first to summarize a function implementing a single action, then prompts an LLM to generate a PDDL action based on the summary, domain background knowledge provided as natural language, and provided predicates and types. Running the pipeline on each action implemented in the bot's controller can create a complete planning model. We evaluated the pipeline on JavaScript functions that implement actions for a Minecraft bot. We ran two primary experiments analyzing what PDDL actions the pipeline was capable of creating, the errors it often made, and the effect different LLM temperature settings had on its output.

Our results show that the pipeline successfully converts Javascript code into valid PDDL, albeit with some of the same issues as previous works on generating PDDL from natural language. The results also revealed that different temperatures exhibit a slight trade-off in action quality. When an action is generated correctly at a low temperature, raising the temperature will decrease its quality. On the other hand, a low-quality action generated at low temperature will slightly improve in quality using a higher temperature.

Acknowledgements

We thank NRL for funding this research. Sarath Sreedharan's research is supported in part by NSF grant 2303019.

References

Callanan, E.; Venezia, R. D.; Armstrong, V.; Paredes, A.; Chakraborti, T.; and Muise, C. 2022. MACQ: A Holistic View of Model Acquisition Techniques. In *Working notes of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS) at ICAPS 2022*.

Chen, Y.; Arkin, J.; Zhang, Y.; Roy, N.; and Fan, C. 2023. AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers. arXiv:2306.06531.

Ding, Y.; Zhang, X.; Amiri, S.; Cao, N.; Yang, H.; Kaminski, A.; Esselink, C.; and Zhang, S. 2023. Integrating Action Knowledge and LLMs for Task Planning and Situation Handling in Open Worlds. *Autonomous Robots*, 47(8): 981–997.

Ge, A.; Mao, W.; Zeng, D. D.; Kong, Q.; and Zhu, H. 2012. Extracting Action Knowledge in Security Informatics. In *2012 IEEE International Conference on Intelligence and Security Informatics*, 174–176. IEEE.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2024. Leveraging Pre-Trained Large Language Models to Construct and Utilize World Models for Model-Based Task Planning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

Guo, W.; Kingston, Z.; and Kavraki, L. E. 2024. CaStL: Constraints as Specifications through LLM Translation for Long-Horizon Task and Motion Planning. arxiv:2410.22225.

Huo, Y.; Tang, J.; Pan, Y.; Zeng, Y.; and Cao, L. 2020. Learning a Planning Domain Model From Natural Language Process Manuals. *IEEE Access*, 8: 143219–143232.

Jin, K.; Chen, H.; and Zhuo, H. H. 2022. Text-Based Action-Model Acquisition for Planning. arxiv:2202.08373.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning Models from Natural Language Action Descriptions. *Proceedings of the International Conference on Automated Planning and Scheduling*, 27: 434–442.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. arxiv:2304.11477.

Oates, T.; Alford, R.; Johnson, S.; and Hall, C. 2024. Using Large Language Models to Extract Planning Knowledge from Common Vulnerabilities and Exposures. In *Working notes of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS) at ICAPS 2024*.

Oswald, J.; Srinivas, K.; Kokel, H.; Lee, J.; Katz, M.; and Sohrabi, S. 2024. Large Language Models as Planning Domain Generators. *Proc. of the International Conference on Automated Planning and Scheduling*.

Pallagani, V.; Muppasani, B. C.; Roy, K.; Fabiano, F.; Loreggia, A.; Murugesan, K.; Srivastava, B.; Rossi, F.; Horesh, L.; and Sheth, A. 2024. On the Prospects of Incorporating Large Language Models (LLMs) in Automated Planning and Scheduling (APS). In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 432–444.

Sil, A.; and Yates, A. 2011. Extracting STRIPS Representations of Actions and Events. In *Proceedings of the International Conference Recent Advances in Natural Language Processing 2011*.

Smirnov, P.; Joublin, F.; Ceravola, A.; and Gienger, M. 2024. Generating Consistent PDDL Domains with Large Language Models. arxiv:2404.07751.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arxiv:2305.16291.

Willard, B. T.; and Louf, R. 2023. Efficient Guided Generation for LLMs. *arXiv preprint arXiv:2307.09702*.

Xie, Y.; Yu, C.; Zhu, T.; Bai, J.; Gong, Z.; and Soh, H. 2023. Translating Natural Language to Planning Goals with Large-Language Models. arXiv:2302.05128.

Yordanova, K.; and Kirste, T. 2016. Learning Models of Human Behaviour from Textual Instructions:. In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence*, 415–422.

Yuan, H.; Zhang, C.; Wang, H.; Xie, F.; Cai, P.; Dong, H.; and Lu, Z. 2023. Plan4MC: Skill Reinforcement Learning and Planning for Open-World Minecraft Tasks. arxiv:2303.16563.

Zhu, X.; Chen, Y.; Tian, H.; Tao, C.; Su, W.; Yang, C.; Huang, G.; Li, B.; Lu, L.; Wang, X.; Qiao, Y.; Zhang, Z.; and Dai, J. 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory. arxiv:2305.17144.

Appendix: Creating PDDL Models from Javascript using LLMs: Preliminary Results

In this appendix we show several examples of correct and incorrect PDDL actions generated during Experiment 1, the work outlined in the main paper. We subsequently present Experiment 2, a preliminary experiment which tests the effect of different predicate sets on actions produced by the pipeline, for which a few example outputs are also provided. Following this, we describe the details of (1) the JavaScript functions comprising the bot controller that we used as input, (2) the hand-written ground-truth model used to compare against the generated model, (3) the natural language domain background information, and the (4) prompts fed into the pipeline.

A Experiment 1: Model Generation Under Different Temperatures

Experiment 1 revealed our pipeline could successfully convert Javascript code into valid PDDL but that different LLM temperature settings exhibited a slight trade-off in the actions produced. Specifically, low temperature settings were shown to produce better quality actions, while higher temperatures produced slightly worse ones. Here, we provide several examples of correct versus incorrect actions produced by the pipeline for crafting a table, planks, sticks, and a wooden sword.

A.1 Craft Table Actions

```
(:action craftCraftTable
 :parameters (?agent - player ?plank1 - plankBlock ?plank2 - plankBlock ?plank3 - plankBlock ?plank4 - pl
ankBlock ?table - craftTable)
 :precondition (and (HasPlankBlock ?agent ?plank1)
                    (HasPlankBlock ?agent ?plank2)
                    (HasPlankBlock ?agent ?plank3)
                    (HasPlankBlock ?agent ?plank4)
                    (not (= ?plank1 ?plank2))
                    (not (= ?plank1 ?plank3))
                    (not (= ?plank1 ?plank4))
                    (not (= ?plank2 ?plank3))
                    (not (= ?plank2 ?plank4))
                    (not (= ?plank3 ?plank4)))
 :effect (and (not (HasPlankBlock ?agent ?plank1))
              (not (HasPlankBlock ?agent ?plank2))
              (not (HasPlankBlock ?agent ?plank3))
              (not (HasPlankBlock ?agent ?plank4))
              (HasCraftTable ?agent ?table)))
```

(A) Correct craft table action for temperature settings of 0.3 for Llama3 and 0.1 for Codestral

```
(:action craftCraftTable
  :parameters (?agent - player)
  :precondition (and
    (HasPlankBlock ?agent ?plank1 - plankBlock)
    (HasPlankBlock ?agent ?plank2 - plankBlock)
    (HasPlankBlock ?agent ?plank3 - plankBlock)
    (HasPlankBlock ?agent ?plank4 - plankBlock)
    (not (= ?plank1 ?plank2))
    (not (= ?plank1 ?plank3))
    (not (= ?plank1 ?plank4))
    (not (= ?plank2 ?plank3))
    (not (= ?plank2 ?plank4))
    (not (= ?plank3 ?plank4))
  )
  :effect (and
    (not (HasPlankBlock ?agent ?plank1))
    (not (HasPlankBlock ?agent ?plank2))
    (not (HasPlankBlock ?agent ?plank3))
    (not (HasPlankBlock ?agent ?plank4))
    (HasCraftTable ?agent ?newTable - craftTable)
  )
)
```

(B) Incorrect craft table action for temperature settings of 0.3 for Llama3 and 0.5 for Codestral

In Minecraft, a crafting table is an essential item which allows players to craft a wide variety of items and tools, vital for advancing in the game. To make a crafting table, a player must possess four unique plank objects. Upon this action's execution, these resources are consumed and the player is left with a crafting table in their inventory.

Figure A shows a correct craft table action generated by the pipeline at temperature settings of 0.3 for Llama3 and 0.1 for Codestral. Here, distinct parameters for the `player`, `craftTable` and each individual `plankBlock` object are passed in. To reflect the four unique plank objects required to make a craft table, Codestral correctly includes four `HasPlankBlock`

predicates along with the necessary number of not-equal constraints to guarantee all plank objects are unique. The effect of this action is that all `HasPlankBlock` predicates are false, indicating this resource has been used and the `HasCraftTable` predicate is true, indicating the player now possesses one.

In contrast, Figure B shows an incorrect craft table action generated by the pipeline at temperature settings of 0.3 for Llama3 and 0.5 for Codestral. Here, Codestral has failed to pass in distinct parameters for `craftTable` and each individual `plankBlock` object. Additionally, it has mistakenly included dashed types inside most of the predicates used inside the action. For example, including `- craftTable` inside the `HasCraftTable` predicate where it does not belong.

A.2 Craft Planks Actions

```
 (:action craftWoodenPlanks
  :parameters (?agent - player ?wood1 ?wood2 ?wood3 ?wood4 - woodBlock ?plank1 ?plank2 ?plank3 ?plank4 -
plankBlock)
  :precondition (and
    (HasWoodBlock ?agent ?wood1)
    (HasWoodBlock ?agent ?wood2)
    (HasWoodBlock ?agent ?wood3)
    (HasWoodBlock ?agent ?wood4)
    (not (= ?wood1 ?wood2))
    (not (= ?wood1 ?wood3))
    (not (= ?wood1 ?wood4))
    (not (= ?wood2 ?wood3))
    (not (= ?wood2 ?wood4))
    (not (= ?wood3 ?wood4))
    (not (= ?plank1 ?plank2))
    (not (= ?plank1 ?plank3))
    (not (= ?plank1 ?plank4))
    (not (= ?plank2 ?plank3))
    (not (= ?plank2 ?plank4))
    (not (= ?plank3 ?plank4))
  )
  :effect (and
    (not (HasWoodBlock ?agent ?wood1))
    (not (HasWoodBlock ?agent ?wood2))
    (not (HasWoodBlock ?agent ?wood3))
    (not (HasWoodBlock ?agent ?wood4))
    (HasPlankBlock ?agent ?plank1)
    (HasPlankBlock ?agent ?plank2)
    (HasPlankBlock ?agent ?plank3)
    (HasPlankBlock ?agent ?plank4)
  )
)
```

(C) Correct craft planks action for temperature settings of 0.3 for Llama3 and 0.5 for Codestral

```
 (:action craftWoodenPlanks
  :parameters (?agent - player ?wood1 - woodBlock ?wood2 - woodBlock ?wood3 - woodBlock ?wood4 - woodBloc
k)
  :precondition (and (HasWoodBlock ?agent ?wood1)
                     (HasWoodBlock ?agent ?wood2)
                     (HasWoodBlock ?agent ?wood3)
                     (HasWoodBlock ?agent ?wood4)
                     (not (= ?wood1 ?wood2))
                     (not (= ?wood1 ?wood3))
                     (not (= ?wood1 ?wood4))
                     (not (= ?wood2 ?wood3))
                     (not (= ?wood2 ?wood4))
                     (not (= ?wood3 ?wood4)))
  :effect (and (not (HasWoodBlock ?agent ?wood1))
               (not (HasWoodBlock ?agent ?wood2))
               (not (HasWoodBlock ?agent ?wood3))
               (not (HasWoodBlock ?agent ?wood4))
               (HasPlankBlock ?agent ?wood1)
               (HasPlankBlock ?agent ?wood2)
               (HasPlankBlock ?agent ?wood3)
               (HasPlankBlock ?agent ?wood4)))
```

(D) Incorrect craft plank action for temperature settings of 0.3 for Llama3 and 0.5 for Codestral

Planks are resources used to create many common items in Minecraft. To craft planks, a player must possess four unique wood block objects. Upon this action's execution, these resources are consumed and the player is left with four unique plank block objects in their inventory.

Figure C shows a correct craft planks action generated by the pipeline at temperature settings of 0.3 for Llama3 and 0.5 for Codestral. Here, distinct parameters for `player` and each individual `woodBlock` and `plankBlock` objects are passed in. In accordance with the Minecraft recipe, Codestral correctly includes four unique wood block objects required to make planks in the action's precondition along with the proper number of not-equal constraints to guarantee all wood and plank block objects are unique. The effect of this action is that all `HasWoodBlock` predicates are false, indicating these resources have been used,

and four true `HasPlankBlock` predicates, indicating the player now possesses these. Here, no `HasCraftTable` predicate is included in the action because in Minecraft craft tables are not required to craft planks.

Figure D shows an incorrect craft planks action generated by the pipeline at the same temperature settings of 0.3 for Llama3 and 0.5 for Codestral. Here, Codestral has failed to pass in distinct parameters for each individual `plankBlock` object (i.e. `?plank1-4`). Perhaps due to this, it has also failed to include not-equal constraints for any of the `plankBlock` objects, and has mistakenly used `wood1`, `wood2`, `wood3` and `wood4` inside the `HasPlankBlock` predicates in the action effect section, instead of the correct variables `plank1`, `plank2`, `plank3` and `plank4`.

A.3 Craft Sticks Actions

```
(:action craftSticks
 :parameters (?agent - player ?plank1 - plankBlock ?plank2 - plankBlock ?stick1 - stickItem ?stick2 - sti
ckItem ?stick3 - stickItem ?stick4 - stickItem)
 :precondition (and (HasPlankBlock ?agent ?plank1)
                    (HasPlankBlock ?agent ?plank2)
                    (not (= ?plank1 ?plank2))
                    (not (= ?stick1 ?stick2))
                    (not (= ?stick1 ?stick3))
                    (not (= ?stick1 ?stick4))
                    (not (= ?stick2 ?stick3))
                    (not (= ?stick2 ?stick4))
                    (not (= ?stick3 ?stick4)))
 :effect (and (not (HasPlankBlock ?agent ?plank1))
              (not (HasPlankBlock ?agent ?plank2))
              (HasStickItem ?agent ?stick1)
              (HasStickItem ?agent ?stick2)
              (HasStickItem ?agent ?stick3)
              (HasStickItem ?agent ?stick4)))
```

(E) Correct craft sticks action for temperature settings of 0.3 for Llama3 and 0.1 for Codestral

```
(:action craftSticks
   :parameters (?agent - player ?plank1 - plankBlock ?plank2 - plankBlock ?stick - stickItem)
   :precondition (and (HasPlankBlock ?agent ?plank1)
                      (HasPlankBlock ?agent ?plank2)
                      (not (= ?plank1 ?plank2)))
   :effect (and (not (HasPlankBlock ?agent ?plank1))
                (not (HasPlankBlock ?agent ?plank2))
                (HasStickItem ?agent ?stick)
                (HasStickItem ?agent ?stick)
                (HasStickItem ?agent ?stick)
                (HasStickItem ?agent ?stick)))
```

(F) Incorrect craft sticks action for temperature settings of 0.3 for Llama3 and 0.5 for Codestral

Sticks are resources that can be crafted or directly gathered, fished or traded for in the Minecraft world. To craft sticks, a player must possess two unique plank block objects. Upon this action's execution, these resources are consumed and the player is left with four unique stick block objects in their inventory.

Figure E shows a correct craft sticks action generated by the pipeline at temperature settings of 0.3 for Llama3 and 0.1 for Codestral. Here, distinct parameters for `player` and each individual `plankBlock` and `stickItem` object are passed in. In the action precondition, Codestral correctly includes the two unique plank block objects to make sticks, and the proper number of not-equal constraints to guarantee all plank and stick objects are unique. The effect of this action is that all `HasPlankBlock` predicates are false, indicating these resources have been used, and four true `HasStickItem` predicates, indicating the player now possesses these. Here, no `HasCraftTable` predicate is included in the action because in Minecraft craft tables are not required to craft sticks.

Figure F shows an incorrect craft sticks action generated by the pipeline at the temperature settings of 0.3 for Llama3 and 0.5 for Codestral. Here, Codestral has failed to pass in distinct parameters for each individual `stickItem` object (i.e. `?stick1-4`), opting to just include `stick - stickItem`. Further, it has failed to include any stick object not-equal constraints in the action's precondition, and any distinct stick variables in the action's effect section, choosing instead to use `?stick` inside all `HasStickItem` predicates rather then `?stick1`, `?stick2`, `?stick3`, and `?stick4`.

A.4 Craft Wooden Sword Actions

Wooden swords are categorized as weapons in Minecraft, most often used during combat. To craft a wooden sword, a player must possess two unique plank block objects, a stick object and a crafting table. Upon this action's execution, the plank and stick resources are consumed and the player is left with a single wooden sword object in their inventory.

Figure G shows a correct craft wooden sword action generated by the pipeline at temperature settings of 0.3 for Llama3 and 0.1 for Codestral. Here, distinct parameters for `player`, `woodenSword`, and each individual `plankBlock` and

```
 (:action craftWoodenSword
  :parameters (?agent - player ?plank1 - plankBlock ?plank2 - plankBlock ?stick - stickItem ?table - craf
tTable ?sword - woodenSword)
  :precondition (and (HasPlankBlock ?agent ?plank1)
                     (HasPlankBlock ?agent ?plank2)
                     (not (= ?plank1 ?plank2))
                     (HasStickItem ?agent ?stick)
                     (HasCraftTable ?agent ?table))
  :effect (and (not (HasPlankBlock ?agent ?plank1))
               (not (HasPlankBlock ?agent ?plank2))
               (not (HasStickItem ?agent ?stick))
               (HasWoodenSword ?agent ?sword)))
```

(G) Correct craft wooden sword action for temperature settings of 0.3 for Llama3 and 0.1 for Codestral

```
 (:action craftWoodenSword
    :parameters (?agent - player ?plank1 - plankBlock ?plank2 - plankBlock ?stick - stickItem ?table - cr
aftTable ?sword - woodenSword)
    :precondition (and
        (not (= ?plank1 ?plank2))
        (HasPlankBlock ?agent ?plank1)
        (HasPlankBlock ?agent ?plank2)
        (HasStickItem ?agent ?stick)
        (HasCraftTable ?agent ?table)
        (not (exists (?sw - woodenSword) (HasWoodenSword ?agent ?sw)))
    )
    :effect (and
        (not (HasPlankBlock ?agent ?plank1))
        (not (HasPlankBlock ?agent ?plank2))
        (not (HasStickItem ?agent ?stick))
        (HasWoodenSword ?agent ?sword)
    )
)
```

(H) Incorrect craft wooden sword action for temperature settings of 0.3 for Llama3 and 1 for Codestral

`stickItem` object are passed in. In the action precondition, Codestral correctly includes the two unique plank block objects, stick object and craft table. Additionally, it includes a single not-equal constraint to guarantee both plank objects are unique. The effect of this action is that the `HasStickItem` and all `HasPlankBlock` predicates are false, indicating these resources have been used, and the `HasWoodenSword` predicate is true, indicating the player now possesses one. Here, a `HasCraftTable` predicate is included because the player must possess one to craft a wooden sword in Minecraft.

Figure H shows an incorrect craft wooden sword action generated by the pipeline at the temperature settings of 0.3 for Llama3 and 1 for Codestral. Here, Codestral has included random predicates in the action's precondition, effectively necessitating that to craft a wooden sword, the player cannot already possess one nor can any exist. In one of these randomly added predicates, Codestral has also mistakenly added a `- wooden sword` dashed type. However, without these mistakes, the action would otherwise be correct.

B Experiment 2: How does predicate specificity impact action quality?

To test how feeding the pipeline different predicate sets might effect the actions it constructs, a preliminary series of tests were run. In addition to the crafting actions created in the previous experiment, here collect, place, and move actions were also generated. During this experiment, Llama3 was set to a temperature of 0, Codestral temperature was varied between the temperatures of 0, 0.5, and 1, and three different predicate pools varying in their level of abstraction and size were used as pipeline inputs.

B.1 Predicate Pools

All three pools contained a basic set of predicates:

- `(agent_located_at ?agent - player ?square - gridSquare)`
- `(object_located_at ?someobject - some_object ?square - gridSquare)`
- `(connected ?current_position - gridSquare ?next_position - gridSquare)`

Added to this basic set to create the different pools were a set of predicates for indicating if the Minecraft agent possessed an item. Predicate pool 1 contained an abstract predicate for this: `object_in_inventory`. Predicate pool 2 contained item-specific predicates for this:

- `(HasWoodBlock ?agent - player ?wood - woodBlock)`
- `(HasPlankBlock ?agent - player ?plank - plankBlock)`
- `(HasStickItem ?agent - player ?stick - stickItem)`
- `(HasWoodenSword ?agent - player ?sword - woodSword)`
- `(HasWoodenPickaxe ?agent - player ?pickaxe - woodPickaxe)`
- `(HasCraftTable ?agent - player ?table - craftTable)`

Predicate pool 3 contained the predicates from pool 1 and 2.

For all temperature setting combinations, a single action was generated for moving, placing an item, collecting an item, and crafting a table, planks, sticks, and a wooden sword, using each of these predicate sets. This resulted in 3 PDDL actions being generated per inputted JavaScript function, for a total of 21 PDDL actions per predicate pool or a total of 63 actions across all predicate pools.

B.2 Using the `object_in_inventory` Predicate Only

To test if using less and more abstract predicates increased Codestral's ability to create correct PDDL actions, predicate pool 1 was used. This approach resulted in the pipeline successfully creating actions for collecting and placing items (shown in Figures J and K) while also getting the craftPlank action correct more times compared to the use of other predicate sets. However, for item-specific crafting actions, this predicate set seemed to introduce more errors, as here the pipeline was never able to successfully create the craftSticks, craftCraftingTable or craftWoodenSword actions or the more generic move action.

Interestingly, most incorrect craftCraftingTable actions (shown in Figure L) stemmed from the `object_located_at` predicate. As Codestral would select the correct number of predicates needed to craft a table, but would mistakenly use `object_located_at` in place of the `object_in_inventory` predicate. The craftWoodenSword action was consistently wrong due to random errors like additional parentheses, and craftSticks as a result of failing to differentiate between its different stick objects. The move action consistently messed up the variables contained within the `object_located_at` predicate. We show this below in Figure I where the agent begins at location `?current_position` and the object it's moving to at `?goal_position`. However, after the action is executed, while the agent is at `?goal_position`, the object they've moved to is now shown as being at their former location, `?current_position`. Additionally, we see Codestral has randomly added the predicate `(different ?current_position ?goal_position)` into the action's precondition.

```
(:action move_player
:parameters (?player - player ?current_position - gridSquare ?goal_position - gridSquare ?goal_item - so
me_object)
:precondition (and (agent_located_at ?player ?current_position)
                   (object_located_at ?goal_item ?goal_position)
                   (connected ?current_position ?goal_position)
                   (different ?current_position ?goal_position)
                   (not (= ?current_position ?goal_position))
                   (not (= ?goal_item ?current_position)))
:effect (and (not (agent_located_at ?player ?current_position))
             (agent_located_at ?player ?goal_position)
             (object_located_at ?goal_item ?current_position)
             (not (object_located_at ?goal_item ?goal_position))))
```

(I) Incorrect move action for Llama3 and Codestral temperature settings of 0 using predicate pool 1

```
(:action collect-object
:parameters (?player - player ?item - some_object ?square - gridSquare)
:precondition (and (agent_located_at ?player ?square)
                   (object_located_at ?item ?square)
                   (not (object_in_inventory ?item))
                   (not (= ?player ?item)))
:effect (and (object_in_inventory ?item)
             (not (object_located_at ?item ?square))))
```

(J) Correct collect action for Llama3 and Codestral temperature settings of 0 using predicate pool 1

B.3 Using Item-Specific Predicates Only

To test if more specific predicates increased Codestral's ability to create correct PDDL actions, it was next given predicate pool 2. Using this item-specific predicate set helped the pipeline create valid item-specific PDDL actions for craftCraftingTable,

```
(:action place_item
:parameters (?player - player ?item - some_object ?square - gridSquare)
:precondition (and (object_in_inventory ?player ?item)
                   (agent_located_at ?player ?square)
                   (not (object_located_at ?item ?square))
                   (not (= ?item ?player)))
:effect (and (not (object_in_inventory ?player ?item))
             (object_located_at ?item ?square)))
```

(K) Correct place action for Llama3 and Codestral temperature settings of 0 using predicate pool 1

```
(:action craft_crafting_table
:parameters (?player - player ?plank1 ?plank2 ?plank3 ?plank4 - plank ?square1 ?square2 ?square3 ?square
4 - gridSquare)
:precondition (and
                (object_located_at ?plank1 ?square1)
                (object_located_at ?plank2 ?square2)
                (object_located_at ?plank3 ?square3)
                (object_located_at ?plank4 ?square4)
                (not (= ?plank1 ?plank2))
                (not (= ?plank1 ?plank3))
                (not (= ?plank1 ?plank4))
                (not (= ?plank2 ?plank3))
                (not (= ?plank2 ?plank4))
                (not (= ?plank3 ?plank4))
                (not (= ?square1 ?square2))
                (not (= ?square1 ?square3))
                (not (= ?square1 ?square4))
                (not (= ?square2 ?square3))
                (not (= ?square2 ?square4))
                (not (= ?square3 ?square4)))
:effect (and
        (not (object_located_at ?plank1 ?square1))
        (not (object_located_at ?plank2 ?square2))
        (not (object_located_at ?plank3 ?square3))
        (not (object_located_at ?plank4 ?square4))
        (object_in_inventory ?player ?crafting_table)))
```

(L) Incorrect craft table action for Llama3 and Codestral temperature settings of 0 using predicate pool 1

craftSticks, craftPlank and craftWoodenSword. However, it reduced the correctness of the more generic collect, place and move actions, getting them all wrong. Here, the collect and place actions (shown below in Figures M and N) incorrectness consistently stemmed from random errors whereas the move action again mixed up the variables contained within the `object_located_at` predicate, like predicate pool 1.

B.4 Using Item-Specific & `object_in_inventory` Predicates

To test if using a combination of abstract and specific predicates increased Codestral's ability to create correct PDDL actions, it was given predicate pool 3. Using this combined predicate set, Codestral was able to achieve the highest accuracy among all constructed actions. Out of three runs, all craftCraftingTable and craftWoodenSword actions, two craftSticks actions, and one craftPlank action was correct. Though an executable collect action was generated across all three temperature settings, due to Codestral modifying one of its predicates compared to the ones given, we classify these as debateable. Unfortunately we see for predicate pool 3, Codestral was never able to correctly generate the place or move actions. Here, the place action suffered from the introduction of random errors and typed predicates where they didn't belong. And the move action again mixed up the variables contained within the `object_located_at` predicate, as shown with predicate pools 1 and 2.

B.5 Discussion

Across all temperature settings and predicate pools we see the pipeline was never able to generate a correct move action. Additionally, we see while predicate pool 3 seemed to have the highest overall accuracy among all predicate sets, predicate pool 1 resulted in the highest number of correct place, collect and craftPlank actions, while predicate pool 2 seemed to more consistently generate correct item-specific actions, such as craftWoodenSword. However, because the number of actions generated among all predicate pools and temperature settings was small, more testing would be needed to confirm the consistency of these results and to identify other emergent trends in the data like whether or not the different temperature settings had much impact on the produced actions.

```
(:action placeItem
 :parameters (?agent - player ?object - minecraft_object ?square - gridSquare)
 :precondition (and
                (agent_located_at ?agent ?square)
                (or
                  (HasWoodBlock ?agent ?object)
                  (HasPlankBlock ?agent ?object)
                  (HasStickItem ?agent ?object)
                  (HasWoodenSword ?agent ?object)
                  (HasWoodenPickaxe ?agent ?object)
                  (HasCraftTable ?agent ?object))
                (not (object_located_at ?object ?square)))
 :effect (and
          (object_located_at ?object ?square)
          (not (HasWoodBlock ?agent ?object))
          (not (HasPlankBlock ?agent ?object))
          (not (HasStickItem ?agent ?object))
          (not (HasWoodenSword ?agent ?object))
          (not (HasWoodenPickaxe ?agent ?object))
          (not (HasCraftTable ?agent ?object))))
```

(M) Incorrect place action for Llama3 and Codestral temperature settings of 0 using predicate pool 2. Here, Codestral forgot to add ”or” to the effect section to pinpoint which item is being placed.

```
 (:action collect
    :parameters (?player - player ?wood - woodBlock ?square - gridSquare)
    :precondition (and
        (agent_located_at ?player ?square)
        (object_located_at ?wood ?square)
        (not (= ?player ?wood))
    )
    :effect (and
        (not (object_located_at ?wood ?square))
        (increase (player_inventory ?player) ?wood)
        (notify "Collected wood.")
    )
)
```

(N) Incorrect collect action for Llama3 and Codestral temperature settings of 0 using predicate pool 2. Here, Codestral became focused on only placing wood objects, only passing in and using the parameter only passed in the parameter `?wood`. Additionally, it added in random, incorrect predicates such as `(notify "Collect wood.")`.

C Pipeline Inputs & Prompts

C.1 Voyager Javascript Functions

All Javascript bot controller functions were sourced from the Git repository of the Voyager project or PrismarineJS/mineflayer-pathfinder repository. We selected four: craftItem, placeItem, goTo, and CollectBlock. Note that the craftItem and placeItem functions call multiple smaller functions during their execution, including the goTo and CollectBlock functions. All downloaded functions were used-as-is, excepting the CollectBlock function, which was extracted from an enclosing class. This was done to help focus the LLM's attention only on the code it should summarize. These functions can be found at the following web addresses:

- **craftItem Function:** https://github.com/MineDojo/Voyager/blob/55e45a880755d0c8c66ca7fb5fe7962ac8974f89/voyager/control_primitives_context/craftItem.js
- **GoTo/Move Function:** https://github.com/PrismarineJS/mineflayer-pathfinder/blob/master/lib/goto.js
- **CollectBlock Function:** https://github.com/MineDojo/Voyager/blob/55e45a880755d0c8c66ca7fb5fe7962ac8974f89/voyager/env/mineflayer/mineflayer-collectblock/src/CollectBlock.ts
- **placeItem:** https://github.com/MineDojo/Voyager/blob/55e45a880755d0c8c66ca7fb5fe7962ac8974f89/voyager/control_primitives_context/placeItem.js

C.2 Ground-Truth Minecraft PDDL Models

In the ground-truth Minecraft domains used in Experiment 1 and 2, the generic functions expressed in the Voyager JavaScript Functions were converted into multiple, specific function instances. For example, the compound craftItem Voyager function

which has a player move to a crafting table, look up a recipe and craft an item, was instead expressed as four different actions in our domains: move, collectWood, craftPlanks, and craftTable. More specifically, the "Make crafting table" domain included move, collectWood, craftPlank and craftTable actions. The "Make a crafting table and wooden sword" domain included all previously mentioned actions, as well as craftWoodenSword action.

C.3 Summarization Stage: Example Output NL Summaries

Here we give example summaries generated by the Summarization Stage for each of our test functions. These summaries are generated using Prompts P1 and P2 and inserted into Prompts P3, P4, and P5 in the Extraction Stage of the Pipeline. Note that for readability, some of the syntax has been omitted from this prompt (Ex: "‘ marks around some responses for later parsing purposes.) or slightly modified (Ex: adding bullets to each item under "Recipes Domain Background Info."). The content itself was not changed.

For Experiment 1 The following is an example summary created during the Summarization Stage for the Voyager function `craftItem`.

Function Name: `craftItem`

NL Function Summary:
This `craftItem` function enables a player to create a specific item in Minecraft by using a set of resources and a crafting table. The function requires the name of the desired item and the quantity to be crafted, and it will automatically gather the necessary resources and perform the crafting process. It first locates the crafting table and then uses its position to navigate the bot towards it before proceeding with the crafting action.

NL Goals, Preconditions & Effects Function Summary:
The goal of the `craftItem` function is to enable a player to create a specific item in Minecraft by using a set of resources and a crafting table, allowing them to acquire the desired quantity of the item. The precondition of the `craftItem` function is that there must be a crafting table present in the game world for the player to use, as well as sufficient resources available to craft the item, and that the player has access to these resources and can navigate to the crafting table. The effect of the `craftItem` function is that it will gather the necessary resources and perform the crafting process at the crafting table, resulting in the creation of the specified quantity of the desired item, which becomes available for the player to collect.

NL Function One Sentence Summary:
The player must have access to a crafting table and sufficient resources to craft an item specified by name and quantity.

For Experiment 2 The following are example summaries created during the Summarization Stage for the Voyager functions `CollectBlock`, `goTo`, and `placeItem`.

Function Name: `CollectBlock`

NL Function Summary:
The collect function is a JavaScript method that simulates the action of collecting specific blocks in Minecraft. It takes two main parameters: ‘target‘ which specifies what blocks to collect, and ‘options‘ which defines various settings for the collection process. The function uses an asynchronous approach and returns a promise indicating its completion or failure.

NL Goals, Preconditions & Effects Function Summary:
The goal of the ’collect’ function is to simulate the collection of specified blocks in Minecraft, allowing players to harvest resources and items. The precondition for this action is that the player must have a pathfinder and tool plugin enabled, as well as specify valid target blocks or an array of blocks to collect, along with optional settings for the collection process. The effect of executing the ’collect’ function is that it will modify the player’s inventory by adding the collected items, potentially clearing any existing targets, and emitting a signal indicating the completion of the collection task.

NL Function One Sentence Summary:
The collect action requires a JavaScript environment and a player with access to a Minecraft game to execute. If these preconditions are met, the function will simulate collecting specified blocks, using an asynchronous approach, and return a promise indicating its completion or failure.

Function Name: `goTo`

NL Function Summary:
Here is a summary of the ‘error‘ function: The ‘error‘ function creates an error object with a given name and message, then returns it. It is used to represent errors that occur during execution of a goal in Minecraft. The function is designed to be reusable and flexible, allowing for customization of error messages and names depending on the specific situation.

NL Goals, Preconditions & Effects Function Summary:
The goal of the ’goto’ function is to allow a player to execute a goal and run a callback when that goal is reached, while handling potential errors and cleanup operations. The precondition for this action is that the player must be able to set a goal using a pathfinder, and there must be no obstacles or conflicts preventing the execution of the goal. The effect of this action is that if the goal is successfully executed, the callback will run and any error messages or events triggered during the execution process will be handled, otherwise the error will be rejected and an error message will be displayed to the player.

NL Function One Sentence Summary:
The ‘error‘ function requires a given name and message to create an error object. If these preconditions are met, it returns the error object for representing errors during goal execution in Minecraft.

Function Name: `placeItem`

NL Function Summary:
The ‘placeItem‘ function is a JavaScript function that allows a player to place an item at a specified position in the Minecraft game world. It takes three parameters: a bot object, the name of the item to be placed, and the position where it should be placed. The function first finds the item in the bot’s inventory and then determines the direction it should be placed based on the surrounding blocks. Once the direction is determined, the function navigates the bot to the placement position, equips the item for placing, and then places the item at the specified position.

NL Goals, Preconditions & Effects Function Summary:
The goal of the ’placeItem’ function is to allow a player to place an item at a specified position in the Minecraft game world. The precondition of the ’placeItem’ function is that the player must have the required item in their inventory, and there must be a solid block in the surrounding area that determines the direction of placement. The effect of the ’placeItem’ function is that it changes the state of the game world by placing the item at the specified position, and also updates the player’s inventory by removing the used item.

NL Function One Sentence Summary:
The player must have the item they want to place in their inventory for this function to work. If the item is available, it will be placed at the specified position and direction based on surrounding blocks.

C.4 NL Domain Background Information

Natural language domain background information is provided to Prompts P3, P4, and P5, which are respectively used to select the types, select the predicates, and generate the action for a particular function.

For Experiment 1 The following are the four user-provided NL Domain Background Information texts corresponding to each item for which a `craftItem` action was generated.

- **Crafting Table:** The precondition for this action is the player must have 4 individual planks. The effect of this action is the player has one craft table.
- **Wooden Planks:** Any four matching wood blocks produce four individual planks. The precondition of this action is the player must have four individual wood blocks. The effect of this action is the player has four individual planks. No crafting table is needed to execute this action.
- **Sticks:** Any two wood planks produce four individual sticks. The precondition of this action is the player must have two individual planks. The effect of this action is the player has four individual sticks. No crafting table is needed to execute this action.
- **Wooden Sword:** Two planks of any kind and one stick produce a wooden sword. The precondition of this action is the player must have two planks and one stick. The effect of this is the player has a wooden sword. The player must have a crafting table to execute this action.

For Experiment 2 The previously shown user-provided `craftItem` NL Domain Background information was given, along with text corresponding to moving, collecting and placing an item in Minecraft.

- **Moving a Player:** This action moves a player from their current location to a new location where an object is located at.The precondition for this action is the location the player is currently located at must be different from the location the player wants to move to where an object is located at, and the grid squares the player is moving between to accomplish this must be connected. The effect of this action is the player is located at a new location where an object is at, an object is located at the location the player is at, and the player is no longer at the location they originally moved from.
- **Collecting an Item:** This action allows a player to collect a single object located at some location, so they have it. The precondition of this action is the player must be in the exact same location as the object they want to collect is. The effect of this action is the player has the object in their inventory, and the object they have collected is no longer located at some location.
- **Placing an Item:** This action allows a player to place a single object they have in their inventory at some location. The precondition for this action is the player must have the object in their inventory and the location they want to place it in can't be the location of a completely different object. The effect of this action is the single object the player placed is now at some location and the player no longer has the single object they placed in their inventory.

C.5 Extraction Stage: Example 'Select Types' Prompt

Here we give an example of prompt P3, which presents Codestral with a list of user-provided types and asks it to choose the ones it thinks are needed to construct a given action. This prompt is constructed using the summary information from prompts P1 and P2, user-provided objects types hierarchy, and the NL domain background information. The example prompt we present is for the `craftSticks` action.

The recipe to craft sticks in Minecraft is: any two wood planks produce four individual sticks. The precondition of this action is the player must have two individual planks. The effect of this action is the player has four individual sticks. No crafting table is needed to execute this action.

If you were asked to create a simple PDDL craftSticks action, which of these types would you need?

The types you can choose from are: player, craftTable, plankBlock, woodBlock, stickItem, woodenSword, woodenPickaxe, gridSquare and some_object. Here, craftTable, plankBlock, woodBlock, stickItem, woodenSword, and woodenPickaxe are all child types of the parent type some_object (i.e. plankBlock woodBlock stickItem woodenSword woodenPickaxe - some_object).

Base the types you select on the following summary of a generic Minecraft action: The player must have access to a crafting table and sufficient resources to craft an item specified by name and quantity. The goal of the craftItem function is to enable a player to create a specific item in Minecraft by using a set of resources and a crafting table, allowing them to acquire the desired quantity of the item. The precondition of the craftItem function is that there must be a crafting table present in the game world for the player to use, as well as sufficient resources available to craft the item, and that the player has access to these resources and can navigate to the crafting table. The effect of the craftItem function is that it will gather the necessary resources and perform the crafting process at the crafting table, resulting in the creation of the specified quantity of the desired item, which becomes available for the player to collect.

Provide your answer within one contiguous code block surrounded by "' symbols. Include only your choice of types, with no other words or code. For example, if the types you selected were animal and bug, you would answer as "'animal, bug"'. Only include the types you absolutely need for this action.

The prompt is generalizable to other domains with the following template:

{NL domain background info}

If you were asked to create a simple PDDL {action name} action, which of these types would you need?

The types you can choose from are: {object types hierarchy}

Base the types you select on the following summary of a generic Minecraft action: {action summary}

Provide your answer within one contiguous code block surrounded by "' symbols. Include only your choice of types, with no other words or code. For example, if the types you selected were animal and bug, you would answer as "'animal, bug"'. Only include the types you absolutely need for this action.

C.6 Extraction Stage: Example 'Select Predicates' Prompt

Here we give an example of prompt P4, which presents Codestral with a list of user-provided predicates and asks it to choose the ones it thinks are needed to construct a given action. This prompt is constructed using the summary information from prompts P1 and P2, user-provided predicate pool, and the NL domain background information. We again present an example prompt for the `craftSticks` action.

If you were asked to create a simple PDDL craftSticks action, what predicates from this list would you need?:

(agent_located_at ?agent - player ?square - gridSquare)
(object_located_at ?item - some_object ?square - gridSquare)
(connected ?current_position - gridSquare ?next_position - gridSquare)
(HasWoodBlock ?agent - player ?wood - woodBlock)
(HasPlankBlock ?agent - player ?plank - plankBlock)
(HasStickItem ?agent - player ?stick - stickItem)
(HasWoodenSword ?agent - player ?sword - woodenSword)
(HasWoodenPickaxe ?agent - player ?pickaxe - woodenPickaxe)
(HasCraftTable ?agent - player ?table - craftTable)

The recipe to craft sticks in Minecraft is: any two wood planks produce four individual sticks. The precondition of this action is the player must have two individual planks. The effect of this action is the player has four individual sticks. No crafting table is needed to execute this action.

Base the predicates you select on the following summary of a generic Minecraft action: The player must have access to a crafting table and sufficient resources to craft an item specified by name and quantity. The goal of the craftItem function is to enable a player to create a specific item in Minecraft by using a set of resources and a crafting table, allowing them to acquire the desired quantity of the item. The precondition of the craftItem function is that there must be a crafting table present in the game world for the player to use, as well as sufficient resources available to craft the item, and that the player has access to these resources and can navigate to the crafting table. The effect of the craftItem function is that it will gather the necessary resources and perform the crafting process at the crafting table, resulting in the creation of the specified quantity of the desired item, which becomes available for the player to collect.

Don't repeat predicates or change the predicates I have given you in any way. You must follow this rule. For example, if the predicates you selected were (IsMammal ?someanimal - animal) and (IsInsect ?somebug - bug) from the list (IsMammal ?someanimal - animal) (IsInsect ?somebug - bug) (IsPlant ?someplant - plant), you would answer as "'(IsMammal ?someanimal - animal), (IsInsect ?somebug - bug)"', where each predicate only appears a single time and has not been changed. For example, (IsMammal ?someanimal - animal) did not become (IsMammal ?someanimal1 - animal) and (IsMammal ?someanimal2 - animal). Provide the predicates you selected within one contiguous code block surrounded by "' symbols. Include only the predicates, with no other words or code."

The prompt is generalizable to other domains with the following template:

If you were asked to create a simple PDDL {action name} action, what predicates from this list would you need?:

{predicate pool}

The recipe to action description based on name in domain is: {NL domain background info}

Base the predicates you select on the following summary of a generic domain action: {action summary}

Don't repeat predicates or change the predicates I have given you in any way. You must follow this rule. For example, if the predicates you selected were (IsMammal ?someanimal - animal) and (IsInsect ?somebug - bug) from the list (IsMammal ?someanimal - animal) (IsInsect ?somebug - bug) (IsPlant ?someplant - plant), you would answer as "'(IsMammal ?someanimal - animal), (IsInsect ?somebug - bug)"', where each predicate only appears a single time and has not been changed. For example, (IsMammal ?someanimal - animal) did not become (IsMammal ?someanimal1 - animal) and (IsMammal ?someanimal2 - animal). Provide the predicates you selected within one contiguous code block surrounded by "' symbols. Include only the predicates, with no other words or code."

C.7 Extraction Stage: Action-Construction Prompt

The final Extraction Stage prompt P5 is constructed using the summary information from prompts P1 and P2, selected types and predicates from prompts P4 and P5 and the user-provided NL domain background information. This prompt instructs Codestral to produce a valid PDDL action. We show an example of this prompt for the craftSticks action.

You are a helpful assistant that is an expert in writing PDDL actions. Your task is to write a PDDL action for crafting sticks in Minecraft.

Base the action you create on the following summary of a generic Minecraft action: The player must have access to a crafting table and sufficient resources to craft an item specified by name and quantity. The goal of the craftItem function is to enable a player to create a specific item in Minecraft by using a set of resources and a crafting table, allowing them to acquire the desired quantity of the item. The precondition of the craftItem function is that there must be a crafting table present in the game world for the player to use, as well as sufficient resources available to craft the item, and that the player has access to these resources and can navigate to the crafting table. The effect of the craftItem function is that it will gather the necessary resources and perform the crafting process at the crafting table, resulting in the creation of the specified quantity of the desired item, which becomes available for the player to collect.

The recipe to craft sticks in Minecraft is: any two wood planks produce four individual sticks. The precondition of this action is the player must have two individual planks. The effect of this action is the player has four individual sticks. No crafting table is needed to execute this action.

Make this action using the Minecraft recipe, and the following types and predicates:

Types: {types LLM selected in prompt #3}

Predicates: {predicates LLM selected in prompt #4}

And follow these rules:

1) All precondition fluents containing objects inside your action must be unique. For example, if (DolphinPod ?dolphin1) and (DolphinPod ?dolphin2) are part of your action's precondition, ?dolphin1 and ?dolphin2 must be distinct objects (i.e. (DolphinPod ?dolphin1) and (DolphinPod ?dolphin2)). Use a not-equal constraint to ensure parameters like this are distinct. This is expressed in PDDL as (not (= ?var1 ?var2)) which ensures that ?var1 and ?var2 are not the same.

2) You are only allowed to use the types and predicates I have given you.

3) Your response should be a single PDDL action. Provide your action within one contiguous code block surrounded by "' symbols. Include only the PDDL action, with no other words or code."

To fully generalize the prompt, the word ‘recipe’ in the sentence ‘Make this action using the Minecraft recipe, and the following types and predicates’ would need to be replaced with something more generic, e.g., “domain background information”. Additionally, ‘crafting sticks’ would need to be replaced by the action name or a phrasal description could be generated by an LLM based on the action name. Otherwise, the prompt is mostly generalizable to other domains with the following template:

> You are a helpful assistant that is an expert in writing PDDL actions. Your task is to write a PDDL action for {short phrasal action description based on action name} in {domain}.
>
> Base the action you create on the following summary of a generic {domain} action: {action summary}
>
> {NL domain background info}
>
> Make this action using the {domain} recipe, and the following types and predicates:
>
> Types: {types LLM selected in prompt #3}
>
> Predicates: {predicates LLM selected in prompt #4}
>
> And follow these rules:
>
> 1) All precondition fluents containing objects inside your action must be unique. For example, if (DolphinPod ?dolphin1) and (DolphinPod ?dolphin2) are part of your action’s precondition, ?dolphin1 and ?dolphin2 must be distinct objects (i.e. (DolphinPod ?dolphin1) and (DolphinPod ?dolphin2)). Use a not-equal constraint to ensure parameters like this are distinct. This is expressed in PDDL as (not (= ?var1 ?var2)) which ensures that ?var1 and ?var2 are not the same.
>
> 2) You are only allowed to use the types and predicates I have given you.
>
> 3) Your response should be a single PDDL action. Provide your action within one contiguous code block surrounded by “‘ symbols. Include only the PDDL action, with no other words or code.”’