NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions

Elliot Gestrin, Marco Kuhlmann, Jendrik Seipp

Linköping University, Sweden
⟨elliot.gestrin, marco.kuhlmann, jendrik.seipp⟩@liu.se

Abstract

Today's classical planners are powerful, but modeling input tasks in formats such as PDDL is tedious and error-prone. In contrast, planning with Large Language Models (LLMs) allows for almost any input text, but offers no guarantees on plan quality or even soundness. In an attempt to merge the best of these two approaches, some work has begun to use LLMs to automate parts of the PDDL creation process. However, these methods still require various degrees of expert input. We present NL2Plan, the first domain-agnostic offline LLM-driven planning system. NL2Plan uses an LLM to incrementally extract the necessary information from a short text prompt before creating a complete PDDL description of both the domain and the problem, which is finally solved by a classical planner. We evaluate NL2Plan on four planning domains and find that it solves 10 out of 15 tasks—a clear improvement over a plain chain-of-thought reasoning LLM approach, which only solves 2 tasks. Moreover, in two out of the five failure cases, instead of returning an invalid plan, NL2Plan reports that it failed to solve the task. In addition to using NL2Plan in end-to-end mode, users can inspect and correct all of its intermediate results, such as the PDDL representation, increasing explainability and making it an assistive tool for PDDL creation.

Introduction

The field of AI planning has developed powerful domain-independent planners such as Fast Downward (Helmert 2006), which can solve planning tasks very efficiently. The plans they generate are sequences of actions that transform the initial state to a goal state (Ghallab, Nau, and Traverso 2004). The most common format for specifying such problems is the Problem Domain Definition Language (PDDL) (McDermott et al. 1998). This is a declarative programming language that can be used to define the available actions, predicates and types, as well as the existing objects, their initial states, and any goal requirements. Specifying the PDDL for the domain-independent planners is, however, a labor-intensive task that requires PDDL-trained users and an understanding of the domain under consideration.

Recent advances regarding large language models (LLMs) have led researchers to develop methods that instead operate directly on natural language descriptions of problems (Valmeekam et al. 2023b). LLMs are models pre-trained on large amounts of textual data and then fine-tuned on specific tasks such as question answering, coding, or solving math problems (Brown et al. 2020). Their ability to plan has been generally poor, however. In particular, LLMs commonly "forget" the long-term effects of actions and their physical requirements even when given domain-specific adaptions and examples (Valmeekam et al. 2023a; Stein et al. 2024; Liu et al. 2023). Nevertheless, LLM-driven online planners (Yao et al. 2023b; Shinn et al. 2023) have been created. These use in-domain examples to select actions one at a time and then receive updates from the environment or simulators, after which they select the next action. Conversely, offline reasoners (Wei et al. 2022; Yao et al. 2023a; Xie et al. 2023a; Hao et al. 2023) have been developed with a focus on high-quality question answering and deduction, as well as domain-dependent offline planners (Hao et al. 2023; Huang et al. 2022). The fact that these systems use natural language texts as input makes them easy to apply even for untrained users and enables many use cases.

In light of the complementary strengths of PDDL- and LLM-based methods, there has been recent work mixing the two approaches in various ways. For example, LLMs have been used to generate parts of the PDDL descriptions (Liu et al. 2023; Dagan, Keller, and Lascarides 2023; Guan et al. 2023; Collins et al. 2022; Zhou et al. 2023; Lyu et al. 2023), to initialize classical planners (Valmeekam et al. 2023b), and to solve PDDL-specified problems (Silver et al. 2022, 2023; Pallagani et al. 2022). Conversely, PDDL descriptions have been used to automatically validate LLM-made plans (Valmeekam et al. 2023b; Zhou et al. 2023). However, to our knowledge, no current approach autonomously creates entire PDDL descriptions from natural text. Instead, each method focuses on generating parts of the PDDL, such as the actions and predicates (Guan et al. 2023), just the goal state (Lyu et al. 2023; Xie et al. 2023b), or both the initial and goal state (Liu et al. 2023; Dagan, Keller, and Lascarides 2023; Collins et al. 2022; Zhou et al. 2023). These methods additionally require structured or domain-dependent inputs, such as other parts of the PDDL and examples from the current domain (Liu et al. 2023; Dagan, Keller, and Lascarides 2023; Collins et al. 2022; Zhou et al. 2023; Lyu et al. 2023) or explicit descriptions of all types and actions (Guan et al. 2023). There are also non-LLM-based methods to generate PDDL autonomously, but these generally use large amounts of text (Huo et al. 2020), examples of successful plans (Gugliermo et al. 2023), or

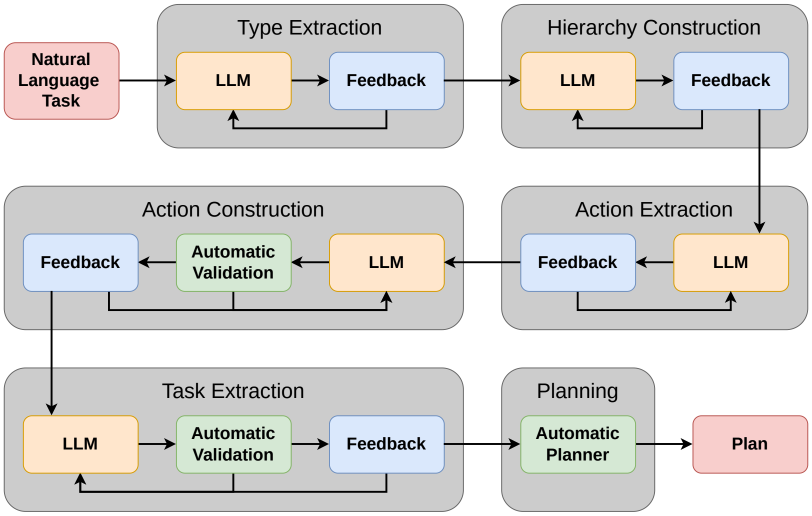


Figure 1: A block scheme of NL2Plan and its six steps. During the Type Extraction step we generate a set of object types, which are then structured into a tree by the Type Hierarchy step. Following this, the Action Extraction step creates a list of natural language action descriptions, which the Action Construction step formalizes in PDDL. Task Extraction is the final LLM-driven step and creates the initial state and goal description. Lastly, the Planning step uses an automatic planner to generate a plan or show that the modeled task is unsolvable. In each LLM-driven step, a human or LLM instance can optionally provide further feedback on the solution. The user only has to interact with NL2Plan to provide the natural language task.

specific input data structures (Malburg, Klein, and Bergmann 2023) and thus are more difficult to apply.

In this paper, we generalize and build upon the existing natural language-to-PDDL systems, adding pre-processing steps and automated common sense feedback to create NL2Plan. NL2Plan is, to our knowledge, the first domain-agnostic offline natural language planning system and uses an LLM to generate complete PDDL descriptions and corresponding plans based on only a few sentences of natural language, without needing any domain-specific adaptations. It employs the LLM to incrementally extract relevant information and to construct the PDDL over five steps: Type Extraction, Hierarchy Construction, Action Extraction, Action Construction, and Task Extraction. Each step optionally utilizes feedback from either a human or another LLM instance to encourage correctness. The Action Construction and Task Extraction steps additionally use automatic validation to enforce correct formatting. In the final step, the PDDL task is then solved by a classical planner, guaranteeing that the final plan is sound for the task model. We illustrate NL2Plan in Figure 1.

To evaluate NL2Plan, we run it on four planning domains, three from Guan et al. (2023) and one from Christen et al. (2023). We find that NL2Plan correctly solves 10 out of 15 tasks, a clear improvement from directly applying an LLM which solves only 2. Furthermore, while many LLM-driven methods are unaware of when they fail and simply return invalid solutions, NL2Plan's use of a classical planner allows it to identify 2 out of 5 failure cases and return "No plan found" instead. This property could be valuable in cases where executing failed plans is expensive, and otherwise, NL2Plan could be automatically re-run on failures to increase the chance of producing valid plans. Additionally, using a PDDL representation allows a user to understand how NL2Plan interprets the task and why it plans in the way it does, making the generated plans explainable and mitigating the black-box nature of applying LLMs. The fact that NL2Plan generates PDDL from simple inputs would also allow it to act as a tool to assist humans in creating domain descriptions for new areas.

Background

In this section, we formally introduce PDDL and describe the prompting techniques used for NL2Plan.

Planning Domain Definition Language (PDDL)

PDDL (McDermott et al. 1998) is the predominant specification language for deterministic planning problems. We assume basic familiarity with the semantics of PDDL and refer to the language definition for details (Fox and Long 2003). We consider "level 1" of PDDL version 2.1, plus action costs. This fragment includes all ADL features (Pednault 1989), such as quantified and conditional effects, and negation, disjunction and quantification in conditions.

For our technical contributions, we only need definitions of the main building blocks of PDDL. A PDDL task consists of a domain description D and a problem specification P. The domain specifies the common properties of the environment: the type hierarchy T, predicate set $\mathcal{P}$, and action schemas A; $D = \langle T, \mathcal{P}, A \rangle$. The problem defines the task specifics: the available objects O, initial state I, and the goal description G; $P = \langle O, I, G \rangle$.

Within domain $D = \langle T, \mathcal{P}, A \rangle$, type hierarchy T specifies which types exist and their relationship. For example, T could specify that the types Truck, Plane, and Vehicle exist and that both Plane and Truck are subtypes of Vehicle.

Each predicate $p \in \mathcal{P}$ is defined over a tuple of arguments, where each argument has a type $t \in T$. By substituting all arguments of a predicate p with objects $o \in O$ of valid types, we obtain a Boolean proposition $p(o_1, \ldots, o_m)$. We call this substitution process *grounding* and the resulting proposition *grounded*. For example, the grounded proposition $at(truck_1, loc_1)$ holds in those states where $truck_1$ is at loc_1. These propositions define the state space of the problem.

Similarly, A is a set of action schemas $a \in A$, each of which in turn is a tuple $a = \langle name(a), pre(a), \mathit{eff}(a)\rangle$. Here, $name(a)$ is the name of action a and $pre(a)$ is its precondition, a first-order logical expression over $\mathcal{P}$ which must be true for a to be executable. Effect $\mathit{eff}(a)$ instead defines what happens when a is executed. Action schemas can then be grounded in the same way as predicates. For example, the grounded action a_{load} which loads a packages p_1 into a truck t_1 at a location l_1 can be defined as: $name(a_{load}) = load$, $pre(a_{load}) = at(p_1, l_1) \wedge at(t_1, l_1)$, $\mathit{eff}(a_{load}) = loaded(p_1, t_1) \wedge \neg at(p_1, l_1)$.

In a problem specification $P = \langle O, I, G\rangle$, set O consists of objects $o_k = \langle n, t\rangle$, where n is the object name and $t \in T$ is its type. Initial state I is a function that assigns a Boolean value to all propositions. Finally, G defines which states are goal states via a logical expression over $\mathcal{P}$. For example, to require package p_1 and p_2 to be at location loc_1 and loc_2, respectively, one can define $G = at(p_1, loc_1) \wedge at(p_2, loc_2)$.

A plan π is then a sequence of m grounded actions $\pi = \langle a_1, \ldots, a_m\rangle$ such that these can be applied in sequence given their preconditions and that this transforms state I into one satisfying G.

Prompting Large Language Models

LLMs are commonly controlled via descriptive inputs, so-called *prompts* (Brown et al. 2020). Various techniques have been developed to improve these prompts and thereby increase the quality of LLM responses. We mainly use two: few-shot prompting (Brown et al. 2020) and chain-of-thought reasoning (Wei et al. 2022).

Few-shot prompting entails adding examples of the desired behavior to the prompt and has been shown to allow LLMs to adapt to various tasks without fine-tuning (Brown et al. 2020). In NL2Plan, each step uses either one or three examples, so-called *one-shot* and *three-shot* prompting respectively.

Chain-of-thought reasoning (CoT) encourages the LLM to reason about its response and to analyze the task step by step, leading to improvements in most tasks for larger LLMs. This can be done either through few-shot examples or as a part of the instructions (Wei et al. 2022; Kojima et al. 2022). We use CoT reasoning for all prompts in NL2Plan.

Schemas for the prompts used can be found in Appendix C. We developed these using GPT-4-1106-preview, a somewhat different LLM than the one used for evaluation (default GPT-4), and a non-testing domain, Logistics (McDermott 2000).

NL2Plan

NL2Plan is to our knowledge the first offline domain-agnostic natural language to plan system. We created it via generalizing existing methods for generating PDDL from natural language, adding pre-processing steps and LLM-based common-sense feedback. The only input needed is a task description in natural language which the system then internally analyses before returning a plan, or identifying that it fails to solve the task. The internal analysis takes the form of an incremental extraction of task-specific information, followed by the formalization thereof into a PDDL domain and problem specification, which lastly is passed to a classical planner. This uses the LLM for its strengths, language comprehension and general world knowledge, while the planning task itself, which LLMs have shown to be poor at, is left to the planner.

The six steps in NL2Plan are the following (see Figure 1):

- **Type Extraction**: The LLM defines which types, such as Truck, Plane, and Vehicle, should be available for the task.
- **Hierarchy Construction**: The LLM organizes the types, for example defining that Truck and Plane are both subtypes of Vehicle.
- **Action Extraction**: The LLM describes which actions should be available in natural language, for example, load_vehicle and fly_plane.
- **Action Construction**: The LLM formalizes the actions in PDDL, dynamically creating the necessary predicates.
- **Task Extraction**: The LLM defines the initial state and goal description.
- **Planning**: The classical planner solves the generated PDDL, returning a plan or proving unsolvability.

Each LLM-based step optionally uses a feedback source, either another LLM instance or a human user. If the feedback source provides advice, a new prompt is formed by concatenating the original prompt, the LLM's first solution, and the feedback. Then, a new solution is generated using the resulting prompt. Such a feedback loop occurs at most once per step. The Action Construction and Task Extraction steps additionally use an automatic validation tool to check that the generated PDDL is valid. Several possible mistakes are checked by the validation tool, see Appendix E for the full lists, and if it finds an error a suggested fix is supplied as feedback in the same way as above. The overall pipeline is visualized in Figure 1 and we showcase illustrative input-output pairs in Figures 2 and 3. Additionally, we share both more complete input-output pairs and the prompt schemas in Appendix C.

Prompting and Feedback

The prompt for the main task in each step is a one-shot CoT prompt consisting of a solution, partially flawed feedback, and a corrected solution. We opted for one-shot CoT as even this results in some steps having very long prompts, and providing more examples would make these longer than GPT-4's context window. While we could provide the simpler steps or the first prompt (when no feedback is available) with more examples, we chose to unify the method. The partially flawed feedback includes both good advice and bad, which is then responded to in the corrected solution. We included flawed feedback due to early testing showing LLMs' tendency to

1: Type Extraction

The AI agent here is a logistics planner that has to plan to transport packages within the locations in a city through a truck and between cities through an airplane. In a city, all the locations are connected. Similarly, cities are directly connected allowing airplanes to travel between them. Also, there is no limit to how many packages a truck or plane can carry.

Available types::
- City: A city contains locations...
- Plane: Planes fly between cities...
- Truck: Trucks travel inside cities...
- Package: Carried by planes and trucks...
- Location: Places within cities where...

2: Hierarchy Construction

- City: A city contains locations...
- Plane: Planes fly between cities...
- Truck: Trucks travel inside cities...
- Package: Carried by planes and trucks...
- Location: Places within cities where...
- Airport: Location where planes...

Hierarchy:
- object: Everything is an object
  - package: Packages are independent.
  - vehicle: Vehicles transport things.
    - truck: Trucks are vehicles.
    - plane: Planes are vehicles.
  - city: Not a location, but contains them.
  - location: Visitable places.

3: Action Extraction

The AI agent here is a logistics planner that has to plan to transport packages...

The available types are:
- object: Everything is an object
  - Package: Carried by planes and...
  - vehicle: Vehicles transport things.
    - truck: Trucks travel inside cities...
    - Plane: Planes fly between cities...

    ...

Actions:
- load: Load a package on a vehicle. Requires...
- drive: Drives a truck within a city. Requires...
- fly: Flies a plane between cities. Requires...

...

4: Action Construction

load
Load a package on a vehicle. Requires that both the package and the vehicle are at the same location. Example: truck1 loads package2 at storage3.

Available predicates:
- (at ?o - object ?l - location): true if the object is at the specified location.

Parameters:
- ?p - package: The package to be loaded
- ?v - vehicle: The vehicle onto which the package is being loaded
- ?l - location: The location where the loading is taking place

Preconditions:
(and ; *Both objects have to be at the location*
    (at ?p ?l) ; *The package is at the location*
    (at ?v ?l) ; *The vehicle is at the location*
)

Effects:
(and ; *All these effects happen*
    (not (at ?p ?l)) ; *The package is no longer at the location*
    (loaded ?p ?v) ; *The package is instead loaded in the vehicle*
)

New Predicates:
- (loaded ?p - package ?v - vehicle): true if the package is in the vehicle.

5: State-Goal Extraction

Currently I've got four packages to ship, 2 in a storage in Paris and the rest in London. Those from London should be sent 1 to Addr1 in Paris and 1 to Addr2 in Berlin. Those from Paris should all be moved to London's storage. The only plane is currently in London, but each city has its own truck and airport.

Available predicates:
- (at ?o - object ?l - location): true if the object is at the specified location.

...

Available Objects:
- p1 p2 p3 p4 - package: The four packages to move
- storeL, storeP, Addr1, Addr2: The locations to ship to and from

...

Initial State:
(at p1 storeP) ; *package 1 is in the Paris storage*
(at p2 storeP) ; *package 2 is in the Paris storage*
...

Goal State:
(and ; *All these are needed*
    (at p1 storeL) (at p2 storeL) ; *Paris packages go to London*
    (at p3 Addr1) (at p4 Addr2) ; *London packages go to Addr1 and Addr2*
)

Figure 2: Illustrative input-output pairs from the NL2Plan steps. For space reasons, we removed large portions of both the inputs and outputs. More complete examples can be found in Appendix C.

be overly agreeable, accepting bad advice from other LLM instances.

The LLM-feedback prompts are all three-shot CoT prompts with checklists. The first two exemplars are task-dependent showcases of flawed inputs, such as a missing required action, followed by answers to the checklist, and finally a list of concrete improvements. The final example is a correct input which is accepted without feedback, after being analyzed according to the checklist, the aim of which is to show feedback does not have to be provided if the input is valid. An example of this is shown in Figure 3 and all checklists are found in Appendix D.

Type Extraction

In the first step, we use the LLM to extract which types should be available in the domain. Notably, explicitly mentioned objects, such as Bob or Paris, should be modeled not with their names but rather with their types, Human and City in this example. The result is a list of types along with descriptive comments. See Figure 2 for an illustrative example and Appendix C for a more complete one.

Hierarchy Construction

Given a list of types, we then let the LLM organize them as sub-types. The LLM first reasons about the provided types, and then places them into a tree structure. It is also able to generate new parent types if needed, such as adding a Vehicle type to organize the requested Plane and Truck types as subtypes thereof. During development, we noticed that the order of the supplied types had a large effect. For simplicity, we supply them in the order generated by the Type Extraction step and rely on the feedback to provide any needed corrections. See Figure 2 for an illustrative example and Appendix C for a more complete one.

Action Extraction

During the Action Extraction step, we make the LLM report which actions are needed for the requested task based on

Feedback Example

Your task is to evaluate if a set of types are correct and sufficent for modelling a given domain.

Domain:
The AI agent here is a logistics planner that has to plan to transport packages within the locations in a city through a truck and between cities through an airplane. In a city, all the locations are connected. Similarly, cities are directly connected allowing airplanes to travel between them. Also, there is no limit to how many packages a truck or plane can carry.

Types:
- city: A city contains locations...
- truck: Trucks travel inside cities...
- package: Carried by trucks...
- house: A location visited by trucks...

1. Are there additional types which are needed to model the domain?
   The domain describes the existence of airplanes, which are not included. As such: Yes.

2. Are additional types needed for organising the type hierarchy?
   The inclusion of both "truck" and "airplan" (see above) motivates the addition of a "vehicle" type. Hence: Yes.

...

My concrete suggestions are:
- Add the following types:
  - airplane: airplanes fly between...
  - vehicle: Trucks and planes which...
- Replace the "house" type with a more general "location" type.
- Update the descriptions of "city" and "package" to reflect the above changes.

Figure 3: An illustrative example of the automatic LLM-driven Type Extraction feedback substep. The checklists vary between NL2Plan steps, see Appendix D.

the defined types and its world knowledge. Each such action is described with a name, description, and usage example. Between each action, the LLM can also reason about which further actions to include. See Figure 2 for an illustrative example and Appendix C for a more complete one.

Action Construction

The Action Construction step is based on work by Guan et al. (2023). In the same way as their work, we make the LLM define a single action at a time by generating its arguments, preconditions, and effects. To this end, the LLM also defines new predicates dynamically for use in the current and subsequent actions. The action is then validated automatically. The validator checks various possible errors such as the use of undefined predicates, unspecified types, and incorrectly placed keywords. We give the full list of properties checked in Appendix E. Only a single property is checked at a time and a total of at most eight error messages are provided, after which the flawed action is accepted. After each action has been defined once, the process restarts and each action is generated a second time with the full list of generated predicates available.

In contrast to previous work, we allow for non-STRIPS actions, such as those that affect each object of a given type, which enables NL2Plan to more easily deal with complex domains. Additionally, we modify the used prompts to be CoT-based and add the feedback substep after the validation to allow for common-sense feedback in addition to syntactical. Due to the previous NL2Plan steps, the user also does not need to explicitly specify the actions and type hierarchy. Lastly, we also finalize this process by pruning the generated types and predicates, keeping only those used in the final iteration. See Figure 2 for an illustrative example and Appendix C for a more complete one.

Task Extraction

In the final LLM-based step, we generate the PDDL problem specification similarly to Liu et al. (2023) and Collins et al. (2022). This includes the objects, the initial state, and the goal condition. All of these are generated in a single call to the LLM. Following the generation, the validator then automatically generates feedback similarly to the Action Construction step. It checks the objects first, then the initial state, and lastly the goal condition. Once it encounters errors in one of these, it suggests fixes to all errors for that section and the entire solution is regenerated. At most 8 such validations can occur, including both before and after receiving any feedback, after which the output is accepted in the flawed state. The full list of checked properties is found in Appendix E. Following this, the feedback substep is performed. See Figure 2 for an illustrative example and Appendix C for a more complete one.

In contrast to existing methods, NL2Plan accepts natural, unstructured descriptions of the initial state and goal conditions. This is in contrast to previous work, which generated the initial state and goal programmatically and presented it in a structured manner while also using hand-crafted PDDL domains. Moreover, our addition of CoT, syntax validation, and automatic feedback are all novel in this context. Lastly, NL2Plan is the first robust domain-agnostic system. While Liu et al. (2023) also introduced a domain-independent zero-shot version of LLM+P, it failed to solve any task.

Planning

Lastly, the planner solves the generated PDDL task. If it does not find a plan, NL2Plan concludes that the modeled PDDL task can not be solved and returns "No plan found".

Experiments

We now describe our experiments for evaluating NL2Plan.

Baseline

Since NL2Plan is the first domain-independent natural language planner, it is difficult to find a suitable baseline algorithm. The LLM-driven reasoning techniques such as CoT+SC (Wang et al. 2023), ToT (Yao et al. 2023a), and Guided Decoding (Xie et al. 2023a) are developed primarily for question-answering deduction (Huang and Chang 2023). While Liu et al. (2023) introduced a version of ToT adapted for planning, it uses in-domain examples, requires explicit action specifications, and is outperformed by both zero-shot and one-shot chain-of-thought methods. Similarly, RAP (Hao et al. 2023) is a deduction method which was also used for planning in the Blocksworld domain, but this required handcrafted domain-specific adaptations and examples. The procedure introduced by Huang et al. (2022) is also an LLM-driven method focused on planning, but uses both in-domain examples and embeddings of all possible grounded actions. The methods using classical planners such as LLM+P (Liu et al. 2023) and L+P (Collins et al. 2022) instead require at least the PDDL domain description as input. Therefore, we follow the evaluation approach of other works (Stein et al. 2024; Liu et al. 2023) and use zero-shot chain-of-thought reasoning as a baseline.

Zero-shot chain-of-thought reasoning was introduced by Kojima et al. (2022) and involves appending the statement “Let’s think step by step” to the task, which triggers chain-of-thought reasoning and improves model performance on various tasks, similarly to the original few-shot chain-of-thought implementation of Wei et al. (2022). The prompt we use and an example response can be found in Appendix G.

Domains

We use four domains to evaluate NL2Plan:

- Blocksworld (Slaney and Thiébaux 2001) is an international planning competition (IPC) domain where a robot stacks blocks on a table. It can only lift a single block at a time and only the topmost block of any stack.
- Tyreworld is another IPC domain. It models an agent using tools to change tyres on cars.
- Household (Guan et al. 2023) is inspired by the ALFWORLD (Shridhar et al. 2021) and VirtualHome (Puig et al. 2018) domains. It involves a single-armed robot operating in a house with a wide array of available actions and objects.
- Independent Set Reconfiguration (ISR) (Christen et al. 2023) is the task of transforming a given independent subset of a graph to another such subset by swapping nodes, while keeping each intermediary set independent.

We include ISR since the PDDL descriptions of Blocksworld and Tyreworld as well as the one of the Household inspiration ALFWORLD (VirtualHome is not defined in PDDL) likely have been included in GPT-4’s training data (OpenAI 2024), whereas the PDDL formulation of ISR probably is too recent and niche to have been included. A fifth domain, Logistics (McDermott 2000), was the only one used during the development of NL2Plan prompts. This was done to evaluate other domains as unbiased and fairly as possible.

Experiment Setup

To avoid accidentally tailoring the domain descriptions to our approach, we use domains from related works and only remove information from the original natural language descriptions. The descriptions for the Blocksworld, Tyreworld, and Household domains are taken from Guan et al. (2023) with the type definitions and explicit action specifications removed. The description for ISR stems from the introduction of the paper that introduced its PDDL formulation, Christen et al. (2023). All domain descriptions can be found in Appendix A. For each domain, we generate three problems: one easy task requiring at least four actions for a successful plan, one medium task requiring eight, and one hard task requiring twelve. For these, we manually describe both the initial state and the goal criteria. We provide these problem formulations in Appendix B.

For the classical planner, NL2Plan uses the first iteration of LAMA (Richter and Westphal 2010), implemented within Scorpion (Seipp, Keller, and Helmert 2020). Both NL2Plan and the baseline CoT method use GPT-4 (OpenAI 2024) as the underlying LLM.

To reduce LLM usage and cost, we only apply the entirety of NL2Plan to the easy task within each domain. For the medium and hard tasks, we instead re-use the domain description generated for the easy task, performing only the Task Extraction and Planning steps. In the Tyreworld and Household domains, however, the medium and hard tasks require types, predicates, and actions not needed for the easy task. Therefore, we manually add these without modifying the existing domain description.

Additionally, to showcase how NL2Plan can be controlled via prompting of the input task description we introduce a variant of the ISR domain where we specifically request that the “reconfiguration” action be modeled in two stages, alternating between an “add” and a “remove” action, imitating the domain model by Christen et al. (2023). We share the modified domain description in Appendix A.

For the the Zero-Shot CoT baseline, we provide the LLM with the domain and task descriptions, request a plan, and append “Let’s think step by step”. We share the exact prompt in Appendix G.

Results

Table 1 summarizes the performance of Zero-Shot CoT and NL2Plan. The exact tasks and plans can be found in Appendix B and two examples of domain descriptions generated by NL2Plan are shown in Appendix F. Before discussing the results, we reiterate that the NL2Plan prompts were developed using only a non-testing domain, Logistics, and a different LLM version, GPT-4-1106-preview. This development approach, combined with how we selected the domain descriptions, makes the experiment representative for applying NL2Plan to new, unseen domains and tasks.

		Zero-Shot CoT		NL2Plan	
Blocksworld	Easy	✓		✓	
	Med.	✗	Moves stack of blocks.	✓	
	Hard	✗	Moves lower block.	✓	
Tyreworld	Easy	∼	Loosens already loose nut.	✓	
	Med.	✓		✓	
	Hard	✗	Fetches non-existent jack.	✗	Incorrectly specifies boot as open. Also domain flaw.
Household	Easy	∼	Places mug on closed cabinet.	✓	
	Med.	✗	Opens container inside fridge.	✗	Fails to use involved predicates. No plan found.
	Hard	✗	Picks up non-pickupable pizza.	✗	Refers to previous iteration. No plan found.
ISR	Easy	✗	Invalid node replacement.	✓	
	Med.	✗	Invalid node replacement.	✗	Defined directed graph. Also domain flaw.
	Hard	✗	Invalid node replacement.	✗	Defined directed graph. Also domain flaw.
ISR Assisted	Easy	✗	Invalid node placement.	✓	
	Med.	✗	Invalid node placement.	✓	
	Hard	✗	Invalid node placement.	✓	

Table 1: Summary of the generated plans. Check marks denote successful plans, crosses denote failed plans, and $\sim$ denotes questionable plans. In the latter two cases, we describe the flaw. For the exact tasks and plans, see Appendix B.

Zero-Shot CoT Results

Planning with Zero-Shot CoT only leads to a successful plan for 2 out of 15 tasks, with another 2 plans being flawed. As such, our results join a line of work showing that directly applying LLMs to planning tasks leads to poor performance (e.g., Valmeekam et al. 2023a; Stein et al. 2024; Valmeekam et al. 2023b). The main failure reason appears to be invalid domain modeling, with Zero-Shot CoT using actions that violate the domain constraints. For example, lifting an entire stack of blocks at once in Blocksworld or always replacing the initial independent set sequentially in the ISR domains. The same flaw is seen for the questionable plans, where it chooses actions that contradict the domain-intent but which, depending on the particular robot used, might be applicable.

NL2Plan Results

Plans NL2Plan shows much higher robustness than Zero-Shot CoT. It successfully solves 10 of the 15 tasks, a superset of those solved by Zero-Shot CoT (including those with questionable plans). The primary cause of failure for NL2Plan is incorrect task modeling, for example, only defining the ISR "neighbor" predicate in a one-directional manner. However, for the Tyreworld and ISR tasks NL2Plan would have failed regardless due to flawed domain modeling. In Tyreworld the domain model required a jack to lower a car, rather than returning one upon doing so, and in the ISR domain a node could incorrectly never be replaced by its neighbors. These functionalities were not needed for the easy task which the domains were created for, but were necessary for these harder tasks. As such, even given a correct problem description NL2Plan would have returned "No plan found" for these.

Automatic Feedback The automatic feedback substep used in NL2Plan varies in its usefulness. The first step, Type Extraction, generates several invalid types for both the Blocksworld and the ISR domains, including defining both actions and predicates as types, and the feedback successfully leads to removing 88.2% of them. In contrast, during the Hierarchy Construction and Action Extraction steps, the initial solutions are always accepted as is or with ineffectual feedback. During the Action Construction step, feedback is returned for 52.8% of actions generated, though 21.1% of said advice is incorrect and would worsen the action if accepted. Lastly, in the Task Extraction step, feedback is returned for 53.3% of the tasks, of which 25% is at least partially harmful. The quality of this feedback could likely be improved with superior prompting, but it is already a net benefit leading to more sensible and correct solutions.

Automatic Validation The automatic validation is similarly varied in usage. During Action Construction, such an error occurred for 77.8% of actions, with the majority caused by invalid predicate arguments. However, in the Task Extraction step errors were only raised for two tasks (13.3%), the medium Tyreworld and the hard Household tasks, occurring once and five times respectively. The most common error was reusing type names for objects, followed by referring to undefined objects. While the validation is not needed each time Task Extraction is performed or an action is generated, it is still valuable since each case when it is needed would have otherwise led to malformed PDDL and parsing failures during the Planning step.

		NL2Plan				
	Zero-Shot CoT	Step 1	Step 2	Step 3	Step 4	Step 5
Blocksworld	314	5669	2640	5261	50866	7409
Tyreworld	612	3475	3012	5549	88738	12315
Household	841	4424	4012	7089	160205	23334
ISR	647	5657	4638	8412	29793	11676
ISR Assisted	735	6160	4884	5411	30567	17443
Average	630	5077	3837	6344	72034	14435

Table 2: The token usage of both Zero-Shot CoT and NL2Plan. Where applicable, the number of tokens is averaged over all tasks and rounded to the nearest integer. Input and output tokens are summed.

Domain Modeling

In our experiments, most of Zero-Shot CoT's failures appear to be caused by an inability to assess action preconditions. Similar results were also found by Liu et al. (2023). However, NL2Plan can use many of these actions correctly despite utilizing the same LLM. We believe this to be caused by GPT-4, the LLM used, being better at "reasoning" about the actions when they are considered in isolation. This insight could prove useful for future work, showing that using the LLM to analyze actions separately ahead of planning can increase plan quality through improved domain modeling.

Invalid Plans and Stochasticity

Whenever Zero-Shot CoT fails it always returns an invalid plan. However, NL2Plan instead returns "No plan found" in two of its five failure cases, and would have done so on all of them given correctly generated problem descriptions. This property of NL2Plan to identify its failures could be valuable in cases where executing invalid plans is expensive. Additionally, our early testing on the Logistics domain revealed that NL2Plan is stochastic, generating different domain and task descriptions for the same input even with the temperature of the LLM set to zero. Similarly, the reasoning of Zero-Shot CoT is stochastic, but the plan itself seems to remain static. As such, in the event of NL2Plan failing to solve a task, it could be automatically re-run, possibly generating a valid plan the second time.

Token Usage

As shown in Table 2, the use of multiple steps, repeated iterations, and long prompts in NL2Plan leads to a large increase in token usage compared to Zero-Shot CoT. This in turn entails increased cost, runtime, and energy usage. Hence, Zero-Shot CoT using fewer tokens than NL2Plan is an advantage. For NL2Plan this usage is dominated by the Action Construction step (Step 4) which on average uses 70.8% of the tokens. As such, any improvements to the Action Construction step would likely significantly reduce the cost of NL2Plan. To further reduce token usage, the NL2Plan pipeline could be started from the Task Extraction step when solving a novel task from an already seen domain, re-using the previous PDDL domain description, lowering token usage and cost to approximately 14.2% of that for a new domain. If no plan is found, the entire pipeline could then be run to generate a domain description better adapted to the new task.

Explainability

The fact that NL2Plan generates intermediate PDDL domain descriptions and problem specifications means that the method is more explainable than Zero-Shot CoT and similar LLM-driven approaches. A user can read these descriptions and understand why the method chose a certain plan, reducing the black-box nature of LLMs. Furthermore, any step could be inspected and controlled further by allowing the human user to be the feedback source rather than an LLM. This would likely also lead to both superior domain models and planning performance.

PDDL Creation Assistance

There are also other use cases of NL2Plan than direct planning. For example, practitioners could use it to easily synthesize PDDL descriptions, which they then manually edit to their liking. While the descriptions might have flaws, such as the inability to replace a node with its neighbors in the ISR domains, a practitioner could simply fix these by hand. This semi-automated approach has the potential to reduce the workload of applying classical planners to new areas. This is further illustrated by the ISR Assisted domain where the "user" specified how the domain should be modeled. These instructions were followed by NL2Plan and led to both superior plans and a domain model adapted to the user's liking.

Concurrent Work

During our development of NL2Plan, Agarwal and Sreepathy (2024) introduced TIC which is a novel approach for generating PDDL problem specifications, and as such performs the same role as our Task Extraction step. TIC incrementally generates a logical representation of the problem, solves it with a logical reasoner and compiles the solution into PDDL. While they use some manually added domain-specific rules, TIC reaches high accuracy across seven planning domains (including Blocksworld and Tyreworld).

Conclusions and Future Work

We presented NL2Plan, the first offline, domain-independent, natural-language-to-plan system. It is a multi-step system combining an LLM and a classical planner to robustly generate plans from only short natural language descriptions via intermediate PDDL domain and task descriptions. Additionally, we showed that NL2Plan outperforms planning directly with LLMs via Zero-Shot CoT, while also increasing explainability, identifying 40% of its failure cases, and being useful as an assistive PDDL-creation tool.

In the future, we intend to build on NL2Plan. The current version is purely LLM-driven and we want to include further step-specific external tools, such as TIC. Additionally, we aim to reduce the token usage of NL2Plan by more efficient prompting and by solving several tasks from the same domain in parallel, combining the first four NL2Plan steps and only performing the Task Extraction step on a per-task basis.

References

Agarwal, S.; and Sreepathy, A. 2024. TIC: Translate-Infer-Compile for accurate 'text to plan' using LLMs and logical intermediate representations. arXiv:2402.06608.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language Models are Few-Shot Learners. In *Proc. NeurIPS*, 1877–1901.

Christen, R.; Eriksson, S.; Katz, M.; Muise, C.; Petrov, A.; Pommerening, F.; Seipp, J.; Sievers, S.; and Speck, D. 2023. PARIS: Planning Algorithms for Reconfiguring Independent Sets. In *Proc. ECAI 2023*, 453–460.

Collins, K. M.; Wong, C.; Feng, J.; Wei, M.; and Tenenbaum, J. B. 2022. Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks. arXiv:2205.05718.

Dagan, G.; Keller, F.; and Lascarides, A. 2023. Dynamic Planning with a LLM. arXiv:2308.06391.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20: 61–124.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning. arXiv:2305.14909.

Gugliermo, S.; Schaffernicht, E.; Koniaris, C.; and Saffiotti, A. 2023. Extracting Planning Domains from Execution Traces: a Progress Report. In *ICAPS 2023 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Hao, S.; Gu, Y.; Ma, H.; Hong, J. J.; Wang, Z.; Wang, D. Z.; and Hu, Z. 2023. Reasoning with Language Model is Planning with World Model. arXiv:2305.14992.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Huang, J.; and Chang, K. C.-C. 2023. Towards Reasoning in Large Language Models: A Survey. arXiv:2212.10403.

Huang, W.; Abbeel, P.; Pathak, D.; and Mordatch, I. 2022. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. *arXiv preprint arXiv:2201.07207*.

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.

Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2022. Large Language Models are Zero-Shot Reasoners. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 22199–22213. Curran Associates, Inc.

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.

Lyu, Q.; Havaldar, S.; Stein, A.; Zhang, L.; Rao, D.; Wong, E.; Apidianaki, M.; and Callison-Burch, C. 2023. Faithful Chain-of-Thought Reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, 305–329. Association for Computational Linguistics.

Malburg, L.; Klein, P.; and Bergmann, R. 2023. Converting semantic web services into formal planning domain descriptions to enable manufacturing process planning and scheduling in industry 4.0. *Engineering Applications of Artificial Intelligence*, 126: 106727.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

McDermott, D. M. 2000. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2): 35.

OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Pallagani, V.; Muppasani, B.; Murugesan, K.; Rossi, F.; Horesh, L.; Srivastava, B.; Fabiano, F.; and Loreggia, A. 2022. Plansformer: Generating Symbolic Plans using Transformers. arXiv:2212.08681.

Pednault, E. P. D. 1989. ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus. In *Proc. KR 1989*, 324–332.

Puig, X.; Ra, K.; Boben, M.; Li, J.; Wang, T.; Fidler, S.; and Torralba, A. 2018. VirtualHome: Simulating Household Activities via Programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39: 127–177.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *JAIR*, 67: 129–167.

Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 8634–8652. Curran Associates, Inc.

Shridhar, M.; Yuan, X.; Côté, M.-A.; Bisk, Y.; Trischler, A.; and Hausknecht, M. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proc. ICLR*.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L. P.; and Katz, M. 2023. Generalized Planning in PDDL Domains with Pretrained Large Language Models. arXiv:2305.11014.

Silver, T.; Hariprasad, V.; Shuttleworth, R. S.; Kumar, N.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. PDDL Planning with Pretrained Large Language Models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Slaney, J.; and Thiébaux, S. 2001. Blocks World revisited. *AIJ*, 125(1–2): 119–153.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2024. AutoPlanBench: Automatically generating benchmarks for LLM planners from PDDL. arXiv:2311.09830.

Valmeekam, K.; Marquez, M.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2023a. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. arXiv:2206.10498.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023b. On the Planning Abilities of Large Language Models - A Critical Investigation. In *Advances in Neural Information Processing Systems*, volume 36, 75993–76005. Curran Associates, Inc.

Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; and Zhou, D. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural*

Information Processing Systems, volume 35, 24824–24837. Curran Associates, Inc.

Xie, Y.; Kawaguchi, K.; Zhao, Y.; Zhao, X.; Kan, M.-Y.; He, J.; and Xie, Q. 2023a. Self-Evaluation Guided Beam Search for Reasoning. arXiv:2305.00633.

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

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T. L.; Cao, Y.; and Narasimhan, K. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629.

Zhou, Z.; Song, J.; Yao, K.; Shu, Z.; and Ma, L. 2023. ISR-LLM: Iterative Self-Refined Large Language Model for Long-Horizon Sequential Task Planning. *arXiv preprint arXiv:2308.13724*.

Appendix

A Domain Descriptions

Blocksworld

The AI agent here is a mechanical robot arm that can pick and place the blocks. Only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved.

Tyreworld

The AI agent here is a robot that has to replace a flat tyre with a spare one. This involves fetching the tools (wrench, jack, pump) from the boot, undoing the nuts on the flat tyre, jacking up the (appropriate) hub(s), removing the tyre, doing up the spare one, etc. Note that there is no restriction on how many objects the AI agent (i.e., the robot) can carry. Also note that each hub has only one nut.

Household

The AI agent here is a household robot that can navigate to various large and normally immovable furniture pieces or appliances in the house to carry out household tasks. Note that the robot has only one gripper, so (a) it can only hold one object; (b) it shouldn't hold any other irrelevant objects in its gripper while performing some manipulation tasks (e.g., opening a drawer or closing a window); (c) operations on small household items should be carried out on furniture with a flat surface to get enough space for manipulation. In this domain, the locations of the robot and small household items (e.g., apples, oranges, bowls, lunch boxes or lamps) are determined by large and normally immovable furniture pieces or appliances.

Independent Set Reconstruction (ISR)

An independent set of a graph is a subset of its vertices such that no two vertices of the subset share an edge. Reconfiguring an independent set means replacing one vertex in the subset with another one such that the new subset is still an independent set. The ISR problem is to find a sequence of such reconfiguration steps to reach a given target configuration from a given start configuration. Model the reconfiguration with two actions, "remove" and "add". "remove" cannot be taken if the last action taken was a "remove" action, and conversly "add" can only be taken if the last action taken was a "remove".

Figure A1: For Blocksworld, Tyreworld, and Household the domain descriptions are shortened versions of those from Guan et al. (2023) and for ISR the description comes from the introduction of Christen et al. (2023). The blue section of the ISR description is our addition, creating the ISR Assisted domain.

B Tasks and Generated Plans

Blocksworld Tasks

	Task	Zero-Shot CoT	NL2Plan
Simple - Original optimal plan is 4 actions	There are four blocks currently. The blue block is on the red which is on the yellow. The yellow and the green are on the table. I want the red on top of the green.	- The AI robot arm should pick up the blue block. - The AI robot arm should place the blue block on the table. - The AI robot arm should then pick up the red block. - The AI robot arm should place the red block on top of the green block.	- pick_block blue red - place_block_on_table blue table1 - pick_block red yellow - place_block_on_block red green
Medium - Original optimal plan is 8 actions	There are four blocks in two piles. The first has blue on red and the second has green on yellow. I want a single stack ordered: red, green, blue, yellow.	- Move the blue block from the red block and place it on the table. - Move the green block from the yellow block and place it on the table. - Move the blue block and place it on the green block. - Move the yellow block and place it on the blue block. - Finally, move the stack of yellow on blue on green and place it on the red block.	- pick_block blue red - place_block_on_table blue table1 - pick_block green yellow - place_block_on_block green red - pick_block blue table1 - place_block_on_block blue green - pick_block yellow table1 - place_block_on_block yellow blue
Hard - Original optimal plan is 12 actions	I've got five blocks of three colours and want to stack them in a single pile such that the red blocks are at the bottom, the blue are on those and the green are on top. Currently they're in two stacks, (red, blue, green) and (green, red) ordered from the bottom up.	- Move the top green block from the second stack (green, red) and place it on the table. - Move the red block from the second stack and place it on top of the red block in the first stack (red, blue, green). - Move the green block from the top of the first stack (red, blue, green) and place it on the table next to the other green block. - Move the blue block from the top of the first stack (red, red) and place it on the table. . . .	- pick_block green1 blue - place_block_on_table green1 table1 - pick_block blue red1 - place_block_on_block blue green1 - pick_block red2 green2 - place_block_on_block red2 red1 - pick_block blue green1 - place_block_on_block blue red2 - pick_block green1 table1 - place_block_on_block green1 blue - pick_block green2 table1 - place_block_on_block green2 green1

Figure A2: The tasks and plans for the Blocksworld domain. Note that CoT action descriptions and NL2Plan names have been shortened. Successful plans are marked in green. Failed plans and their first invalid action are marked in red. Questionable plans that likely differ from user intent or similar are marked in orange.

Tyreworld Tasks

Simple - Original optimal plan is 4 actions

Task

The robot is already holding all the tools and has a whole, but uninflated, wheel. The flat tire is already loosened and jacked up, but still left on the hub. Safely replace the flat tire with the whole one and prepare it.

Zero-Shot CoT

- Ensure that the vehicle is in a safe and stable position.
- Use the wrench to fully loosen the nut on the hub of the flat tire.
- Remove the flat tire from the hub.
- Place the new, uninflated tire onto the hub.
- Use the wrench to tighten the nut onto the hub.
- Use the pump to inflate the tire
- Lower the vehicle from the jack
- Finally, check the tightness of the nut
- Return the tools to the boot.
- Perform a final check to ensure the replaced tire is properly inflated and securely attached

NL2Plan

- inflate_spare_tyre spare_tyre1 pump1
- attach_spare_tyre wrench1 spare_tyre1 hub1
- remove_flat_tyre flat_tyre1 hub1 wrench1

Medium - Original optimal plan is 8 actions

Task

The robot currently only has a wrench, but the other tools and a spare (already inflated) tyre are in the open boot. The back left wheel is flat, and still attached. It should be safely replaced.

Zero-Shot CoT

- Identify the Tools and Spare Tyre
- Fetch the Jack, Pump and Spare Tyre
- Position the Jack
- Raise the Vehicle
- Loosen the Nut
- Remove the Flat Tyre
- Install the Spare Tyre
- Tighten the Nut
- Lower the Vehicle
- Check the Tyre Pressure
- Return the Tools and Old Tyre

NL2Plan

- collect my_jack my_boot
- lift_car my_hub my_jack
- remove_flat_tyre my_flat_tyre my_hub my_wrench
- collect my_spare_tyre my_boot
- attach_spare_tyre my_wrench my_spare_tyre my_hub

Hard - Original optimal plan is 12 actions

Task

The car is already in the process of having wheels changed, so the front left hub is raised with the jack and its wheel removed. I want you to remove and replace the back left tyre. You'll find an uninflated tyre, the wrench and a pump in the closed trunk.

Zero-Shot CoT

- Open the Trunk
- Retrieve the Wrench, Jack, Pump and Spare Tyre
- Inflate the Spare Tyre
- Position the Jack
- Raise the Car
- Remove the Flat Tyre
- Install the Spare Tyre
- Lower the Car
- Tighten the Nut
- Fully Lower the Car
- Check the Tyre Pressure
- Return the Tools
- Final Check

NL2Plan

- collect wrench1 boot1
- collect spare_tyre1 boot1
- collect pump1 boot1
- inflate_spare_tyre spare_tyre1 pump1
- lift_car hub_back_left jack1
- collect jack1 boot1
- attach_spare_tyre wrench1 spare_tyre1 hub_back_left
- remove_flat_tyre flat_tyre1 hub_back_left wrench1
- lower_car boot1 hub_back_left jack1

[NL2Plan failed as the generated PDDL problem description specified that the boot was already open. This was caused by incorrect LLM-driven feedback. Furthermore, the generated "lower_car" action requires and consumes a jack, making the intended task impossible.]

Figure A3: The tasks and plans for the Tyreworld domain. Note that CoT action descriptions and NL2Plan names have been shortened. Successful plans are marked in green. Failed plans and their first invalid action are marked in red. Questionable plans that likely differ from user intent or similar are marked in orange. Note that we allow for tightening the nuts either on the ground or while jacked up, since both are generally recommended.

Household Tasks

Simple - Original optimal plan is 4 actions

Task

Please toggle lamp_1 on and humidifier_1 off starting from the following state:

- cabinet_1 is not opened
- lamp_1 in/on side_table_1
- lamp_1 is turned off
- humidifier_1 in/on side_table_1
- humidifier_1 is turned on
- robot at cabinet_1
- robot is holding mug_1

[Unnecessary state information omitted]

Zero-Shot CoT

- Place mug 1 on cabinet 1.
- Navigate to side table 1.
- Toggle lamp 1 on.
- Toggle humidifier 1 off.
- Await further tasks.

NL2Plan

- navigate robot1 cabinet1 side_table1
- put_down robot1 mug1 side_table1
- turn_off robot1 humidifier1 side_table1
- turn_on robot1 lamp1 side_table1

Medium - Original optimal plan is 8 actions

Task

Your task is to transfer potato_1 from a lunch box to plate_1 and close the fridge after yourself.
This is the current state:

- fridge_1 is opened
- lunch_box_2 in/on fridge_1
- plate_1 in/on dining_table_1
- lunch_box_2 is not opened
- potato_1 in/on lunch_box_2
- potato_1 is not sliced
- robot at cabinet_2
- robot is not holding anything

[Unnecessary state information omitted]

Zero-Shot CoT

- Navigate from cabinet_2 to fridge_1.
- Once at fridge_1, the robot should open lunch_box_2.
- After opening lunch_box_2, the robot should pick up potato_1.
- With potato_1 in its gripper, the robot should navigate from fridge_1 to dining_table_1.
- Once at dining_table_1, the robot should place potato_1 on plate_1.
- After placing potato_1 on plate_1, the robot should navigate back to fridge_1.
- Once at fridge_1, the robot should close fridge_1.

NL2Plan

[No plan found.]

[NL2Plan failed as the generated PDDL problem failed to use the “can_be_opened” predicate in conjunction with “closed” and “open” predicates. Additionally, it modeled “plate_1” as a “small_object” instead of as a “container” and did not specify the table as a “flat_surface”.]

Hard - Original optimal plan is 12 actions

Task

I want you to heat pizza_1 with pan_2.
The following are the current objects and their state:

- drawer_1 is opened
- cabinet_2 is not opened
- pan_1 in/on stove_burner_1
- pan_2 in/on dining_table_1
- pizza_box_1 in/on cabinet_2
- pizza_box_1 is not opened
- pizza_1 in/on pizza_box_1
- pizza_1 is not pickupable
- robot at drawer_1
- robot is holding mug_1

[Unnecessary state information omitted]

Zero-Shot CoT

- Navigate to a flat surface, such as dining_table_1, and place mug_1 on it.
- Navigate to cabinet_2 where pizza_box_1 is located.
- Open pizza_box_1 to access pizza_1.
- Pick up pizza_1 from the pizza_box_1.
- Navigate to dining_table_1 where pan_2 is located.
- Place pizza_1 in pan_2.
- Pick up pan_2 with pizza_1 in it.
- Navigate to the stove and place pan_2 on stove_burner_1.
- Turn on the stove to heat the pizza.
- After a suitable amount of time, the robot should turn off the stove.
- Pick up pan_2 from the stove.
- Navigate back to dining_table_1 and place pan_2 with the heated pizza on it.
- The task is now complete. The robot should wait for the next command.

NL2Plan

[No plan found.]

[NL2Plan failed as the Type Extraction step referred to previous solutions while correcting for feedback with statements such as ‘The state of the furniture, the location of the small items, and the state of the switchable items remain the same.“. Due to this, large amounts of the initiation were discarded.]

Figure A4: The tasks and plans for the Household domain. Note that CoT action descriptions and NL2Plan names have been shortened. Successful plans are marked in green. Failed plans and their first invalid action are marked in red. Questionable plans that likely differ from user intent or similar are marked in orange.

ISR Tasks

Simple - Original optimal plan is 4 actions

Task

The current graph has the following neighbors: - (1,2)
- (2,3)
- (3,4)
- (4,5)
- (5,6)
- (6,7)
- (7,8)
- (7,9)
- (9,5)

And the initial independent set is (4,6,9). Transform it to (3,5,7)

Zero-Shot CoT

- Replace 4 with 3 to get (3,6,9)
- Replace 6 with 5 to get (3,5,9)
- Replace 9 with 7 to get (3,5,7)

NL2Plan

- add_vertex_to_set vertex1 vertex4
- add_vertex_to_set vertex3 vertex1
- add_vertex_to_set vertex1 vertex6
- add_vertex_to_set vertex5 vertex1
- add_vertex_to_set vertex1 vertex8
- add_vertex_to_set vertex7 vertex1

Medium - Original optimal plan is 8 actions

Task

So, I want to go from the set (C,E,G,K,M,O,Q) to (D,F,H,J,L,N,P) in a graph with the following neighbors:
- (A,B)
- (B,C)
- (C,D)

...

Zero-Shot CoT - 14 Actions

- Replace C with D.
- Replace E with F.
- Replace G with H.
- Replace K with J.
- Replace M with L.
- Replace O with N.
- Finally, replace Q with P.

NL2Plan

- add_vertex_to_set vertexh vertexc
- add_vertex_to_set vertexb vertexk
- add_vertex_to_set vertexj vertexb
- add_vertex_to_set vertexb vertexg

...

[NL2Plan failed as the problem specification only used the "neighbors" predicate one way, which the generated domain did not support. Had the problem been correct, it would have failed with "No plan found" due to the domain model not allowing replacement with neighbors.]

Hard - Original optimal plan is 12 actions

Task

The graph has the following neighbors:
(A, E)
(B, C)
(B, F)
...

And I'm starting with the set (A,C,F,H,L,N,P,R,S,T,V) and want to end with the set (B,D,E,I,K,M,O,Q,U,X,Z).

Zero-Shot CoT - 8.5 Actions

- Replace vertex A with vertex B.
- Replace vertex C with vertex D.
- Replace vertex F with vertex E.
- Replace vertex H with vertex I.
- Replace vertex L with vertex K.
- Replace vertex N with vertex M.
- Replace vertex P with vertex O.
- Replace vertex R with vertex Q.
- Replace vertex S with vertex U.
- Replace vertex T with vertex X.
- Finally, replace vertex V with vertex Z.

NL2Plan

- add_vertex_to_set x a
- add_vertex_to_set d c
- add_vertex_to_set z x
- add_vertex_to_set x d

...

[NL2Plan failed as the problem specification only used the "neighbors" predicate one way, which the generated domain did not support. Had the problem been correct, it would have failed with "No plan found" due to the domain model not allowing replacement with neighbors.]

Task Graphs

Simple

```
          6
         / \
1-2-3-4-5   7-8
         \ /
          9
```

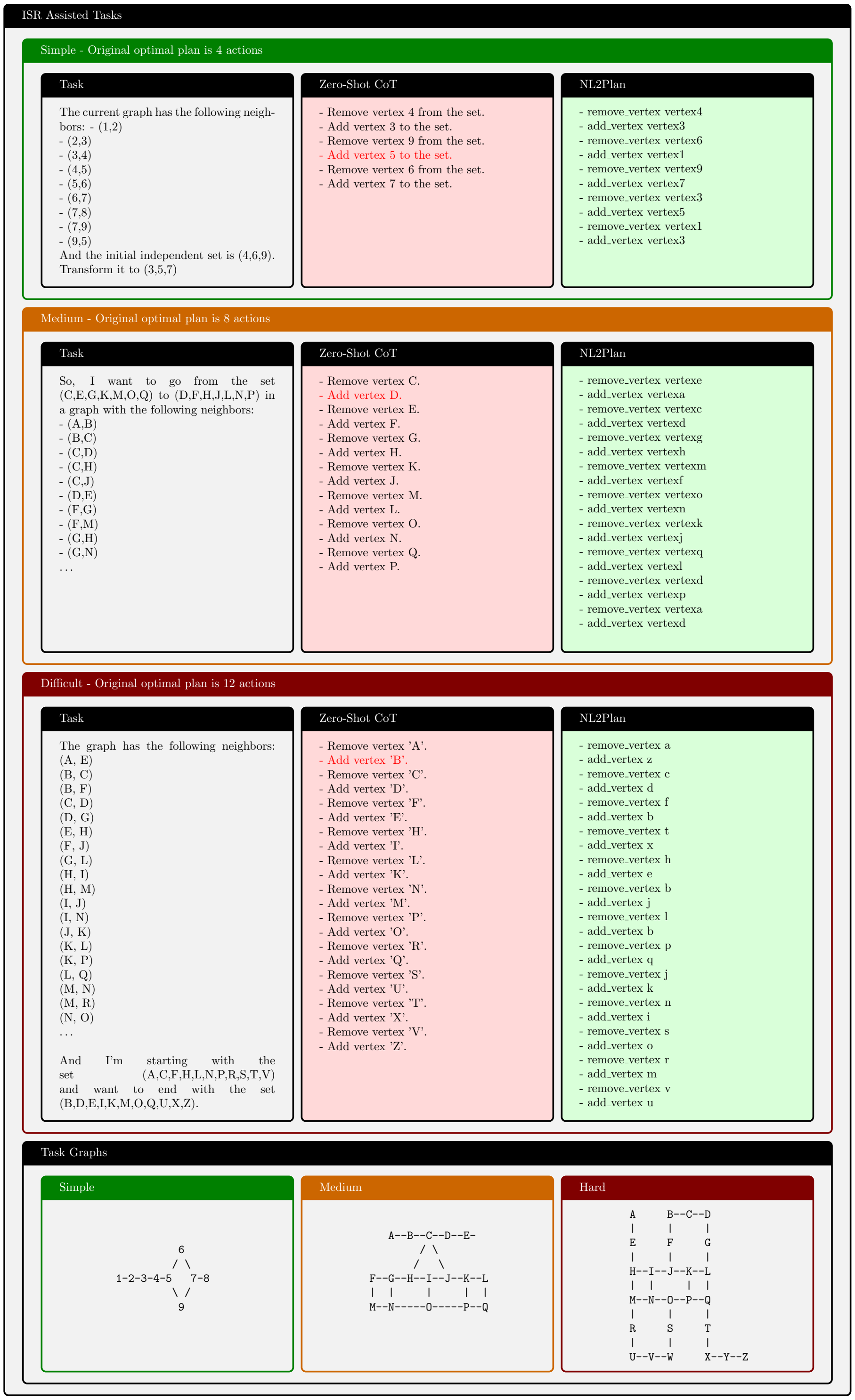
Medium

```
   A--B--C--D--E-
        / \
       /   \
F--G--H--I--J--K--L
|  |     |     |  |
M--N-----O-----P--Q
```

Hard

```
A     B--C--D
|     |     |
E     F     G
|     |     |
H--I--J--K--L
|  |     |  |
M--N--O--P--Q
|     |     |
R     S     T
|     |     |
U--V--W     X--Y--Z
```

Figure A5: The tasks and plans for the ISR domain. Note that CoT action descriptions and NL2Plan names have been shortened. Successful plans are marked in green. Failed plans and their first invalid action are marked in red. Questionable plans that likely differ from user intent or similar are marked in orange. Note that the "add_vertex_to_set" action is implemented as a "reconfigure_set" action, adding the first argument to the set and removing the second.



ISR Assisted Tasks

Simple - Original optimal plan is 4 actions

Task

The current graph has the following neighbors: - (1,2)
- (2,3)
- (3,4)
- (4,5)
- (5,6)
- (6,7)
- (7,8)
- (7,9)
- (9,5)

And the initial independent set is (4,6,9). Transform it to (3,5,7)

Zero-Shot CoT

- Remove vertex 4 from the set.
- Add vertex 3 to the set.
- Remove vertex 9 from the set.
- Add vertex 5 to the set.
- Remove vertex 6 from the set.
- Add vertex 7 to the set.

NL2Plan

- remove_vertex vertex4
- add_vertex vertex3
- remove_vertex vertex6
- add_vertex vertex1
- remove_vertex vertex9
- add_vertex vertex7
- remove_vertex vertex3
- add_vertex vertex5
- remove_vertex vertex1
- add_vertex vertex3

Medium - Original optimal plan is 8 actions

Task

So, I want to go from the set (C,E,G,K,M,O,Q) to (D,F,H,J,L,N,P) in a graph with the following neighbors:
- (A,B)
- (B,C)
- (C,D)
- (C,H)
- (C,J)
- (D,E)
- (F,G)
- (F,M)
- (G,H)
- (G,N)

. . .

Zero-Shot CoT

- Remove vertex C.
- Add vertex D.
- Remove vertex E.
- Add vertex F.
- Remove vertex G.
- Add vertex H.
- Remove vertex K.
- Add vertex J.
- Remove vertex M.
- Add vertex L.
- Remove vertex O.
- Add vertex N.
- Remove vertex Q.
- Add vertex P.

NL2Plan

- remove_vertex vertexe
- add_vertex vertexa
- remove_vertex vertexc
- add_vertex vertexd
- remove_vertex vertexg
- add_vertex vertexh
- remove_vertex vertexm
- add_vertex vertexf
- remove_vertex vertexo
- add_vertex vertexn
- remove_vertex vertexk
- add_vertex vertexj
- remove_vertex vertexq
- add_vertex vertexl
- remove_vertex vertexd
- add_vertex vertexp
- remove_vertex vertexa
- add_vertex vertexd

Difficult - Original optimal plan is 12 actions

Task

The graph has the following neighbors:
(A, E)
(B, C)
(B, F)
(C, D)
(D, G)
(E, H)
(F, J)
(G, L)
(H, I)
(H, M)
(I, J)
(I, N)
(J, K)
(K, L)
(K, P)
(L, Q)
(M, N)
(M, R)
(N, O)
. . .

And I'm starting with the set (A,C,F,H,L,N,P,R,S,T,V) and want to end with the set (B,D,E,I,K,M,O,Q,U,X,Z).

Zero-Shot CoT

- Remove vertex 'A'.
- Add vertex 'B'.
- Remove vertex 'C'.
- Add vertex 'D'.
- Remove vertex 'F'.
- Add vertex 'E'.
- Remove vertex 'H'.
- Add vertex 'I'.
- Remove vertex 'L'.
- Add vertex 'K'.
- Remove vertex 'N'.
- Add vertex 'M'.
- Remove vertex 'P'.
- Add vertex 'O'.
- Remove vertex 'R'.
- Add vertex 'Q'.
- Remove vertex 'S'.
- Add vertex 'U'.
- Remove vertex 'T'.
- Add vertex 'X'.
- Remove vertex 'V'.
- Add vertex 'Z'.

NL2Plan

- remove_vertex a
- add_vertex z
- remove_vertex c
- add_vertex d
- remove_vertex f
- add_vertex b
- remove_vertex t
- add_vertex x
- remove_vertex h
- add_vertex e
- remove_vertex b
- add_vertex j
- remove_vertex l
- add_vertex b
- remove_vertex p
- add_vertex q
- remove_vertex j
- add_vertex k
- remove_vertex n
- add_vertex i
- remove_vertex s
- add_vertex o
- remove_vertex r
- add_vertex m
- remove_vertex v
- add_vertex u

Task Graphs

Simple

```
        6
       / \
1-2-3-4-5   7-8
       \ /
        9
```

Medium

```
   A--B--C--D--E-
         / \
        /   \
F--G--H--I--J--K--L
|  |     |     |  |
M--N-----O-----P--Q
```

Hard

```
A     B--C--D
|     |     |
E     F     G
|     |     |
H--I--J--K--L
|  |     |  |
M--N--O--P--Q
|     |     |
R     S     T
|     |     |
U--V--W     X--Y--Z
```

Figure A6: The tasks and plans for the ISR Assisted domain. Note that CoT action descriptions and NL2Plan names have been shortened. Successful plans are marked in green. Failed plans and their first invalid action are marked in red. Questionable plans that likely differ from user intent or similar are marked in orange.

C NL2Plan Step Prompt Schemas and Examples

1: Type Extraction

Role
Your task is to identify the different types of objects which exist and are relevant in a domain. [Further task details.]

Example
[Chain-of-Thought Example]

Task
Domain
The AI agent here is a logistics planner that has to plan to transport packages within the locations in a city through a truck and between cities through an airplane. In a city, all the locations are connected. Similarly, cities are directly connected allowing airplanes to travel between them. Also, there is no limit to how many packages a truck or plane can carry.

Currently, I've got four packages to ship. . .

Types
First, we need types related to locations and transportation.
‘‘‘
- city: Each city contains an airport and other locations.
- location: Places within cities trucks can visit.
- airport: A location where planes land and take off.

‘‘‘

Next, we consider the transportation vehicles.
‘‘‘
- plane: A type of vehicle used for transporting packages between cities.
- truck: A type of vehicle used for transporting packages within a city.

‘‘‘

Lastly, we need the to define the items being transported.
‘‘‘
- package: An item that needs to be transported from one location to another.

‘‘‘

Figure A7: An example of an input and a solution from the Type Extraction step. Note that parts have been omitted for brevity.

1: Type Extraction

Role
Your task is to identify the different types of objects which exist and are relevant in a domain. [Further task details.]

Example
[Chain-of-Thought Example]

Task
Domain
The AI agent here is a logistics planner that has to plan to transport packages within the locations in a city through a truck and between cities through an airplane. In a city, all the locations are connected. Similarly, cities are directly connected allowing airplanes to travel between them. Also, there is no limit to how many packages a truck or plane can carry.

Currently, I've got four packages to ship…

Types
First, we need types related to locations and transportation.
```
- city: Each city contains an airport and other locations.
- location: Places within cities trucks can visit.
- airport: A location where planes land and take off.
```

Next, we consider the transportation vehicles.
```
- plane: A type of vehicle used for transporting packages between cities.
- truck: A type of vehicle used for transporting packages within a city.
```

Lastly, we need the to define the items being transported.
```
- package: An item that needs to be transported from one location to another.
```

Figure A8: An example of an input and a solution from the Type Hierarchy step. Note that parts have been omitted for brevity.

3: Action Extraction

Role
Your task is to identify what actions an AI Agent would have available in a domain. [Further task details.]

Example
[Chain-of-Thought Example]

Task
Domain
The AI agent here is a logistics planner that has to plan to transport packages...

Currently, I've got four packages to ship...

##Types:
- object: Everything is an object
  - city: Each city contains...
  - location: Places within cities...
    - airport: A location where planes...
  - vehicle: Vehicles transport packages.
    - truck: A type of vehicle....
    - plane: A type of vehicle...
  - package: An item that needs to...

##Actions
###Package related actions
Packages need to be loaded onto vehicles and unloaded at the destination.
‘‘‘
load_package

A package is loaded onto a vehicle at a location. Requires that the package and the truck to be at the same location. Example: package_1 is loaded onto truck_1 at location_1.
‘‘‘

‘‘‘
unload_package

A package is unloaded from a vehicle at a location. Requires the package to be on the vehicle and the vehicle to be at the destination location. Example: package_2 is unloaded from plane_1 at airport_1.
‘‘‘

[Further headers and actions.]

Figure A9: An example of an input and a solution from the Type Extraction step. Note that parts have been omitted for brevity.

4: Action Construction

```
# Role
You are defining the preconditions and effects (represented in PDDL format) of an AI agent's actions. [Further task details.]

# Example
[Chain-of-Thought Example]

# Task
## Domain
The AI agent here is a logistics planner that has to plan to transport packages...

Currently, I've got four packages to ship...

##Types:
- object: Everything is an object
  - city: Each city contains...
  - location: Places within cities...
    - airport: A location where planes...
  - vehicle: Vehicles transport packages.
    - truck: A type of vehicle....
    - plane: A type of vehicle...
  - package: An item that needs to...

## Future actions
The following actions will be defined later and together they make up the entire domain:
- load_package: A package is loaded...
- unload_package: A package is unloaded...
[Remaining actions.]

## Action
load_package

A package is loaded onto a vehicle at a location. Requires that the package and the truck to be at the same location. Example: package_1 is loaded onto truck_1 at location_1.

### Available Predicates
No predicate has been defined yet.
```

```
### Action Parameters
First, we need to know which vehicle and package are being loaded:
‘‘‘
- ?v - vehicle: The vehicle being loaded.
- ?p - package: The package being loaded.
‘‘‘

Lastly, we need to know where the loading is taking place:
‘‘‘
- ?l - location: The location where the loading happens.
‘‘‘

### Action Preconditions
To be able to load at ?l it's specified that:
1: The vehicle is at the location.
2: The package is at the location.

For this we need a predicate:
- "at" to define where something is. This should work for both packages and vehicles, so let's define "(at ?o - object ?l - location)"

Let's specify this in PDDL:
‘‘‘
(and ; All these have to hold
  (at ?v ?l) ; The vehicle is at the location
  (at ?p ?l) ; The package is at the location
)
‘‘‘

### Action Effects
So, what happens when the action is performed?
- The package is no longer at the location
- The package is instead inside the vehicle

For this, we need a new predicate
- "loaded" to specify when and where a package is loaded. "(loaded ?p - package ?v - vehicle)"

Specifying this in PDDL we get:
‘‘‘
(and ; All these have to hold
  (not (at ?p ?l)) ; The package is not at the location
  (loaded ?p ?v) ; The package is instead loaded in the vehicle
)
‘‘‘

### New Predicates
We used two new predicates. Those have to be specified and re-iterated:
‘‘‘
- (at ?o - object ?l - location): true if the object ?o (a vehicle or package) is at the location ?l
- (loaded ?p - package ?v - vehicle): true if the package ?p is loaded in the vehicle ?v
‘‘‘
```

Figure A10: An example of an input and a solution from the Action Construction step. Note that parts have been omitted for brevity.

5: Task Extraction

```
# Role
Your task is to estimate the initial state and the goal state for a PDDL problem based on a domain description and the available actions. [Further task details.]

# Example
[Chain-of-Thought Example]

# Task
## Domain
The AI agent here is a logistics planner that has to plan to transport packages...

Currently, I've got four packages to ship. Two are in a London storage and the rest in Paris. Those from London should be sent to Addr1 in Berlin and to Addr2 in Paris. Those from Paris should both be moved to the London storage.

##Types:
- object: Everything is an object
  - city: Each city contains...
  - location: Places within cities...
    - airport: A location where planes...
  - vehicle: Vehicles transport packages.
    - truck: A type of vehicle....
    - plane: A type of vehicle...
  - package: An item that needs to...

## Predicates
- (at ?o - object ?l - location): true if the object ?o (a vehicle or package) is at the location ?l
- (loaded ?p - package ?v - vehicle): true if the package ?p is loaded in the vehicle ?v
[Further predicates.]
```

```
## Object Instances
There are four packages. The first two start in London, and the remaining two start in Paris:
‘‘‘
- L1 - package: The first London package
- L2 - package: The second London package
- P1 - package: The first Paris package
- P2 - package: The second Paris package
‘‘‘

[Further object instances.]

## State
The London packages all start in the London storage:
‘‘‘
(at L1 LStorage): The first London package location
(at L2 LStorage): The second London package location
‘‘‘

[Further initial predicates.]

## Goal
The goal is for L1 to go to Addr1 and for L2 to be delivered to Addr2, as well as for both P1 and P2 to be transported to London storage. Here's how we can define the goal:
‘‘‘
(and ; All these have to hold
  (at L1 Addr1)) ; L1 is delivered
  (at L2 Addr2)) ; L1 is delivered
  (at P1 LStorage)) ; L1 is delivered
  (at P2 LStorage)) ; L1 is delivered
)
‘‘‘
```

Figure A11: An example of an input and a solution from the Task Extraction step. Note that parts have been omitted for brevity.

D Feedback Checklists

Feedback Checklists

1: Type Extraction

1. Are there additional types which are needed to model the domain?
2. Are additional types needed for organising the type hierarchy?
3. Are any of the types actually objects?
4. Are any of the types actually actionc?
5. Are any of the types actually properties?
6. Is the acting agent itself or the resulting plans included?
7. Will any of the included types only ever be used once?
8. Do any of the types fit better as goals, initial states or predicates?

2: Hierarchy Construction

1. Is any child not a subtype of its parent?
2. Is any subtype not a child of its parent type?
3. Are any new types needed for organisation?

3: Action Extraction

1. Are there additional actions needed for this domain?
2. Should any of the actions be split or combined?
3. Should any of the actions be removed?
4. Should any preconditions be changed?
5. Should any effects be changed?
6. Should any action examples be modified?

4: Action Construction

1. Are any necessary precondition checks missing?
2. Are any unnecessary preconditions checked?
3. Are any necessary effects missing?
4. Are any unnecessary effects included?
5. Can the used predicates be improved?
6. Should any predicate be used in a symmetrical manner?

5: Task Extraction

1. Are any necessary objects missing?
2. Are any unnecessary objects included?
3. Are any objects defined with the wrong type?
4. Are any unnecessary or incorrect predicates declared?
5. Are any necessary predicates missing from the initial state?
6. Is anything missing from the goal description?
7. Is anything unnecessary included in the goal description?
8. Should any predicate be used in a symmetrical manner?

Figure A12: The checklists used for the automatic LLM-driven feedback substeps in NL2Plan. Each feedback prompt includes at least one example of failure for each point on the checklist.

E Automatic Validation

Automatically Validated Properties

Action Construction

- **header_inclusion**: Checks that each expected header is included.
- **keyword_usage**: Checks if one of "forall", "exists", or "if" are used in the action effects.
- **param_types**: Checks if all action parameters use defined types.
- **predicate_names**: Checks if the names of new predicates are duplicates of types or existing predicates. If the new predicate shares arguments with the existing predicate, such as reiterating a predicate, this does not raise an error.
- **predicate_format**: Checks that all predicate arguments are correctly specified and typed.
- **predicate_usage**: Checks that the predicates are used correctly. Confirms that the number of parameters is correct, that all used arguments are defined, and that all argument types are valid.

Task Extraction

Object list properties:

- **obj_type**: Checks if any objects use undefined types.
- **obj_type_name**: Checks if any object name is a type.
- **obj_pred_name**: Checks if any object name is a predicate.

Initial state and goal description properties:

- **pred_exists**: Checks if any used predicate is undefined.
- **pred_nargs**: Checks that the number of predicted arguments are correct.
- **pred_objs**: Checks that all predicate argument objects exist.
- **pred_types**: Checs that all predicate argument types are valid.

Figure A13: The properties checked by the automatic validation. Note that "not" can currently be used in the Task Extraction for the goal description, but not for the initial state.

F NL2Plan Generated Domains

Blocksworld PDDL

```
(define (domain blocksworld)
   (:requirements
      :strips :typing :equality :negative-preconditions :disjunctive-preconditions
      :universal-preconditions :conditional-effects
   )

   (:types
      location - object ; A type of object where blocks can be placed.
      block - location ;  the blocks are the main objects that the robot arm interacts with. they can be moved...
      table - location ;  the table is the surface on which the blocks are placed. it is a location...
   )

   (:predicates
      (on ?b1 - block ?l - location)
      (holding ?b - block)
   )

   (:action pick_block
      :parameters (
         ?b - block
         ?l - location
      )
      :precondition
         (and ; All these have to hold
             (not (exists (?b2 - block) (holding ?b2))) ; The robot arm is not holding any block
             (not (exists (?b2 - block) (on ?b2 ?b))) ; The block to be picked up is not under any other block
             (on ?b ?l) ; The block to be picked up is on the location ?l
         )
      :effect
         (and
             (holding ?b) ; The robot arm is now holding the block
             (not (on ?b ?l)) ; The block is no longer on its previous location
         )
   )

   (:action place_block_on_table
      :parameters (
         ?b - block
         ?t - table
      )
      :precondition
         (and ; All these have to hold
             (holding ?b) ; The robot arm is holding the block
         )
      :effect
         (and
             (not (holding ?b)) ; The robot arm is no longer holding the block
             (on ?b ?t) ; The block is now on the table
         )
   )

   (:action place_block_on_block
      :parameters (
         ?b1 - block
         ?b2 - block
      )
      :precondition
         (and ; All these have to hold
             (holding ?b1) ; The robot arm is holding ?b1
             (not (exists (?b3 - block) (on ?b3 ?b2))) ; There is no block on ?b2
             (not (= ?b1 ?b2)) ; The blocks are different
         )
      :effect
         (and
             (not (holding ?b1)) ; The robot arm is no longer holding ?b1
             (on ?b1 ?b2) ; ?b1 is now on ?b2
         )
   )
)
```

Figure A14: The Blocksworld domain description generated by NL2Plan. The requirement list is fixed and covers all requirements NL2Plan has ever required.

ISR PDDL

```
(define (domain isr)
    (:requirements
        :strips :typing :equality :negative-preconditions :disjunctive-preconditions
        :universal-preconditions :conditional-effects
    )

    (:types
        graph_component - object ; A type of object consisting of components of a graph.
        vertex - graph_component ; a point in the graph. can be part of an independent set.
    )

    (:predicates
        (in_set ?v - vertex) ;  true if the vertex ?v is in the independent set
        (neighbors ?v1 - vertex ?v2 - vertex) ;  true if the vertices ?v1 and ?v2 share an edge
    )

    (:action add_vertex_to_set
        :parameters (
            ?v - vertex
            ?v_remove - vertex
        )
        :precondition
            (and
                (not (in_set ?v)) ; The vertex is not in the set
                (in_set ?v_remove) ; The vertex to remove is in the set
                (not (neighbors ?v ?v_remove)) ; The vertices are not neighbors
                (forall (?n - vertex) ; For all vertices
                    (or ; Either
                        (not (neighbors ?v ?n)) ; The vertex is not a neighbor
                        (not (in_set ?n)) ; Or the neighbor is not in the set
                    )
                )
            )
        :effect
            (and
                (in_set ?v) ; ?v is now in the set
                (not (in_set ?v_remove)) ; ?v_remove is no longer in the set
            )
    )

    (:action remove_vertex_from_set
        :parameters (
            ?v - vertex
        )
        :precondition
            (in_set ?v) ; The vertex is in the independent set
        :effect
            (not (in_set ?v)) ; The vertex is no longer in the independent set
    )
)
```

Figure A15: The ISR domain description generated by NL2Plan. The requirement list is fixed and covers all requirements NL2Plan has ever required. Notably, the Action Construction step implemented the “add_vertex_to_set” as a “reconfigure_set” action. Furthermore, this action both explicitly and implicitly (via the “forall” statement) disallows replacing a node with its neighbor. Allowing neighbor-replacement (and optionally removing “remove_ vertex_from_set”) would result in an entirely correct PDDL description of ISR.

G Zero-Shot Chain-of-Thought Prompt Schemas and Examples

1: Type Extraction

Your job is to provide me with a plan that would solve the following task. Specify each action you wish to take clearly.

The task:
The AI agent here is a mechanical robot arm that can pick and place the blocks. Only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved.

There are four blocks currently. The blue block is on the red which is on the yellow. The yellow and the green are on the table. I want the red on top of the green.

Let's think step by step.

Step 1: The AI robot arm should pick up the blue block which is on top of the red block.

Step 2: The AI robot arm should place the blue block on the table.

Step 3: The AI robot arm should then pick up the red block which is on top of the yellow block.

Step 4: The AI robot arm should place the red block on top of the green block which is on the table.

Now, the red block is on top of the green block as required.

Figure A16: The Zero-Shot Chain-of-Thought prompt schema and an example for the easy Blocksworld task. The section in blue is replaced by the current domain and task descriptions from Appendix A and Appendix B, respectively.