Improved Generalized Planning with LLMs through Strategy Refinement and Reflection

Katharina Stein[1], Nils Hodel[1], Daniel Fišer[3], Jörg Hoffmann[1,2], Michael Katz[4], Alexander Koller[1]

[1]Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
[2]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[3]Aalborg University, Denmark; [4]IBM Research, US
Correspondence: kstein@lst.uni-saarland.de

Abstract

LLMs have recently been used to generate Python programs representing generalized plans in PDDL planning, i.e., plans that generalize across the tasks of a given PDDL domain. Previous work proposed a framework consisting of three steps: the LLM first generates a summary and then a strategy for the domain, both in natural language, and then implements that strategy as a Python program, that gets debugged on example planning tasks. In that work, only one strategy is generated and passed directly to the program generation. If the strategy is incorrect, its implementation will therefore result in an incorrect generalized plan. Here, we introduce an approach that generates the strategy in the form of pseudocode and enables automatic debugging of the pseudocode, hence allowing us to identify and fix errors prior to the generation of the generalized plan itself. Additionally, we extend the Python debugging phase with a reflection step prompting the LLM to pinpoint the reason for the observed plan failure. Finally, we take inspiration from LLM code generation to produce several program variants and pick the best one. Running experiments on 17 benchmark domains with two reasoning and two non-reasoning LLMs, we show that these extensions substantially improve the quality of the generalized plans. Our best performing configuration achieves an average coverage of 82% across the domains.

Code —
https://github.com/coli-saar/genplan-strategy-refine

Datasets — https://github.com/coli-saar/genplan-strategy-refine/tree/main/data

Introduction

Large Language Models (LLMs) have revolutionized a large variety of tasks not only from the field of natural language processing but also from other areas of AI research. One very active area of research deals with LLMs in the context of reasoning problems, and there has been growing interest in using LLMs for symbolic planning in the PDDL language (McDermott 2000; Haslum et al. 2019).

First approaches use LLMs to generate a plan based on the PDDL or natural language (NL) definition of a task. Non-reasoning LLMs tend to not perform well in this set-up (e.g. Stein et al. 2025; Kambhampati et al. 2024; Silver et al. 2022). Improvements have been achieved by incorporating thoughts and automatic corrections based on feedback into the process (e.g. Stein et al. 2025; Stechly, Valmeekam, and Kambhampati 2025), and reasoning LLMs achieve much better results. Yet scalability to larger tasks still tends to be inferior to the symbolic state of the art (e.g. Corrêa, Pereira, and Seipp 2025; Valmeekam et al. 2025), especially on unseen domains. In addition, even where they scale, these approaches can become costly in the number of LLM calls and processed tokens, being called on every planning instance (sometimes on every state in a plan), and with the number of tokens generated growing linearly in plan length.

Silver et al. (2024) proposed an approach that has the potential to overcome these issues. Instead of using LLMs to generate plans for individual tasks, they prompt the LLM to produce a *generalized plan*, that generalizes across the tasks of a given PDDL domain (e.g. Srivastava, Immerman, and Zilberstein 2011). A generalized plan contains branches (if-then-else behavior) and loops to deal with different cases and scaling task size. Silver et al. (2024) show how to use LLMs to generate Python programs representing such plans. This solves the issue regarding cost for LLM calls, as that cost is now per-domain instead of per-task. It also potentially addresses the scalability issue: if the generalized plan is correct, planning tasks of arbitrary size can be solved.

Silver et al. let an LLM generate strategy a in NL for the given PDDL domain. They then prompt the LLM to generate Python code for that strategy, that is then debugged (see Figure 1, top part). Silver et al.'s approach achieves good performance on tasks of varying size in 5 out of 7 tested domains when using GPT-4. However, when extending their evaluation to a larger set of domains, we find that their approach struggles with generating correct generalized plans.

A key bottleneck of their approach is the strategy generation step: they use a simple prompting approach to let the LLM create a generalizing NL strategy, which is directly passed to the code generation. If the strategy is incorrect, the LLM is hence prompted to generate a Python function that implements an inadequate logic.

Here we address this limitation by treating the strategy generation not only as a Chain-of-Thought (CoT) step (Wei et al. 2022) but as a central part of the generalized planning framework that is responsible for an important sub-task. Figure 1 (bottom) provides an overview of our pipeline. Our main contribution is an approach that allows us to automatically validate and refine the strategy before passing it to the

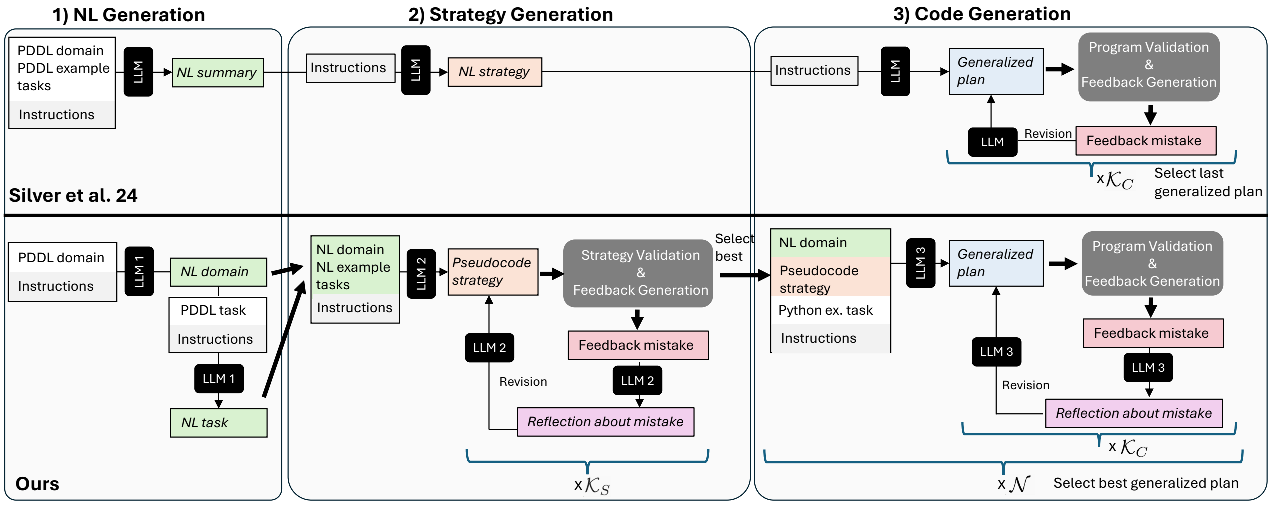


Figure 1: Overview of the framework of Silver et al. (2024) (top) and our framework (bottom). The main three parts for both are the NL generation, the strategy generation and lastly the code generation, i.e. the generation of the generalized plan.

code generation. Furthermore, our approach generates the strategy in the form of *pseudocode*, that is already closer to the final target structure. For the refinement, we let an LLM generate PDDL plans for a set of debugging tasks based on the pseudocode, and we check correctness of these plans. We then pass the feedback about errors into a reflection step (inspired by e.g. Shinn et al., 2023; Madaan et al., 2023). In that step, the LLM is prompted to identify the responsible location in the pseudocode, and the reason for the mistake. The LLM is then prompted to update the pseudocode accordingly. We select the best pseudocode based on the debugging tasks as the strategy to be implemented.

We also introduce some improvements over Silver et al.'s approach in the code generation step. First, we also add a reflection step to the automated debugging of the Python programs. Second, we take inspiration from LLM-based code generation to produce several initial versions of the program (e.g. Tang et al. 2024; Wang et al. 2024). We pick the best program based on performance on the debugging tasks.

We empirically evaluate our method on 17 PDDL domains, including the ones Silver et al. ran their experiments on, using GPT-4o, Llama3.3, DeepSeek-V3.2 and Qwen3-Thinking as the LLMs. Compared to Silver et al., our approach improves average performance across domains substantially for all four LLMs. Our approach in combination with DeepSeek solves on average 82% of the evaluation tasks. In 14 domains, our approach achieves perfect coverage for at least one of three runs. We manually verified that our 100% coverage programs generalize beyond the evaluation data and to all tasks that can be generated using the respective instance generator. In experiments on a range of "costumed" benchmarks that do not appear in the LLM training data, our approach also exhibits good performance, indicating its generalization capabilities.

```
(:predicates (object ?obj) (location ?loc) (at ?obj ?loc) ...)

(:action load-truck
   :parameters (?obj  ?truck  ?loc)
   :precondition (and (object ?obj) (truck ?truck) (location ?loc)
                      (at ?truck ?loc) (at ?obj ?loc))
   :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))
```

```
(:objects c0  t0  l0-0  l1-0  p0  a0)
(:init (truck t0)  (location l0-0) (location l1-0) (object p0)
       (airplane a0) (airport l0-0) (at t0 l0-0) (at p0 l1-0) ...)
(:goal (and (at p0 l0-0)) )
```

Figure 2: Excerpt from the Logistics PDDL domain (top) and a Logistics PDDL problem (bottom).

Background

Classical planning. In classical planning the task is to find a sequence of actions (a plan) that leads from a given initial state into a state that satisfies a goal condition. A commonly used formalism to define classical planning tasks is the Planning Domain Definition Language (PDDL) (McDermott 2000; Haslum et al. 2019). In PDDL, a planning task is specified by a domain along with a problem. The domain defines the world model, including the predicates for describing the possible world states and all actions that can be used to change the state. Each action has preconditions specifying what needs to be true in order to apply the action, and effects that specify how applying the action changes the world state. A specific problem file defines a set of available objects, the initial world state and the goal. The solution is a plan consisting of actions from the domain.

Figure 2 (top) shows an excerpt from the Logistics domain that models transporting packages with trucks within cities and with planes between cities. The action "load-truck" can only be executed if the parameter "?truck" is a truck, "?obj" an object and "?loc" a location, and if "?truck" and "?obj" are both at "?loc" (precondition). Applying the action changes the location of the package from ?loc to the

?truck (effect). Figure 2 (bottom) shows part of a task where the goal is to move package “p0” from “l1-0” to “l0-0”.

While there are no formal constraints on the possible initial states and goals, the instance generators used to construct benchmarks usually only generate a subset of all possible tasks. For example, Logistics benchmarks only include tasks where the goal specifies locations of packages but never e.g. a location of a vehicle.

Generalized planning. Generalized planning (e.g. Bonet, Palacios, and Geffner 2009; Srivastava, Immerman, and Zilberstein 2011; Jiménez, Segovia-Aguas, and Jonsson 2019) seeks plans that generalize over a set of planning tasks. Different variants of this problem have been discussed in the literature. Here, we follow up on Silver et al.’s (2024) work, which generates Python programs intended to generalize over all tasks in a given PDDL domain. The right part of Figure 3 shows an excerpt of such a program that outputs a plan for a specific input task.

The top part of Figure 1 illustrates Silver et al.’s pipeline. It consists of three steps, of which the first two serve as CoT steps. First, the LLM is prompted to generate a short summary of the domain (green color in the figure) based on the PDDL domain file and exemplary PDDL tasks. Afterwards, the LLM receives a prompt stating that there exists “a simple strategy for solving all tasks in this domain without using search”, and is prompted to tell the strategy (peach color).

Then, the LLM is asked to implement that strategy as a Python program, i.e. the generalized plan (blue). For this step, it receives the function signature and a short description of the inputs and output. Silver et al. then use an automatic debugging approach to iteratively revise the generalized plan based on the outcome of running the program on a set of training tasks. If the program interrupts with an error, reaches a timeout or does not return a correct plan - as determined by the plan validator VAL (Howey, Long, and Fox 2004) - the LLM receives a new prompt with a feedback (coral color) and the instruction to fix the code. The feedback includes details about the error that occurred and the PDDL defition of the task for which it occurred. This process continues until all training tasks are solved or a maximum number of revisions, $\mathcal{K}_C$, is reached. The last Python program obtained in this manner is selected as the output.

Generating and Refining Pseudocode Strategies

Generating generalized plans for planning domains using LLMs is a complex task that poses two main challenges. Because the LLM only has access to the domain and example tasks, it first needs to abstract away from individual tasks to the higher-level logic that generalizes across the domain, i.e. a strategy. Second, the LLM is required to implement that strategy in an executable form, a Python program in our case. The correctness of the final program therefore heavily depends on the quality of the generated strategy, as this serves as a kind of program specification. We therefore treat the strategy generation as a separate subtask in our framework with the dedicated purpose of generating a strategy that is correct and closely matches the specification of the target program, hence reducing the complexity of the code generation itself.

Generating Pseudocode Strategies

Our goal is to improve the quality of the strategies that the LLM is asked to implement in order to shift most of the work beyond the mere conversion into Python to the previous step of the generation framework. We therefore instruct the LLM to generate the strategy in the form of pseudocode that should be detailed and specific enough to be converted into an executable program in a straightforward way. The prompt for this step consists of the NL descriptions of the domain and two example tasks and instructions to think step-by-step (zero-shot CoT, Kojima et al., 2022) for developing a strategy that can be turned into a program.

The left part of Figure 3 shows part of the output generated for the Logistics domain consisting of the thoughts (top, yellow) and the pseudocode (bottom, peach color) that gets extracted for the subsequent steps. We show inputs to the LLM in regular font and LLM outputs in *italics* in all Figures. The right part of Figure 3 shows an excerpt of a generalized plan for Logistics. While the pseudocode strategy is expressed in natural language, it includes key words such as “for each”, “if”, “continue”. Furthermore, the steps are enumerated in a structured and nested way that closely matches the overall structure of the final Python program as indicated by the arrows.

Pseudocode strategies hence express the strategy in a more detailed form, specifically structured in a way that is useful for its actual target use case. If an LLM is simply asked to generate a strategy and produces a simple, natural language summary of it, more work needs to be done (implicitly) to map this strategy into a program.

Debugging at the Strategy Level

If the strategy generated by the LLM is wrong, then an implementation of it will also result in a wrong generalized plan. We address the challenge of improving the correctness of the strategy by introducing an approach for automatically validating and refining the pseudocode.

Validating the pseudocode strategies without a human in the loop is hard as the pseudocode is not executable, i.e. we cannot run it on example tasks and assess the correctness of the outcome. Letting an LLM judge its own output for reasoning problems can even lead to worse performance (e.g. Stechly, Valmeekam, and Kambhampati 2025). Therefore, we introduce an approach that indirectly validates the correctness of the pseudocode using an LLM and a symbolic plan validator as illustrated in Figure 4 (left). We use a small set of tasks from the target domain as *debugging tasks*. In particular, we provide the pseudocode strategy to an LLM and prompt it to generate the PDDL plan for a given debugging task (in NL) by following the strategy. The generated plan is then validated using VAL. If the plan is incorrect, the validation output is converted into a feedback message. For the conversion, we incorporate the feedback generator Stein et al. (2025) used for their experiments on PDDL inputs.

Instead of directly prompting the LLM to update the pseudocode based on the feedback, we add a reflection step, in-

```
### Step-by-Step Strategy Development:
1. **Identify the Initial and Goal States:**
  - Determine where each package is initially located and where it ...
2. **Plan the Movement of Packages:**
  - For each package, determine if it needs to be moved. If a package
    is already at its goal location, no action is needed for that package.
  - If a package needs to be moved to a different city, plan to use an
    airplane. Otherwise, use a truck.  ...
```

```
```
1. Initialize the current location of all vehicles (trucks and airplane).
2. For each package p in the list of packages:
  a. If p is already at its goal location, continue to the next package.
  b. Determine the current location of p and its goal location.
  c. If p's current city is different from its goal city:
    i. Identify the nearest truck to p's current location.
    ii. If the truck is not at p's current location, drive the truck to
        p's current location.
    iii. Load p onto the truck.
    iv. Drive the truck to the nearest airport in the current city.
    v. Unload p from the truck at the airport.
    vi. Check the current location of the airplane. If the airplane is
        not at the airport, fly it to the airport.
    vii. ...
  d. If p's current city is the same as its goal city:
    i. Identify the nearest truck to p's current location.
    ii. ...
```

```python
def generate_solution(objects: Set[str], init: Set[Tuple], goal: Set[Tuple]) -> List[str]:
  ...

  def move_truck(truck, loc_from, loc_to, city, state):
     if loc_from != loc_to:
        solution.append(f'(drive-truck {truck} {loc_from} {loc_to} {city})')
  ...

  solution = []

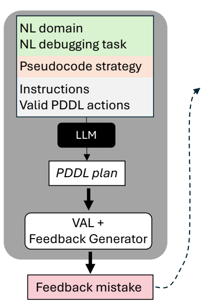
  current_state = {}  # Initialize the current state from the initial facts
  for fact in init:
    ...

  for fact in goal:
     if fact[0] == 'at':
       obj, goal_loc = fact[1], fact[2]
       current_loc = find_location(obj, current_state)
       if current_loc == goal_loc:
          continue

       current_city = find_city(current_loc, current_state)
       goal_city = find_city(goal_loc, current_state)

       if current_city != goal_city:
          truck, truck_loc = find_nearest_truck(current_loc, current_city, current_state)
          move_truck(truck, truck_loc, current_loc, current_city, current_state)
          solution.append(f'(load-truck {obj} {truck} {current_loc})' )
          ...
       else:
          truck, truck_loc = find_nearest_truck(current_loc, current_city, current_state)
          ...
  return solution
```

Figure 3: Left: Excerpt of the LLM output for generating a pseudocode strategy for the Logistics domain, consisting of a CoT (top, yellow color) and the pseudocode (bottom, peach color). Underlined steps were initially missing and added during debugging. Right: excerpt of a generalized plan implementing the pseudocode strategy. Arrows illustrate corresponding parts.




```
The following plan was created based on the strategy:
(load-truck p0 t0 l0-2)
(drive-truck t0 l0-2 l0-0 c0)
(fly-airplane a0 l1-0 l0-0)
(load-airplane p0 a0 l0-0)
...
The action (load-airplane p0 a0 l0-0) at step 3 is not executable
because the preconditions of the action are not satisfied:
At that specific step
- it is not the case that (at p0 l0-0)
---
Please help me to understand why this mistake happened.
Think about
1) which specific part of the strategy lead to the mistake and
2) why this part of the strategy is not adequate
Do not provide any revisions yet but enough details to understand
why the strategy does not work for the task without revisions.
```

Figure 4: Left: approach for generating a PDDL plan for a debugging task based on the pseudocode and obtaining a feedback message (coral color). Right: example prompt for the reflection step consisting of a feedback message for a Logistics task and instructions (grey color).

```
1. **Specific Part of the Strategy**:
  - ... occurs at the step where the package is supposed to be loaded onto the airplane.
2. **Why This Part of the Strategy is Not Adequate**:
  - The strategy assumes that once the truck reaches the airport, the package can be
    directly  loaded onto the airplane. However, the strategy does not explicitly include ...
  - The precondition for loading an object onto an airplane is that the object must be at
    the same location as the airplane. In the plan, the package p0 is still inside the truck t0
    when the action to load it onto the airplane is attempted. This violoates the ...
The strategy needs to be revised to include an explicit step to unload the package from
the truck at the airport before attempting to load it onto the airplane...
```

Figure 5: Excerpt of the reflection generated for the example in Figure 4

spired by approaches that let LLMs reflect about ways to improve over previous outputs (e.g. Madaan et al. 2023; Shinn et al. 2023). We combine the feedback about the mistake and the generated plan and with instructions to reflect about the part of the pseudocode that caused the mistake and the reason why that part is incorrect. After generating the reflection response based on that prompt, the LLM is then asked to correct the pseudocode by thinking step-by-step. This process is continued until the LLM generates correct plans for all debugging tasks or a maximum number of debugging iterations, $\mathcal{K}_S$, is reached. Then the pseudocode that resulted in the highest number of solved tasks is selected as the pseudocode for the code generation step.

One bottleneck is that there is no guarantee that the LLM will generate a correct plan given a correct strategy or that a mistake in the plan is actually caused by a mistake in the strategy. However, our approach guarantees that the feedback the LLM achieves about the mistake is always correct with respect to the plan. Additionally, if the pseudocode is missing important details or steps, and the LLM generates a plan reflecting this issue, then our approach makes it possible to automatically find and potentially correct these issues.

Figure 4 (right) shows an example of a reflection prompt, including the feedback message for a plan (coral color) that was generated based on the first version of the pseudocode in Figure 3, where the underlined steps were missing before debugging. In particular, the step of unloading the package from the truck before loading it onto the airplane (v.) was missing and the LLM generated a plan that was missing that step as well. In order to load a package onto an airplane in Logistics it needs to be at the same location and not in another vehicle. The excerpt of the generated LLM reflection
```

in Figure 5 illustrates that the LLM correctly identified the mistake and the required extension of the pseudocode.

NL descriptions. For the strategy validation approach, we provide the domain and debugging task in NL form. Therefore, we require a separate NL description for each debugging task. We obtain the NL descriptions in a two-step process (see NL Generation, Figure 1): First, the LLM is prompted to generate the NL domain description given the PDDL domain. Afterwards, the NL description of each debugging task is generated based on its PDDL definition and the PDDL and NL domain descriptions. We also use that NL domain description and two debugging task descriptions as input for the pseudocode generation.

Adding Reflection to Code Debugging

While LLMs perform well on generating short, single-fuction code, generating larger code with several, dependent functions is complex (Tang et al. 2024; Du et al. 2024). Therefore, the automatic refinement based on feedback is important. However, debugging itself can also be complex, especially when the mistake that occurs needs to be traced back to the actual, logical error in the code. We therefore use a similar approach as for the strategy debugging, where the LLM is first asked to reflect on the location and reason of the error before revising the program (see Figure 1, step 3).

We run the generated program on all debugging tasks and create not only negative feedback but also positive feedback as additional information for the debugging. We include all solved debugging tasks in their Python format together with the correct outputs in the feedback prompt. We then add one task for which the program returned an incorrect output together with the feedback message. Figure 6 shows an example of the positive and negative feedback (coral color) and the reflection instructions (grey color). If the code returns an incorrect output, we again use the feedback generator from Stein et al. (2025) to convert the output of VAL. Additionally, we also enumerate the steps in the output plan in the feedback message, to make it explicit to which action a feedback of the form "the action ACTION in step X is not executable" actually refers. We provide more details about the feedback messages in the supplementary material.

```
The code you provided me solved the following tasks correctly:
objects = {...} \n init = {...}  \n. goal = {...}
The code returned the correct output: ...
------------------------------------------------------------
The code failed on the following task:
objects = {'a0', 'c0', 'c1', 'l0-0', 'l0-1', 'l1-0', 'l1-1', 'p0', 'p1', 'p2', 't0', 't1'}
init = {('airplane', 'a0'), ('airport', 'l0-0 ), ('at', 'a0', 'l0-0'), ('at', 'p0', 'l1-1 ), ...}
goal = {('at', 'p0', 'l0-1'), ('at', 'p1', 'l1-1'), ('at', 'p2', 'l1-0')}

The code raised the following exception:
Traceback (most recent call last):
  File "<file-name-omitted>", line 56, in generate_solution
    current_state.remove(('at', truck, current_loc))
KeyError: ('at', 't0', 'l0-0')
------------------------------------------------------------
Please help me to understand why this mistake happened.
Think about
1) which specific part of the code lead to the mistake and
2) why this part of the code is not adequate for implementing a correct strategy
Do not provide any revisions yet but enough details to understand why the code does
not work for the task without revisions and which specific parts need to be adapted.
```

Figure 6: Prompt for the reflection about mistakes in the generated Python program, consisting of the reflection instructions (grey color) and an example feedback message obtained for a Logistics debugging task (coral color).

Producing Multiple Code Versions

One common approach used for LLM-based code generation is to generate not only a single program but several output programs by using a higher temperature or nucleus sampling (Holtzman et al. 2020), i.e. increasing the cumulative probability threshold based on which the set of tokens to sample from is determined (e.g. Tang et al. 2024). We also propose to generate multiple program versions based on the same strategy, but operationalize this in a different way and keep greedy decoding and the temperature of 0. Instead, we randomly change the order in which the objects and the facts of the goal state of the Python example task are presented in the prompt. Apart from this small change, the input prompts are the same for generating all initial programs.

The different initial programs are generated and debugged one after the other. Specifically, the LLM generates the first program, and the debugged versions of it, as described in the previous section. If none of the programs solves all the debugging tasks, the code generation part is restarted with the newly sampled ordering. The code generation stops if a program solves all debugging tasks or a defined limit $\mathcal{N}$ of initial programs is reached. Finally, the best program is selected from all generated ones based on the debugging data.

Experiments

Benchmarks. We consider domains expressed using a STRIPS subset of PDDL that allows variable typing and is restricted to conjunctive conditions with negation. We conduct experiments on the seven domains on which Silver et al. (2024) evaluated their approach, and on 10 of the domains on which Stein et al. (2025) ran LLM action-choice experiments. We remove all action costs from the domains. For each domain, we compose a dataset of tasks taken from previous work and tasks generated by us using available instance generators. We randomly select 6 debugging tasks per domain that are small compared to the tasks in the evaluation data. In particular, we only consider tasks for which we can obtain optimal plans, and the number of objects and optimal plan length of each debugging task is among the 16 smallest values of object number and plan length in the overall dataset (see supplementary material for more details).

Costumed and anonymized benchmark variants. Inspired by ideas outside planning (Duchnowski, Pavlick, and Koller 2025), we also run experiments on "costumed" variants of all domains from Silver et al. (2024) that are structurally equivalent but phrased differently and therefore have not been part of the training data of the LLMs. We define new names for the actions, predicates and objects in the original PDDL, in a way that is still semantically reasonable. For example, in the costumed ferry domain, the ferry is a squirrel that needs to jump between trees in order to move nuts.

Additionally, we follow previous work and create anonymized benchmark variants without real-world related semantics (e.g. Silver et al. 2024). In particular, we replace all names with generic names of the form "action", "predicate", "object" and "type" and number these (e.g. "action_1"). For these variants we slightly adapt the prompts for the NL generation step to emphasize that all names from the PDDL need to be included in the output in their exact form.

Set-up. We run our experiments using two non-reasoning models, GPT-4o and Llama3.3-70B Instruct, and two reasoning models, DeepSeek-V3.2 and Qwen3-30B-A3B Thinking (see supplementary material for details). We use the same prompts for all all models but remove instructions to think for the reasoning models.

In all experiments, we select the generated program for the final evaluation based on the best performance on the debugging data. In case of ties, we select the one generated at a later step. We also apply the same approach for selecting the pseudocode that is passed to the code generation. If a program does not terminate within 45 seconds, it is interrupted and a timeout feedback is generated.

For each domain and version of the pipeline, we conduct three runs. We split the debugging tasks into three pairs and use a different pair as the examples for the generation of the strategy for each run. All six tasks are used for debugging.

When generating the initial programs, we provide the LLM with one debugging task in Python format and a corresponding plan as an example. If the LLM generated a correct plan for any debugging task during the pseudocode validation, we select that task and plan as the example. Otherwise, we show a plan generated by an optimal symbolic planner.

Evaluation. For running the Python programs on the evaluation tasks, we impose the same time limit of 45s as in debugging. Our main evaluation metric is coverage, the percentage of evaluation tasks for which the Python program generates a correct plan. We report both the average over all runs and the coverage of the run with the highest coverage on the evaluation data. As the Python program output can depend on the ordering of objects and initial/goal facts in the input, we run 4 random orderings and treat the output as correct only if all runs succeed.

Our framework. We test our generalized planning framework for two different combinations of the maximum number of initial programs ($\mathcal{N}$) and code debugging steps ($\mathcal{K}_C$). For one experiment we set $\mathcal{N}$ = 3 and $\mathcal{K}_C$ = 6, resulting in a maximum of 21 generated programs. For the other experiment, we set $\mathcal{N}$ = 5 and $\mathcal{K}_C$ = 3, hence increasing the number of initial programs while keeping the maximum number of generated programs similar (20). We refer to the two versions as `F3-6` and `F5-3`. For both versions we set $\mathcal{K}_S$ = 5.

Ablations. We conduct three ablation experiments to assess the effect of our pipeline extensions. The base approach for all ablation experiments is `F3-6`. We assess the effect of generating multiple initial programs by setting $\mathcal{N}$ = 1 (`-MC`). In order to test to what extent debugging at the strategy level is beneficial we set $\mathcal{K}_S$ to 0 (`-SD`). Lastly, we prompt the LLM to revise the code directly based on the feedback, to assess the effect of the reflection step (`-CR`).

Baselines. We compare the performance of our approach to the framework by Silver et al. (2024) (`Sil`) and to a reimplementation of their pipeline (`Bas`). For `Bas` we make a number of smaller changes to the original pipeline for a fairer comparison. First, we adapt the phrasing of the prompts to be more similar to our prompts. We also separate the three parts of the pipeline and use the output of the previous step as part of the input for the next step, as done in our main framework. To account for the fact that no PDDL is available at code generation time, we provide the definition of the example task and of the failed task in Python format. The final program is selected based on the debugging data.

Symbolic planners. Our LLM-generated programs come without any guarantees, and are quite different in nature to symbolic planners providing guarantees through search, so a direct comparison is not possible. To nevertheless provide a bit of a measuring line, we run $A^\star$ with the LM-cut heuristic (`lm`) (Helmert and Domshlak 2009) and GBFS with the FF heuristic (`ff`) (Hoffmann and Nebel 2001), as baselines for optimal and satisficing symbolic planning respectively. We ran these planners on Intel Xeon E5-2687W processors with limits of 30m and 8GB. We also report coverage for the same 45s limit applied to the execution of generalized plans.

Results

Improvements over baselines. Table 1 and Table 2 show the percentage of solved tasks per domain for the best run as well as averaged over all three runs for the non-reasoning models and the reasoning models respectively. Comparing the average of the best baseline (`Sil`, `Bas`) and our best approach (`F3-6`, `F5-3`), our approach improves over the baseline by 23 percentage points when using GPT-4o, 20 points when using Qwen Thinking, 14 using DeepSeek and 12 using Llama. Overall, the reasoning models perform better than the non-reasoning models but even for them our approach outperforms the baselines. In particular, the configuration with the highest across-domain average is our `F5-3` approach with DeepSeek.

Comparing the per-domain averages of `F3-6` and `F5-3`, we observe that none is consistently better than the other. As the benefit of continuing to debug vs. generating a program from scratch depends the type of mistake and the complexity to fix it, it is likely that a good balance between both depends on the specific domain and program.

For the ablations, we find that removing each of the three ablated parts of the approach has a negative effect on some of the domains. Overall, the ablation results illustrate that all three of our contributions are needed to achieve high performance across different domains and LLMs.

We also provide an overview of the distribution of the types of errors encountered in the automatic evaluation for each of the LLMs in the supplementary material.

Generalization power of our 100% policies. We manually analyzed the 100% coverage programs generated by our `F5-3` configuration using GPT-4o (12 domains) and DeepSeek (14 domains) relative to the respective instance generators. In all these domains, the programs generalize beyond the evaluation dataset and indeed solve all tasks that

Domains	Avg coverage three runs														Coverage best run													
	GPT-4o							Llama							GPT-4o							Llama3.3						
	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`
Domains from Silver et al. (2024)																												
delivery	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
ferry	33	**100**	**100**	**100**	35	**100**	**100**	**100**	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
gripper	79	**100**	**100**	88	**100**	**100**	**100**	64	55	**100**	67	**100**	88	88	**100**	**100**	**100**	**100**	**100**	**100**	**100**	64	**100**	**100**	**100**	**100**	**100**	**100**
heavy	**100**	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	88	92	97	53	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
hiking	**100**	0	33	67	0	0	67	**100**	43	95	67	81	**100**	67	**100**	0	**100**	**100**	0	0	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
miconic	11	4	**68**	33	0	1	4	**41**	4	5	4	10	0	0	32	12	**100**	**100**	0	3	12	**100**	12	12	9	18	0	0
spanner	0	6	33	**67**	33	**67**	33	0	0	33	**67**	0	1	12	0	15	**100**	**100**	**100**	**100**	**100**	0	0	**100**	**100**	0	3	35
Additional Domains																												
beluga	0	0	0	0	0	0	0	**7**	0	0	0	0	0	0	0	0	0	0	0	0	0	**10**	0	0	0	0	0	0
blocksw.	2	**12**	6	7	6	5	4	**50**	0	14	11	5	8	11	4	**20**	12	13	8	6	6	**100**	1	20	22	14	12	14
goldminer	6	0	4	**11**	2	3	2	**3**	0	0	1	0	2	0	14	0	6	**24**	6	6	6	**5**	0	0	4	0	**5**	0
grippers	**100**	33	**100**	**100**	71	**100**	**100**	71	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	93	**100**	**100**	**100**	**100**	**100**	**100**
logistics	2	45	**100**	94	94	77	74	7	12	42	46	**60**	21	16	6	**100**	**100**	**100**	**100**	**100**	**100**	14	19	94	**100**	83	26	41
minigrid	0	31	48	**61**	37	36	42	21	26	**65**	51	41	53	46	0	42	54	**72**	68	42	47	42	37	**82**	60	42	64	54
rovers	0	0	**7**	0	0	1	0	0	0	0	0	0	0	0	0	0	**20**	0	0	4	0	0	0	0	0	0	0	0
satellite	33	48	**69**	29	67	60	45	31	4	36	36	35	32	**37**	60	68	**100**	44	72	**100**	52	36	12	**44**	**44**	**44**	**44**	**44**
transport	0	0	33	**67**	0	0	0	0	0	**60**	26	19	59	33	0	0	**100**	**100**	0	0	0	0	0	89	79	57	**100**	**100**
visitall	70	80	80	**100**	33	78	51	20	15	60	52	55	88	47	**100**	**100**	**100**	**100**	**100**	**100**	**100**	35	20	81	**100**	**100**	**100**	91
Avg	37	37	58	**60**	40	49	48	42	31	**54**	48	45	50	42	48	50	**76**	74	56	57	60	53	41	**66**	**66**	56	56	58

Table 1: Percentage of solved tasks using non-reasoning LLMs for the original framework by Silver et al. (2024) (`Sil`) and the re-implemented baseline (`Bas`) and our generalized planning approach with $\mathcal{N}$ = 3, $\mathcal{K}_C$ = 6 (`F3-6`) and $\mathcal{N}$ = 5, $\mathcal{K}_C$ = 3 (`F5-3`). The three ablations `-MC`, `-SD` and `-CR` are based on `F3-6`. We report the average coverage over three runs and coverage of the best run. For both, we show in **bold** the best generalized planning approach for each model.

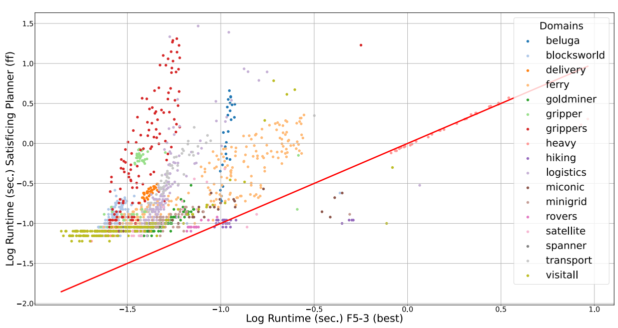


Figure 7: Runtime of the best generalized plan by `F5-3` (DeepSeek) (x-axis) and of `ff` (y-axis) for each commonly solved task. Diagonal is plotted in red.

can be generated with the instance generators. In particular, the programs generalize to tasks of arbitrary size. For example, the programs for Ferry can solve tasks with arbitrary numbers of cars and locations, provided that cars are initially not on the ferry and the ferry location is not part of the goal (which are exactly the restrictions inherent in the instance generators). This shows that although LLMs fail to generalize to larger task sizes and plan lengths when generating plans directly (e.g. Valmeekam et al. 2025), their knowledge from pretraining can be exploited to generate programs that do generalize.

A full documentation of our manual program analysis is in the supplementary material. Briefly summarized, the main control structure of most of the analyzed programs is a loop that loops over all goal facts (or objects part of the goal) and that contains the code for generating the sub-plan required to arrive at a state satisfying that goal fact (e.g. see first for-loop in Figure 3). If the loops themselves correctly implement the sub-strategies and cover all relevant possible state conditions (e.g., whether a truck is already at the package location and if not) then solving tasks with a higher number of objects that require longer plans comes down to simply iterating through the loops more often.

Comparison to symbolic planners. As the rightmost columns in Table 2 show, optimal planning becomes hard for our evaluation tasks within the given time limits, and is often outperformed by the Python programs. But satisficing planning still reigns supreme in coverage, being beaten only in the Spanner domain.

Looking beyond coverage however, the Python programs have substantial advantages. As pointed out above, many of the best programs generalize to the entire domain. Given the polynomial runtime in input task size, the programs are thus bound to eventually outscale any symbolic planner based on search. More generally, program execution is most of the time much faster than plan generation via search.

To give an assessment of this aspect in our benchmarks (in many of which, state-of-the-art symbolic planners perform quite well), Figure 7 compares times for the best Python program (`F5-3`, DeepSeek) vs. the satisficing planner `ff`. Focusing on tasks solved by both, program execution is considerably faster for 97% of the tasks (note the exponential scaling in Figure 7).

Domains	Avg coverage three runs											Coverage best run											Cov. symbolic			
	DeepSeek				Qwen3 Thinking							DeepSeek				Qwen3 Thinking							`lm`		`ff`	
	`Sil`	`Bas`	`F5-3`	`F3-6`	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`	`Sil`	`Bas`	`F5-3`	`F3-6`	`Sil`	`Bas`	`F5-3`	`F3-6`	`-MC`	`-SD`	`-CR`	45s	30m	45s	30m
Domains from Silver et al. (2024)																										
delivery	**100**	**100**	67	**100**	70	**100**	**100**	70	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	0	0	100	100
ferry	**100**	**100**	**100**	**100**	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	31	43	100	100
gripper	67	88	**100**	**100**	43	64	43	64	64	76	60	**100**	**100**	**100**	**100**	64	64	64	64	64	**100**	64	15	40	100	100
heavy	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	100	100	100	100
hiking	33	86	**100**	86	**76**	27	44	33	33	0	33	**100**	**100**	**100**	**100**	**100**	29	**100**	**100**	**100**	0	**100**	100	100	100	100
miconic	41	71	**100**	83	46	21	41	41	89	11	9	**100**	**100**	**100**	**100**	**100**	50	**100**	**100**	**100**	12	12	56	62	100	100
spanner	33	**100**	**100**	**100**	15	67	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	44	**100**	**100**	**100**	**100**	**100**	**100**	15	41	15	59
Additional Domains																										
beluga	0	2	24	**30**	1	0	0	**7**	2	3	0	0	5	**43**	**43**	2	0	0	**10**	7	10	0	0	0	100	100
blocksworld	67	42	**100**	**100**	34	71	42	78	**85**	59	44	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	79	87	100	100
goldminer	66	73	67	**85**	1	3	28	**37**	34	27	8	**100**	**100**	**100**	92	1	8	55	**65**	64	64	14	89	96	99	100
grippers	91	**100**	80	**100**	98	98	**100**	**100**	98	**100**	91	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	22	27	100	100
logistics	0	**88**	**88**	**88**	4	15	**94**	85	67	**94**	**94**	1	**100**	**100**	**100**	6	21	**100**	**100**	**100**	**100**	**100**	38	45	100	100
minigrid	**77**	62	72	64	14	18	55	53	34	**61**	37	85	**96**	78	78	27	43	77	65	60	**81**	57	99	100	100	100
rovers	16	20	35	**60**	0	3	13	**20**	15	11	5	48	**60**	**60**	**60**	0	4	28	**48**	36	16	16	88	96	100	100
satellite	15	52	**63**	45	33	35	**47**	43	44	44	44	44	72	**100**	48	52	44	**48**	44	44	**48**	44	76	84	100	100
transport	**100**	0	67	67	33	0	33	**91**	67	41	33	**100**	0	**100**	**100**	**100**	0	**100**	**100**	**100**	**100**	**100**	15	26	100	100
visitall	**83**	81	82	82	32	30	**77**	65	74	70	52	**100**	**100**	**100**	**100**	51	50	**100**	**100**	**100**	**100**	54	82	89	99	100
Avg	56	68	79	**82**	39	44	60	64	**65**	59	54	81	84	**93**	89	62	54	81	**82**	81	72	68	53	61	95	98

Table 2: Percentage of solved tasks using reasoning models for the original framework by Silver et al. (2024) (`Sil`) and the re-implemented baseline (`Bas`) and our generalized planning approach with $\mathcal{N} = 3$, $\mathcal{K}_C = 6$ (`F3-6`) and $\mathcal{N} = 5$, $\mathcal{K}_C = 3$ (`F5-3`). The ablations `-MC`, `-SD` and `-CR` are based on `F3-6`. For both, we show in **bold** the best generalized planning approach for each model. The symbolic baselines were run for the same time limit as the generalized plans (45s) and for 30m (`lm` and `ff`).

This runtime efficacy increase comes at a mild price in plan quality. Comparing plan length on commonly solved tasks, the plans generated by the Python programs are only 1.1 times longer on average than those generated by `ff`.

Cost of generating the programs. Focusing on `F5-3` with DeepSeek, the generation of a program for Heavy took the least time, namely 664s on average, and for Goldminer the most time, almost 5.5h on average. Note however, that we used caching, and retrieving outputs for already processed inputs, is faster than generating them the first time. This concerns all parts of the pipeline that are shared between different variants of the framework, e.g. the NL domain description is only generated once per domain in our experiments and then retrieved from the cache for all other runs (with the exception of `Sil` which uses different prompts).

In sum, almost 15M tokens (input + output) were processed by DeepSeek and `F5-3` for generating the Python programs across all domains. This corresponds to the negligible cost of ca. 5.5 USD for the DeepSeek variant used.

Results on costumed and anonymized benchmark variants. Table 3 shows the results of our `F5-3` approach ran with GPT-4o on the anonymized and costumized variants for the domains of Silver et al. (2024).

Regarding the anonymized variants, unsurprisingly (and in line with previous work) we find that LLMs struggle, as no world knowledge can be leveraged when names in a domain carry no information.

Regarding the costumed variants however, interestingly we observe the performance of our LLM-generated programs does not degrade, with the single exception of the Spanner domain. This result indicates that the LLMs do not only replicate solutions that have already been part of the pretraining data, but are capable of actual reasoning over potential strategies for a domain, as long as the domains have a connection to real world semantics.

Domain	Avg three runs			Best run		
	original	costume	anonym	original	costume	anonym
delivery	100	100	100	100	100	100
ferry	100	100	1	100	100	2
gripper	100	67	33	100	100	100
heavy	100	100	0	100	100	0
hiking	33	100	67	100	100	100
miconic	68	67	33	100	100	100
spanner	33	0	12	100	0	35
Avg	76	76	35	100	86	62

Table 3: Percentage of solved tasks for `F5-3` with GPT-4o on the original, costumed and anonymized versions of the domains from Silver et al. (2024).

Conclusion

We show that generalized planning with LLMs can be made substantially more effective through pseudocode strategy refinement, code reflection and generating multiple code candidates. Our approach generates Python programs that achieve an average coverage of 82% across 17 domains.

In future work, it would be interesting to investigate if and how knowledge about a domain can be exploited to create a more effective set of debugging tasks and potentially extend it automatically during the generation as needed. Another important direction, given the lack of intrinsic guarantees and the fundamental limitation to polynomial-time programs, is the combination with symbolic search methods.

References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. *Proceedings of the International Conference on Automated Planning and Scheduling*, 19(1): 34–41.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. The 2025 Planning Performance of Frontier Large Language Models. arXiv:2511.09378.

Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; and Lou, Y. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. New York, NY, USA: Association for Computing Machinery. ISBN 9798400702174.

Duchnowski, A.; Pavlick, E.; and Koller, A. 2025. A Knapsack by Any Other Name: Presentation impacts LLM performance on NP-hard problems. In Christodoulopoulos, C.; Chakraborty, T.; Rose, C.; and Peng, V., eds., *Findings of the Association for Computational Linguistics: EMNLP 2025*, 6628–6651. Suzhou, China: Association for Computational Linguistics. ISBN 979-8-89176-335-7.

Eisenhut, J.; Schuler, X.; Fišer, D.; Höller, D.; Christakis, M.; and Hoffmann, J. 2024. New Fuzzing Biases for Action Policy Testing. *Proceedings of the International Conference on Automated Planning and Scheduling*, 34(1): 162–167.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. ISBN 978-3-031-00456-8.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS*. AAAI.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Holtzman, A.; Buys, J.; Du, L.; Forbes, M.; and Choi, Y. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 294–301. IEEE Computer Society.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A review of generalized planning. *The Knowledge Engineering Review*, 34: e5.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and Murthy, A. B. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*.

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.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; Gupta, S.; Majumder, B. P.; Hermann, K.; Welleck, S.; Yazdanbakhsh, A.; and Clark, P. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 46534–46594. Curran Associates, Inc.

McDermott, D. M. 2000. The 1998 AI planning systems competition. *AI magazine*, 21(2): 35–35.

Seipp, J.; Torralba, Á.; and Hoffmann, J. 2022. PDDL Generators. https://doi.org/10.5281/zenodo.6382173.

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

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18): 20256–20264.

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*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artif. Intell.*, 175(2): 615–647.

Stechly, K.; Valmeekam, K.; and Kambhampati, S. 2025. On the self-verification limitations of large language models on reasoning and planning tasks. In *The Thirteenth International Conference on Learning Representations*.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2025. Automating the Generation of Prompts for LLM-based Action Choice in PDDL Planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS'25)*.

Tang, H.; Hu, K.; Zhou, J. P.; Zhong, S.; Zheng, W.-L.; Si, X.; and Ellis, K. 2024. Code Repair with LLMs gives an Exploration-Exploitation Tradeoff. In Globerson, A.; Mackey, L.; Belgrave, D.; Fan, A.; Paquet, U.; Tomczak, J.; and Zhang, C., eds., *Advances in Neural Information Processing Systems*, volume 37, 117954–117996. Curran Associates, Inc.

Valmeekam, K.; Stechly, K.; Gundawar, A.; and Kambhampati, S. 2025. A Systematic Evaluation of the Planning and Scheduling Abilities of the Reasoning Model o1. *Transactions on Machine Learning Research*.

Wang, E.; Cassano, F.; Wu, C.; Bai, Y.; Song, W.; Nath, V.; Han, Z.; Hendryx, S.; Yue, S.; and Zhang, H. 2024. Planning In Natural Language Improves LLM Search For Code Generation. arXiv:2409.03733.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; ichter, b.; Xia, F.; Chi, E.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 24824–24837. Curran Associates, Inc.

Model Parameters

We run our experiments with the gpt-4o-2024-08-06[1] and LLama-3.3-70B-Instruct[2] non-reasoning models and the DeepSeek-V3.2-Exp[3] and Qwen3-30B-A3B-Thinking-2507[4] reasoning models. For GPT and DeepSeek we use the OpenAI and DeepSeek APIs respectively. The experiments with Llama and Qwen were run on two GPUs of type Tesla V100-PCIE-32GB.

The evaluation of the Python programs was run on the same processors as the symbolic planners, i.e. on Intel Xeon E5-2687W processors.

Parameter	GPT-4o	DeepSeek	Llama	Qwen
context window	128000	128000	128000	262144
temperature	0	NA	0.7	0.6
max tokens	16384	32000	20000	81920
top p	NA	NA	0.8	0.95
top k	NA	NA	20	20
presence pen.	NA	NA	0	0
seed	1	NA	NA	NA

Table 4: Parameters of the LLMs for the generation.

Additional Results.

Runtimes and Plan Lengths We compare the time required for generating a plan and the length of the generated plans by `ff` and by `F5-3` for all commonly solved tasks. We report the average over all per-task ratios, computed by dividing the runtime / plan length for `F5-3` by the corresponding value for `ff`. Additionally, we report the percentage of tasks for which the Python program is faster than `ff` and the percentage of tasks for which the Python program generates shorter plans.

Table 5 shows the result for the comparison to `F5-3` (best run) with each of the four tested LLMs. All four LLMs generate Python programs that are faster for a large majority of tasks. The programs generated by DeepSeek are the fastest - relative to `ff`- requiring only 0.36 time the time of `ff` on average while the programs generated by Llama have runtimes closer to `ff`.

Focusing on the plan length, we observe that `ff` generates plans that are on average between 1.1 (DeepSeek) and 1.57 (GPT) times shorter.

Model	Runtime		Plan length	
	Avg Ratio	F5-3 faster	Avg Ratio	F5-3 shorter
GPT-4o	0.42	96%	1.57	32%
DeepSeek	0.36	97%	1.10	51%
Llama	0.89	88%	1.34	37%
Qwen	0.43	96%	1.29	46%

Table 5: Comparison of the runtime and generated plan length for the best Python program generated using the `F5-3` configuration and `ff` on commonly solved tasks.

[1]https://platform.openai.com/docs/models/gpt-4o

[2]https://huggingface.co/nvidia/Llama-3.3-70B-Instruct-FP8

[3]https://api-docs.deepseek.com/news/news250929

[4]https://huggingface.co/Qwen/Qwen3-30B-A3B-Thinking-2507

Error distributions. Table 6 provides an overview of the distribution of error types. Considering all three runs for each configuration, we report the number of runs where the generation itself stopped with an error. All these errors were due to reaching the maximum number of output tokens or the maximum number of tokens the model can process, i.e. the context window limit.

For all runs that finished without an error, we consider all final programs that resulted in an error for at least one evaluation task and computed the percentage of runs for which each error type occurred for at least one task.

Model	Python except.	Timeout	Invalid actions	Goal not satisfied	Incompl. run
GPT-4o	23%	14%	37%	27%	0
DeepSeek	18%	15%	35%	32%	11
Llama	13%	15%	35%	32%	3
Qwen	13%	12%	51%	24%	11

Table 6: The number of runs (out of all configurations and domains) for which the generation of the programs stopped with and error (Incompl. run) and the percentage of runs where one of the four error types occurred for at least one task out of all runs with any error.

Datasets

Sources. For our experiments, we focus on domains that have previously been used in research on LLMs in the context of classical planning. In particular, we use the domains from Silver et al. (2024)'s generalized planning experiments and Stein et al. (2025)'s LLM action-choice experiments. Table 7 shows for each of the domains the sources of the tasks we include in our experiments. All tasks included in our experiments are solvable.

Instance generators. The right column of Table 7 shows the origin of the instance generators that we used to generate additional tasks for some of the domains, and that we used for the manual evaluation of the generalization power of our generalized plans. For the domains from Silver et al. (2024), we mostly focused on their generators[5] and for all other domains, except Minigrid, we used the generators from the PDDL-Generators repository[6] (Seipp, Torralba, and Hoffmann 2022).

For Blocksworld, we renamed two of the predicates in the newly generated problem files to match the domain file used by Stein et al. (2025) ("on-table" to "ontable" and "arm-empty" to "handempty"). Additionally, we modified all domains and tasks with non-uniform action costs or functions.

Debugging and eval tasks. For the each domain, we randomly select 6 tasks that are small with respect to the sized of the evaluation tasks as debugging tasks. In particular, we only consider tasks for which the number of objects and optimal plan length of each debugging task is among the 16 smallest values of object number and plan length in the overall dataset. One exception is the Beluga domain for which

[5]https://github.com/tomsilver/llm-genplan

[6]https://github.com/AI-Planning/pddl-generators

Domain	Source of tasks	Instance Generators
delivery	Silver et al. (2024)	Silver et al. (2024)
ferry	Silver et al. (2024)	Silver et al. (2024)
	Stein et al. (2025) Generated by us	Seipp, Torralba, and Hoffmann (2022)
gripper	Silver et al. (2024)	Silver et al. (2024)
	IPC gripper98	Seipp, Torralba, and Hoffmann (2022)
heavy	Silver et al. (2024)	Silver et al. (2024)
hiking	Silver et al. (2024)	Silver et al. (2024)
miconic	Silver et al. (2024)	Silver et al. (2024)
spanner	Silver et al. (2024)	Silver et al. (2024)
beluga	Eisenhut et al. (2024)	Eisenhut et al. (2024)
blocks.	Stein et al. (2025) IPC blocks00 Generated by us	Seipp, Torralba, and Hoffmann (2022)
goldminer	Stein et al. (2025)	Seipp, Torralba, and Hoffmann (2022)
grippers	Stein et al. (2025) Generated by us	Seipp, Torralba, and Hoffmann (2022)
logistics	Stein et al. (2025) IPC logistics98 IPC logistics00	Seipp, Torralba, and Hoffmann (2022)
minigrid	https://github.com/ bonetblai/mini-grid/	https://github.com/bonetblai/mini-grid/
rovers	Stein et al. (2025) Generated by us	Seipp, Torralba, and Hoffmann (2022)
satellite	Stein et al. (2025) Generated by us	Seipp, Torralba, and Hoffmann (2022)
transport	IPC transport08 IPC transport11 IPC transport14	Seipp, Torralba, and Hoffmann (2022)
visitall	Stein et al. (2025) IPC visitall11 IPC visitall14 Generated by us	Seipp, Torralba, and Hoffmann (2022)

Table 7: The origin of all tasks that we used for our experiments and the instance generators that we used for the manual evaluation and for generating additional data for some of the domains.

the optimal planner (`lm`) did not solve any task. We therefore based the debugging task selection on the lengths of the plans generated by the satisficing planner (`ff`).

Table 8 gives an overview of the sizes of all tasks from our experiments. It shows the average length of the plans generated by the optimal planner and the satisficing planner for the debugging and evaluation tasks, as well as the average number of objects for both sets of tasks. Additionally, we include the minimum and maximum values for the plan lengths and number of objects of the evaluation tasks. The overview shows that the tasks on which we evaluate our generate programs include tasks that are much larger than the ones used during the generation of the generalized plans (i.e. as examples and for debugging). Figure 8 illustrates the distribution of the number of objects per debugging and evaluation task for the Miconic and Logistics domains.

Costumed variants. Duchnowski, Pavlick, and Koller (2025) showed for NP-hard problems that different ways of phrasing the same mathematical problem impact the performance of LLMs. We adapt their idea of generating costumed

domain	N	Optimal Plan Length (`lm`)				Satisficing Plan Length (`ff`)				Number of objects			
		debug	eval			debug	eval			debug	eval		
		avg	avg	min	max	avg	avg	min	max	avg	avg	min	max
Domains from Silver et al. (2024)													
delivery	30	10	None	None	None	12	96	79	115	10	62	50	73
ferry	275	7	28	4	56	7	104	4	301	8	45	5	116
gripper	53	9	35	15	52	10	77	18	165	28	53	10	81
heavy	34	6	128	4	209	6	128	4	209	6	128	4	209
hiking	28	7	13	2	26	7	13	2	26	97	108	80	121
miconic	34	23	27	11	66	25	63	12	186	20	53	9	104
spanner	34	16	33	10	52	16	37	10	52	19	47	13	64
Additional Domains													
beluga	61	None	None	None	None	12	48	10	159	17	27	15	34
blocksworld	191	7	18	0	42	9	38	0	218	4	9	3	20
goldminer	115	10	16	5	32	11	42	5	370	7	17	4	49
grippers	131	5	18	4	54	5	126	4	340	11	63	9	144
logistics	178	7	17	0	48	7	59	0	361	12	54	9	438
minigrid	74	6	12	0	83	6	13	0	122	25	34	10	96
rovers	25	8	14	6	36	8	15	8	37	12	16	11	28
satellite	25	6	10	6	23	6	17	6	79	8	31	9	110
transport	53	10	22	16	36	10	55	16	152	14	37	18	75
visitall	193	5	15	0	50	5	87	0	2308	6	31	1	324

Table 8: The number of tasks for each domain (N), and the average (avg), minimum (min) and maximum (max) values of the plans derived by the `lm` and `ff` symbolic planners and number of objects for the evaluation tasks (eval) as well as the average values of the debugging tasks (debug). Tasks for which the symbolic planner did not find a plan were left out in the computation of the average plan length values.

versions to planning domains. For each actions, predicate, object and type name we manually create a new replacement name, hence generating a new variant of a domain that preserves the exact logical structure of the original domain. We provide a brief description of the original and costumed versions.

Ferry. Cars must be transported between different locations using a ferry that can carry only one car at a time.

Costumed ferry. A squirrel needs to jump between different trees in order to move nuts to the goal tree. It can only carry one nut in its paws.

Delivery. The goal is delivery newspapers to different locations. All newspapers need to be picked up at the home based and be carried to the locations that want a newspaper.

Costumed delivery. The goal is to deliver seedlings to places that are planning to create a garden. All seedlings need to be collected at the nursery and driven to the target location where the seedling is planted.

Gripper. A robot needs to carry balls between rooms. The robot has two grippers, each can carry one ball.

Costumed gripper. The goal is to sail between different hideouts in order to find chests at their initial hideout and hide them at the goal hideout. The boat has space for two chests, one at the port and one at the starboard side.

Heavy. Objects need to be placed on top of each other in a box such that the heaviest item is the bottom-most one and no object is placed on an object that is lighter than itself.

Costumed heavy. A number of tasks needs to be scheduled such that the easiest task is scheduled first and no task is scheduled after a more difficult task.

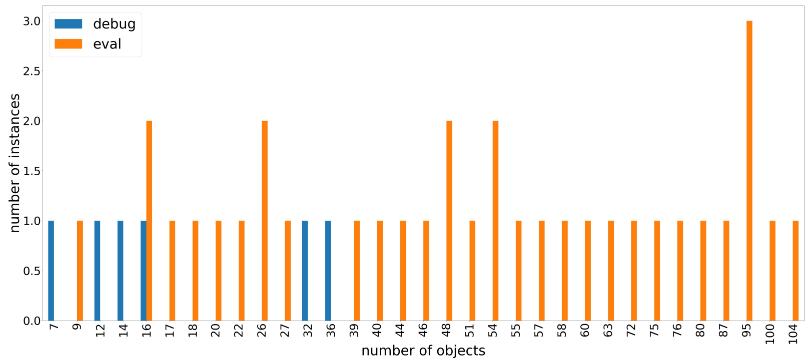
Hiking. The goal is to navigate from an initial location in a 2D grid to a goal location. Some locations are water or a hill. Moving to a hill location requires a climbing action instead of walking and it is not possible to move to locations with water. There is one defined trail leading from the initial to the goal location but other paths are possible.

Costumed hiking. The goal is to navigate from an initial location in a 2D grid to a goal location. Some locations are colored white, black or red. Moving to the white locations requires a jump actions instead of moving and it is not possible to move to a black location. All red locations form a path from the initial to the goal location.

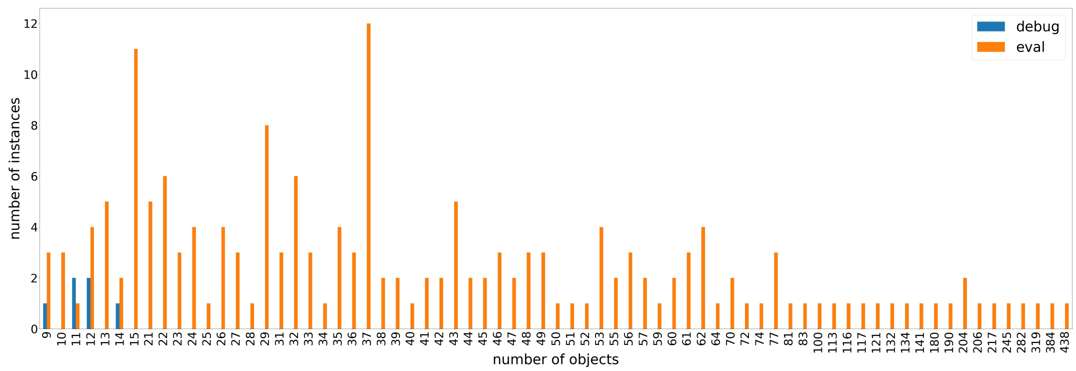
Miconic. The goal is to transport passengers between different floors in a building. Passengers can be picked up at their initial floor and dropped off at another floor. There can be several buildings and passengers can only move within the same building.

Costumed miconic. The goal is submit a process when a machine is in a specific mode. There can be several machines and each machine has different modes that are ordered. It is possible to switch the mode down to lower modes or up to higher modes. Each process requires a specific mode in order to be set up before the mode can be switched to the goal mode for submitting the process.

Spanner. An agent must tighten all nuts using spanners. Each spanner can only be used once to tighten a nut. All



(a) Miconic domain



(b) Logistics domain

Figure 8: Number debugging tasks (blue) and number of evaluation tasks (orange) with a specific number of objects (x-axis).

locations are connected in the form of a one-way path and all nuts are at the last location. The agent needs to move from the its start location to the last location and pick up the spanners needed for tightening the nuts along the way.

Costumed spanner. A squirrel must feed its babies with nuts. There is a number of tree hollows along a tree that contain nuts and the squirrel can only move from the bottom of the tree to the top of the tree where the babies are located. It needs to climb upt the tree and pick up the nuts needed for feeding each baby one nut.

Feedback Types for Debugging

Code debugging. For the debugging of the generated Python programs, the LLM is provided feedback about the outcome of running the program on the debugging task. In particular, we provide feedback on one of the tasks for which the program did not return any output or an incorrect output. For our approach (but not for the baseline) we also include the tasks for which the program returned the correct output together with the actual outputs. Figure 9a and 9b show the templates used for creating the feedback messages if at least one task was solved by the program and if none was solved respectively.

We provide the information about the failed task (and the solved ones if available) in Python format. If the program returned an incorrect output, this output is included in the feedback prompt (part between the dashed lines). The actual feedback message depends on the type of error that occurred.

Table 9 shows the feedback messages generated in our pipeline for the different types of errors. Following Silver et al. (2024) we differentiate between timeouts (1), Python exceptions (2), an output of an invalid type (3) and outputs that are not a valid plan for the task (4). We make the feedback message for outputs that are not valid plans for the input task more informative by incorporating the feedback generator from Stein et al. (2025) (see Table 9, 4.1 - 4.6). This approach allows us to give more precise feedback, e.g. for the wrong number of action parameters, or parameters not matching objects defined by the specific task.

Pseudocode debugging. Figure 9c shows the template used for creating the feedback prompt for the debugging of the pseudocode strategy. The prompt does not include the definition of the task for which a wrong plan was generated because the NL task description is already included in the context of the LLM. The different types of errors considered for the pseudocode debugging and the corresponding feedback messages are the same as 4.1 - 4.6 in Table 9.

Manual Evaluation of Generalization Power

We manually evaluated the generalization power of all generalized plans generated by `F5-3` using GPT-4o and `F5-3` us-

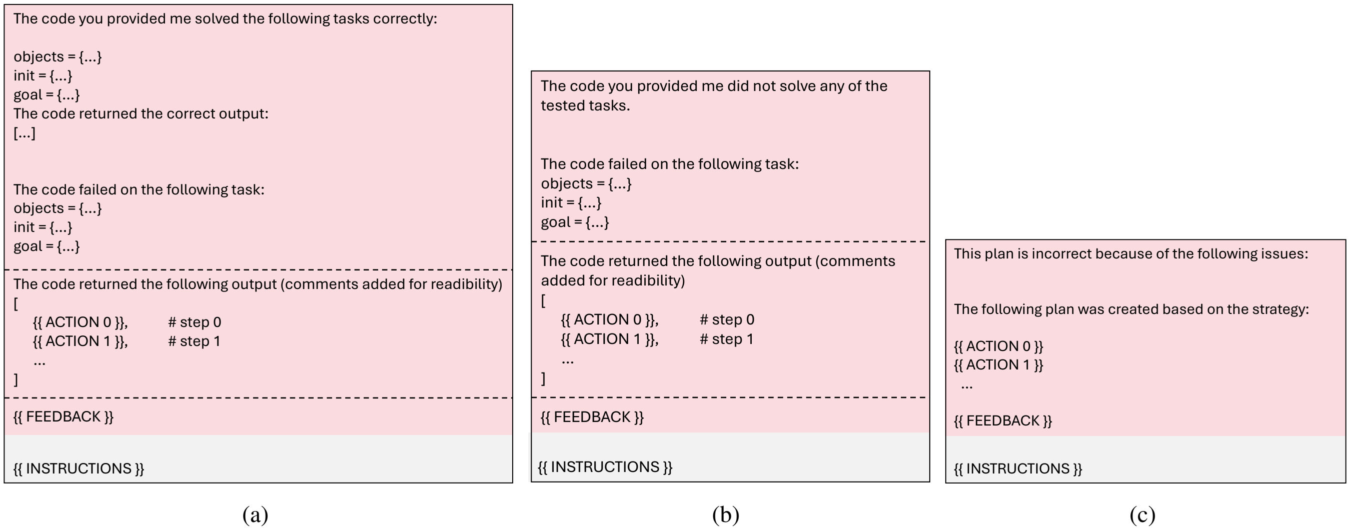


Figure 9: 9a and 9b show the templates for the reflection prompts provided during the code debugging step when some of the debugging tasks were solved by the program (9a) and none were solved (9b). The part between the dashed lines is only included if running the program on the task did actually generate an output. 9c shows the template for the reflection prompt provided during the debugging of the pseudocode strategy.

ing DeepSeek-R1 that achieved 100% coverage on our evaluation datasets. We find that for all 12 domains where `F5-3` with GPT-4o generated at least one program that solved all evaluation tasks, that program generalizes to all tasks that can be generated with the respective instance generators. The some holds for the 14 domains for which `F5-3` generated at least one program that solved all evaluation tasks when using DeepSeek-R1.

Here, we provide more details about the results of the manual analysis. We report the details for the programs generated by GPT-4o but the observations for the programs generated by DeepSeek-R1 are similar. For each domain, we provide a summary of the types of tasks generated by the instance generator and briefly describe the most relevant parts of the evaluated Python program. If several programs achieved 100% coverage, we evaluated all of them but provide a description only for one of them. All evaluated programs are included in the supplementary material. We report whether the evaluated programs can generalize to all tasks generated by the instance generator. Additionally, we provide a brief overview of the generalization beyond the tasks generated by the specific instance generators. For both, we only consider solvable tasks.

Delivery

Instance generator. The generator creates tasks with a specified number of locations. “location-0” is always the home base and always the start location. Newspapers must be picked up in the home base, moved to a location that wants a paper and then be delivered. The locations that want a newspaper are randomly distributed over the locations. The goal specifies that each location that wants a paper should be satisfied in the end.

Program description (seed 1). First, the home base, all safe locations, all locations that want newspapers and all unpacked newspapers are determined. Afterwards, the code loops over all locations that want a newspaper and adds all actions required for delivering a newspaper from the home base to that location to the solution.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of locations, newspapers and locations that want packages.

Generalization beyond generator. If the start location is different from the home base, two of the programs will not generate correct plans as they set the start location to the home base. Additionally, all three programs will fail if the goal specifies the final position of the deliverer. However, all programs generalize to tasks where the home base, i.e. the location of the newspapers, is different from “location-0”.

Ferry

Instance generator. For all generated tasks, there is a ferry, a number of cars, and a number of possible locations. The cars are randomly distributed across locations. The ferry is initially empty. The goal specifies for each car a goal location, which can be identical to the initial location.

Program description (seed 1). First, all cars and the initial location of the ferry are determined. Then the program loops over all cars and checks whether the current and goal location are the same. If not, all actions for transporting the car to the goal location are added to the solution.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of cars and locations.

	Feedback Messages	Error Type
1	The code was interrupted because it did not terminate within the time limit ({{ TIME LIMIT }} seconds). This is likely caused by an infinite loop. Please check the loops again.	Execution of the program timed out.
2	The code raised the following exception: {{ TRACEBACK without file paths }}	Execution of the program raised a Python exception.
3	The code returned {{ OUTPUT }} which is not a list of actions. Please make sure that your code returns a list of actions, i.e. of type List[str].	The returned output does not have the correct type.
4.1	The action {{ ACTION X }} at step {{ X }} is not executable because {{ STRING }} is not an available object in this task.	The returned plan contains parameters that do not match objects in this task.
4.2	The action {{ ACTIONX }} at step {{ X }} is not executable because {{ ACTIONX }} does not match any possible actions.	The returned plan contains actions that are not part of the domain.
4.3	The action {{ ACTIONX }} at step {{ X }} is not executable because {{ ACTIONX }} requires exactly {{ CORRECT NUMBER PARAMETERS }} objects as arguments but {{ INCORRECT NUMBER PARAMETERS }} were given.	The returned plan contains actions with wrong number of parameters.
4.4	The action {{ ACTIONX }} at step {{ X }} is not executable because the preconditions of the action are not satisfied: At that specific step − it is not the case that {{ PRECONDITION }}	The returned plan contains actions with unsatisfied preconditions which are non-static predicates, i.e. there are actions that can make the precondition true.
4.5	The action {{ ACTIONX }} at step {{ X }} is not executable because the preconditions of the action are not satisfied: In this task instance − it is not the case that {{ PRECONDITION }}	The returned plan contains actions with unsatisfied preconditions which are static predicates, i.e. there is no action that can make this precondition true.
4.6	The generated plan does not reach the goal: The following needs to be false but is true after executing all actions: {{ NOT SATISFIED NEGATIVE GOAL FACTS }} The following needs to be true but is false after executing all actions: {{ NOT SATISFIED POSITIVE GOAL FACTS }}	The returned plans does only contain applicable actions but not all goal facts are satisfied in the end.

Table 9: The feedback messages generated for the different types of errors.

Generalization beyond generator. The programs cannot generalize to tasks where a car is initially already on the ferry. Additionally, the programs will fail on tasks where the goal specifies a target position of the ferry or if a car needs to be on the ferry.

Gripper

Instance generator. The instance generator generates initial states where the balls are randomly distributed over all rooms. The robot always starts in "room-0" and has two grippers that are initially free. The generated goals specify for some of the balls one room as the goal location, i.e. the goal is always to transport some, but not necessarily all, balls to the goal room.

Program description (seed 1). First, the initial location of the robot is determined and afterwards the program loops over the goal input set and checks for goal facts starting wit "at". For each of those, it checks for the current location, moves the robot there, frees up a gripper and continues with the remaining actions required for moving the ball to the goal location.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of balls, rooms and goal locations.

Generalization beyond generator. All three programs cannot generalize to tasks where the goal specifies a target location for the robot. Additionally, they do not generalize to tasks where balls are already being carried in the initial state.

Heavy

Instance generator. For all generated tasks, the box is initially empty and all objects are unpacked. Additionally, the initial state fully defines the heavier relation between all objects, i.e. if there are n objects, then the heaviest object is heavier than $n-1$ objects, the second heaviest is heavier than $n-2$, and so on. The goal is that every object is packed in the box.

Program description. All three programs organize the objects into a list, sorted in descending order based on the frequency with which each object is heavier than another object. To generate the plan, the objects are stacked on top of each other in exactly that order.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of objects.

Generalization beyond generator. All programs pack all available objects into the box. If the goal state specifies that some objects should not be packed the programs will not be able to generalize valid plans.

Hiking

Instance generator. The instance generators generate tasks that are grids where some cells are of type dirt, water or hill. In the initial state, these types are randomly distributed over the cells and a trail from the initial start position to the target position is generated, consisting of cells which are not of type water. The goal is to reach the target position. This can be achieved by simply following the trail, or finding another path through the grid.

Program description. First, the start and target position are determined. Afterwards, the program loops over all facts of the initial state to find the location that is adjacent to the current one and on the trail. If that location is a hill, the action for climbing to the location is added to the solution, otherwise the action for walking is added. This process is continued until the current location equals the goal location.

Generalization. The evaluated program solves every task generated by the instance generator. It generalizes to tasks with an arbitrary grid size, as long as there exists a trail between the initial and target position.

Generalization beyond generator. If the trail is interrupted, or the initial or target position are not part of the trail, then the program will fail.

Miconic

Instance generator. The generator generates tasks with a specified number of buildings. Every building has the same number of floors and passengers. The passengers are then randomly distributed across the floors of a building. Furthermore, every building has one lift which is initially on a random floor. The goal is to bring all passengers from their initial floor to their destination floor within the same building using the lifts.

Program description (seed 2). First, the position of the lifts is determined, as well as the connections between the floors, i.e. which floors are in the same building, and the current and destination floor for each passenger. The program then loops over all passengers and their initial location, moves the lift that is in the same building to the passenger and adds all remaining actions for moving the passenger to the destination floor.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of buildings, passengers and floors.

Generalization beyond generator. If the goal specifies a floor as target location for any lift, then the all three programs will fail. Additionally, if the names of the floors do not follow the naming scheme of FLOOR−BUILDING (e.g. f1-b0) the programs cannot correctly determine anymore which floors belong to the same building.

Spanner

Instance generator. The instance generator generates tasks with a specified number of locations, and two special locations, the shed and the gate. In the initial state, a man is at the shed and a number of loose nuts is at the gate. An arbitrary number of spanners is distributed across the locations and all locations are connected such that they form a one-way path from the shed to the gate. The goal is to tighten all loose nuts.

Program description. First, a direct path from the gate to the shed is determined using a breadth-first search approach. Then the man is moved from location to location along this path. If there are spanners at an location they are all picked up. After arriving at the gate, the program loops over the loose nuts and selects one spanner after the other to tighten each nut.

Generalization. The evaluated program solves every task generated by the instance generator. It generalizes to tasks with an arbitrary number of nuts, locations and spanners.

Generalization beyond generator. The program will fail if the nuts are not only at the gate location or if the goal specifies a target location of the man. Additionally, the program would fail if the connections between locations would allow moving in more than one direction, and the man would need to move between locations in a way different from a direct path between shed and gate.

Grippers

Instance generator. The instance generator generates initial states where the robots and balls are randomly distributed over all rooms, and all robots have two grippers that are initially free. The generated goals specify for each ball one room as the goal location, i.e. the goal is always to transport each ball, that is not already at its goal location, to the goal room. In contrast to Gripper above, there can be several robots.

Program description (seed 1). First, all robots and objects (i.e. balls) are determined. The program then loops over all facts in the goal input set and checks for goal facts starting with "at". For each of them, it checks whether the initial location of the specified ball is identical to the goal location. If not, it selects a robot, moves it to the initial location of the ball and frees up a gripper if necessary, and continues with the remaining actions required for moving the ball to the goal room.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of balls, rooms, and robots.

Generalization beyond generator. All three programs cannot generalize to tasks where the goal specifies a target location for any robot. Additionally, they do not generalize to tasks where balls are already being carried in the initial state.

Logistics

Instance generator. In all initial states generated by the instance generator, all cities have the same number of locations and each city has one airport. A specified number of airplanes are randomly distributed across all airports. Each task also includes a specified number of trucks, with the only condition that there are at least as many trucks as cities. There can be multiple trucks and airplanes in the same location. A specified number of packages is distributed over all possible locations. The goal specifies for each package a goal location which can be identical to the initial location.

Program description (seed 2). First, the locations of all trucks and a randomly selected airplane are determined. Afterwards, the program loops over all facts in the goal input set and identifies all goal facts that are about the location of packages. For each of them, it checks whether the current location and goal location are in the same city or not. For both cases, the required vehicles and steps for moving the package from the current location to the goal are then determined.

Generalization. All three evaluated programs solve every task generated by the instance generator. They generalize to tasks with an arbitrary number of vehicles, cities, locations, and packages.

Generalization beyond generator. All three programs cannot generalize to tasks where the location of a vehicle is part of the goal or where the target location of a package is inside a vehicle. If some packages are initially in a vehicle the generalized plans will fail as well.

Satellite

Instance generator. The initial state of the tasks generated defines which instruments a satellite has on board and the modes the instruments support. Furthermore, it gives the calibration targets for the instruments. Additionally, all satellites have "power_avail" which is needed to switch on instruments, and satellites are pointing in some direction. The goal is to take images of some observations and, in some instances, to additionally point the satellite to a specific direction (target or observation).

Program description. The program first loops over all facts in the goal input set and checks for goals that are about taking images. It then determines the required mode, instrument, satellite and calibration target and adds all steps taking the picture and turning off the instruments afterwards. After taking care of all pictures, the code loops over all goal facts that are about pointing in some direction and adds the actions required for turning the satellites accordingly.

Generalization. The evaluated program solves every task generated by the instance generator. It generalizes to tasks with an arbitrary number of satellites, modes, targets, observations and maximum number of instruments per satellite.

Generalization beyond generator. If in the initial state, there is no "power_avail" for some satellites, i.e. an instrument has "power_on" from the beginning, the program might fail depending on whether that specific instrument is used first or not. The order in which the instruments are used depends on the order in which the program iterates over an (unordered) set.

Transport

Instance generator. Each task generated by the instance generator consists of a specified number of locations, vehicles (trucks), packages, and vehicle capacity. Initially, packages and trucks are distributed across the available locations and trucks are assigned a capacity of at least 2. Additionally, there exist roads between locations. The goal is to bring every package to its goal location.

Program description. First, all goal locations of packages are determined. The program then loops over all packages and their initial locations, checks for a vehicle with capacity that is not equal to "capacity-0" and finds a path from the location of the vehicle to the package using a breadth-first search based approach. Then the vehicle is move there, the package is picked-up, the capacity is updated and the path for getting to the goal location is determined in the same way. The package is dropped at the goal location, the capacity is updated and the loop continues with the next package.

Generalization. The evaluated program solves every task generated by the instance generator. It generalizes to tasks with any number of trucks, packages, locations, cities, roads, and capacity numbers.

Generalization beyond generator. Similar to Logistics, the program cannot generalize to tasks where the location of a vehicle is part of the goal or where the target location of a package is inside a vehicle. Additionally, if a package is initially already in a truck the program will fail as well.

Visitall

Instance generator. The generator creates tasks consisting of a grid of some size n x m. The initial state consists of the random location of the robot and the connections between adjacent locations. It is possible to define unavailable locations, i.e. locations for which no connections to other locations in the grid are defined. Depending on the ratio of cells in the goal state, the goal is to reach either all locations or a random subset of all locations.

Program description. First, the initial location of the robot and all connections between locations are determined, as well as all already visited and not yet visited locations. The program then loops over all locations adjacent to the current location to find a not yet visited one. If one is found, the robot moves there. Otherwise, the program backtracks to the last visited location that has still unvisited adjacent locations (and moves the robot there), and continues from there.

Generalization. The evaluated program solves every task generated by the instance generator. It generalizes to tasks with an arbitrarily large grid, any number of unavailable locations, and any ratio of locations in the goal state.

Generalization beyond generator. The program would fail only for tasks where the goal specifies a specific target position of the robot.

Example Generalized Plans

Generalized Plan for Gripper (DeepSeek, F5-3)

```python
from typing import List, Tuple, Set

def generate_solution(objects: Set[str], init: Set[Tuple], goal: Set[Tuple]) -> List[str]:
    solution = []

    # Extract initial state information
    robby_location = None
    ball_locations = {}
    grippers = []
    free_grippers = []

    for fact in init:
        if fact[0] == 'at-robby':
            robby_location = fact[1]
        elif fact[0] == 'at' and len(fact) == 3:
            ball_locations[fact[1]] = fact[2]
        elif fact[0] == 'gripper':
            grippers.append(fact[1])
        elif fact[0] == 'free':
            free_grippers.append(fact[1])

    # Extract goals
    goals = []
    for fact in goal:
        if fact[0] == 'at' and len(fact) == 3:
            goals.append((fact[1], fact[2]))

    # Process each goal
    for ball, target_room in goals:
        current_room = ball_locations[ball]

        # Skip if ball is already in target room
        if current_room == target_room:
            continue

        # Move Robby to ball's current room if needed
        if robby_location != current_room:
            solution.append(f"(move {robby_location} {current_room})")
            robby_location = current_room

        # Pick up the ball with a free gripper
        if free_grippers:
            gripper = free_grippers.pop(0)
            solution.append(f"(pick {ball} {current_room} {gripper})")

            # Move Robby to target room if needed
            if robby_location != target_room:
                solution.append(f"(move {robby_location} {target_room})")
                robby_location = target_room

            # Drop the ball
            solution.append(f"(drop {ball} {target_room} {gripper})")
            free_grippers.append(gripper)

            # Update ball location
            ball_locations[ball] = target_room
```

```

    return solution
```

Generalized Plan for Blocksworld (DeepSeek, F5-3)

```
from typing import List, Tuple, Set

def generate_solution(objects: Set[str], init: Set[Tuple], goal: Set[Tuple]) -> List[str]:
    solution = []

    # Initialize current state
    current_clear = set()
    current_ontable = set()
    current_hand = None
    current_on = {}

    for fact in init:
        if fact[0] == 'clear':
            current_clear.add(fact[1])
        elif fact[0] == 'ontable':
            current_ontable.add(fact[1])
        elif fact[0] == 'handempty':
            current_hand = None
        elif fact[0] == 'holding':
            current_hand = fact[1]
        elif fact[0] == 'on':
            current_on[fact[1]] = fact[2]

    # Phase 1: Move all blocks to the table
    while any(obj not in current_ontable for obj in objects):
        if current_hand is not None:
            # Put down the block in hand
            block = current_hand
            solution.append(f"(put-down {block})")
            current_ontable.add(block)
            current_clear.add(block)
            current_hand = None
        else:
            # Find a clear block that is not on the table
            found_block = None
            for block in current_clear:
                if block not in current_ontable:
                    found_block = block
                    break
            if found_block is None:
                break

            base_block = current_on[found_block]
            solution.append(f"(unstack {found_block} {base_block})")
            current_hand = found_block
            current_clear.remove(found_block)
            current_clear.add(base_block)
            del current_on[found_block]

    # Phase 2: Build goal stacks
    goal_on = {}
    for fact in goal:
        if fact[0] == 'on':
            goal_on[fact[1]] = fact[2]
```

```python

    # Compute goal stacks
    goal_ontable = objects - set(goal_on.keys())
    goal_children = {}
    for top, bottom in goal_on.items():
        goal_children[bottom] = top

    stacks = []
    visited = set()
    for block in goal_ontable:
        if block in visited:
            continue
        stack = [block]
        visited.add(block)
        current = block
        while current in goal_children:
            next_block = goal_children[current]
            stack.append(next_block)
            visited.add(next_block)
            current = next_block
        stacks.append(stack)

    # Build each stack from bottom up
    for stack in stacks:
        for i in range(1, len(stack)):
            solution.append(f"(pick-up {stack[i]})")
            solution.append(f"(stack {stack[i]} {stack[i-1]})")

    return solution
```