Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

Improved Generalized Planning with LLMs Through Strategy Refinement and Reflection

Katharina Stein[1], **Nils Hodel**[1], **Daniel Fišer**[2], **Jörg Hoffmann**[1,3], **Michael Katz**[4], **Alexander Koller**[1]
[1]Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
[2]Aalborg University, Denmark
[3]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[4]IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
{kstein,koller,niho}@lst.uni-saarland.de, danfis@danfis.cz, hoffmann@cs.uni-saarland.de, Michael.Katz1@ibm.com

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 and Dataset — https://github.com/coli-saar/genplan-strategy-refine
Extended version — https://arxiv.org/abs/2508.13876

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 a strategy 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

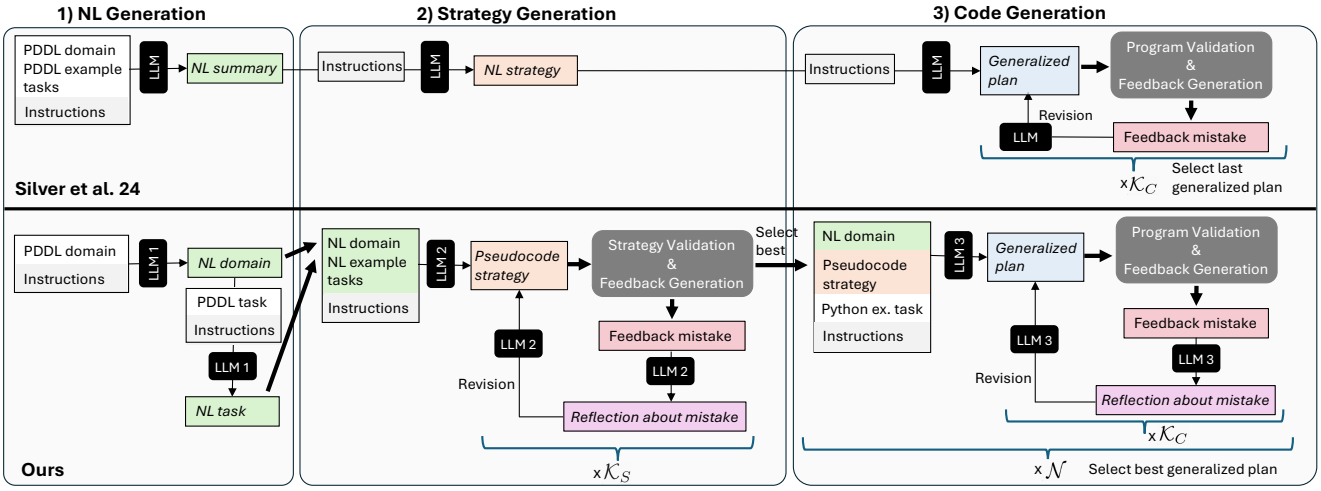

Figure 1: Overview of the framework of Silver et al. (2024) (top) and our framework (bottom). The main three parts for both are the NL generation, the strategy generation and lastly the code generation, i.e. the generation of the generalized plan.

main contribution is an approach that allows us to automatically validate and refine the strategy before passing it to the 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 feedback (coral color) and the instruction to fix the code. The feedback includes details about the error that occurred and the PDDL definition 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 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 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 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 given the feedback, we add a reflection step, inspired by approaches that let LLMs reflect on ways to improve over previous outputs (e.g. Madaan et al. 2023; Shinn et al. 2023). We combine the feedback and the generated plan with instructions to reflect on the part of the pseudocode that caused

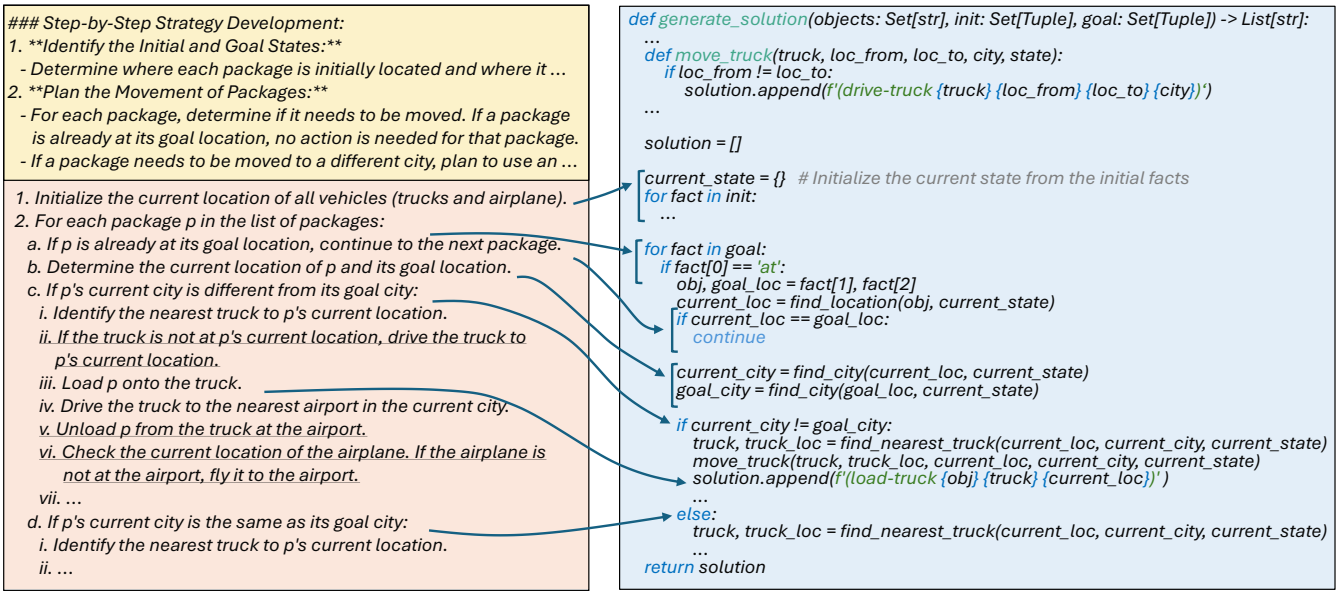

Figure 3: Left: Excerpt of the LLM output for generating a pseudocode strategy for the Logistics domain, consisting of a CoT (top, yellow color) and the pseudocode (bottom, peach color). Underlined steps were initially missing and added during debugging. Right: excerpt of a generalized plan implementing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

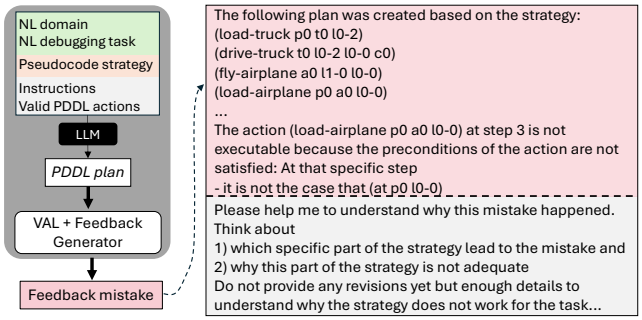

Figure 4: Left: approach for generating a PDDL plan for a debugging task based on the pseudocode and obtaining a feedback message (coral color). Right: example prompt for the reflection step consisting of a feedback message for a Logistics task and instructions (grey color).

the mistake and the reason why that part is incorrect. After generating the reflection response based on that prompt, the LLM is asked to think step-by-step and correct the pseudocode. 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. The pseudocode that resulted in the highest number of solved tasks is selected as the pseudocode for the code generation.

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 receives about the mistake is always correct with respect to the plan. Additionally, if the pseudocode is

1. **Specific Part of the Strategy**:
- ... 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 ...
- 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 ...
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

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 an NL description for each debugging task. We obtain them 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. Then, the NL

```
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).

description of each debugging task is generated based on its PDDL definition and the PDDL and NL domain descriptions. We also use the NL description of the domain and two debugging tasks as input for the pseudocode generation.

Adding Reflection to Code Debugging

While LLMs perform well on generating short, single-function 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 and inputs in the extended version.

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 extended paper version 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 extended version for details). We use the same prompts for 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 re-implementation 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* 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 on 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 extended paper version.

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 tasks and indeed solve all tasks that 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 any number of cars and locations if the cars are initially not on the ferry and the ferry location is not part of the goal (i.e. 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

Domains	Avg coverage three runs														Coverage best run													
	GPT-4o							Llama3.3							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.

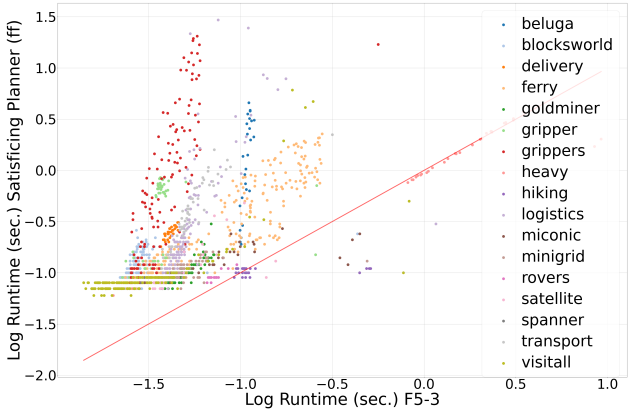

Figure 7: Runtime (log scale) of the best generalized plan by `F5-3` with DeepSeek (x-axis) and of `ff` (y-axis) for each commonly solved task. Diagonal is plotted in red.

(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 available in the extended paper version. 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; see extended version for more comparisons).

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`.

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).

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).

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 run with GPT-4o on the anonymized and costumed 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.

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.

Acknowledgments

The work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the project number 232722074 – SFB 1102. It was also funded in part by the Deutsche Forschungsgemeinschaft - GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" - project number 471607914.

References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 34–41. AAAI Press.

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

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 2024)*. Association for Computing Machinery.

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. Association for Computational Linguistics.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI Press.

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. In *Proceedings of the Eighth International Conference on Learning Representations (ICLR 2020)*. OpenReview.net.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *Proceedings of the 16th International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, 294–301. IEEE.

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 *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, 22895–22907. JMLR.org.

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 (NeurIPS 2022)*, 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 (NeurIPS 2023)*, volume 36, 46534–46594. Curran Associates, Inc.

McDermott, D. M. 2000. The 1998 AI planning systems competition. *AI magazine*, 21(2): 35–55.

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 (NeurIPS 2023)*, volume 36, 8634–8652. Curran Associates, Inc.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J.; Pack Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In Dy, J.; and Natarajan, S., eds., *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, 20256–20264. AAAI Press.

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 Workshop on Foundation Models for Decision Making*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 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 *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR 2025)*. OpenReview.net.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2025. Automating the Generation of Prompts for LLM-based Action Choice in PDDL Planning. In Lipovetzky, N.; Sardina, S.; Harabor, D.; and Ramirez, M., eds., *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2025)*, 250–259. AAAI Press.

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 *Advances in Neural*

Information Processing Systems (NeurIPS 2024), 117954–117996.

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 [cs.LG].

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 (NeurIPS 2022)*, volume 35, 24824–24837. Curran Associates, Inc.