Benchmarking LLM Pipelines for Natural Language to Automated Planning Models

Marcus Tantakoun[1,2], **Christian Muise**[1,2], **Xiaodan Zhu**[1,3]
[1]Ingenuity Labs Research Institute, Queen's University
[2]School of Computing, Queen's University
[3]Department of Electrical and Computer Engineering, Queen's University
{20mt1, christian.muise, xiaodan.zhu}@queensu.ca

Abstract

Large Language Models (LLMs) excel in a wide variety of natural language processing (NLP) tasks. However, they struggle with long-horizon planning problems requiring structured reasoning, which is essential for many applications. This has led to increased interest in developing better methods within the Automated Planning (AP) and NLP communities. Amid this surge of interest, a growing number of frameworks and LLM-based methods are being proposed. Yet, it remains unclear what works effectively in practice. The evaluation of these methods is limited by the lack of prompt standardization, testing with the most recent state-of-the-art LLMs, and the absence of large-scale benchmarks. This paper makes contributions by bridging that gap, providing a systematic comparison and insights into natural-language-to-AP frameworks using unified datasets, evaluated through rigorous benchmarking on accuracy, scalability, and robustness, and serving as an investigation to uncover their limitations and strengths. We further introduce **GRAN2PLAN**, a novel LLM-based framework for constructing PDDL domains. Through this evaluation, we identify common pitfalls and clarify ambiguities in LLM-guided planning formalisms.

1 Introduction

In the past few years, Large Language Models (LLMs) have profoundly reshaped the foundations of AI research and applications (Plaat et al. 2025). Despite their strengths in System I tasks (Daniel 2017), the existing models struggle with long-horizon planning problems requiring structured reasoning—an inherently System II cognitive function (Bengio 2020), which is essential for developing reliable and responsible models for many applications. These challenges are evident in tasks that require long-term reasoning and consistency, where LLMs often produce incomplete or unreliable plans as task complexity increases (Valmeekam, Stechly, and Kambhampati 2024).

While end-to-end LLM-based planning lacks soundness guarantees and exhibits principled limitations (Valmeekam et al. 2024), a hybrid approach shows promise: LLMs can extract and refine planning model specifications from natural language, which Automated Planning (AP) systems can then use to generate executable plans. Leveraging powerful LLMs to assist in constructing planning models is a critical research direction, as verifiable planning modules remain essential structural backbones for ensuring reliability, robustness, and explainability for humans.

However, current methods—ranging from incremental construction to end-to-end synthesis—differ in datasets and input assumptions, leading to fragmented research. This highlights the need for cohesive benchmarking and standardization to ensure fair comparison and enable the integration of LLM-based planning into scalable, real-world AI systems. In this paper, we investigate state-of-the-art frameworks and studies under a unified standard, enabling an understanding and resolution of ambiguities that are of concern to the NLP and Automated Planning (AP) communities. By evaluating these techniques on a common benchmark, we contribute to revealing shared pitfalls and key trade-offs to guide the future of impactful LLM+AP research. Our results show that 1) DEEPSEEK-R1 and CLAUDE-SONNET-4 consistently achieve the strongest performance across pipelines; 2) our framework *GRAN2PLAN* performs competitively with the top systems; and 3) LLMs substantially improve in model quality generation when provided with more informative, structured error feedback.

Our contributions are threefold:

- To the best of our knowledge, this is the first comprehensive study and analysis of existing NL-to-PDDL pipelines across multiple state-of-the-art LLMs.
- Under a unified standard, this work is able to highlight strengths, limitations, and failure modes of current techniques. We also release our framework, dataset, and evaluation tools to support future studies and reproducibility.
- We introduce **GRAN2PLAN**, short for **GRAN**ular-to-**PLAN**, which is a novel LLM-driven framework for fine-grained, stepwise generation of PDDL domain models.

2 Related Work

LLMs as Planners: Recent work has explored the use of LLMs as reasoning agents for planning tasks. We refer to this direction as *LLMs as Planners*, where LLMs are employed either to analyze and critique action steps (Zhang et al. 2024) or to propose initial plans that are iteratively refined (Gundawar et al. 2024; Arora and Kambhampati 2023). However, beyond a certain threshold of problem complexity, performance consistently drops significantly (Shojaee et al. 2025), reflecting their limitations in

handling more challenging planning tasks. These limitations have pushed investigation to alternative approaches.

LLMs as Planning Formalizers: In contrast, our work explores the alternative *LLMs as Planning Formalizers* paradigm, where LLMs translate natural language (NL) descriptions into formal, machine-executable planning specifications (i.e., PDDL) that calculates plans using an external planner (see Figure 1). In light of recent success, researchers are exploring how to incorporate LLM-PDDL generation into their pipelines (Liu et al. 2023; Smirnov et al. 2024).

Despite growing progress, LLM-PDDL frameworks remain under-evaluated by standardized benchmarks. This gap is further complicated by LLMs' sensitivity to prompt design, dataset variation, and the lack of consensus on how to assess their quality (Katz et al. 2025; Behnke and Bercher 2024). Some studies measure success via end-to-end plan outcomes (Guan et al. 2023; Bohnet et al. 2024), but this obscures specific failure modes. Oswald et al. (2024) evaluate generated domain actions both syntactically and behaviorally. For syntactic evaluation, they introduce the Action Reconstruction Error (ARE) metric, which compares predicted action predicates against ground-truth. For behavior, they propose Heuristic Domain Equivalence (HDE), which assesses plan validity by testing generated domains against ground-truth planning instances, though exhaustive plan checks constrain benchmark scalability.

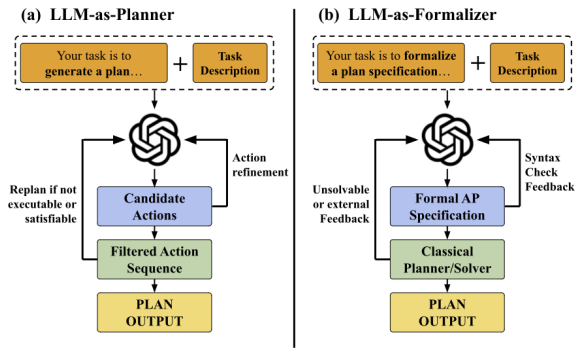

Figure 1: General planning methods with LLMs: (a) LLM-as-Planner: direct I/O planning; (b) LLM-as-Formalizer: specification generation for planners (e.g., PDDL).

3 Preliminaries

3.1 Automated Planning

Automated Planning (AP) is concerned with synthesizing action sequences that transform an initial state into a desired goal state while respecting environmental constraints. The category most well studied in this field is the *Classical Planning Problem* (Russell and Norvig 2020), which can be formally represented as a tuple:

$$\mathcal{M} = \langle \mathcal{S}, s_{\mathcal{I}}, s_{\mathcal{G}}, \mathcal{A}, \mathcal{T} \rangle$$

where $\mathcal{S}$ denotes a finite, discrete set of world states, with each state $s \in \mathcal{S}$ characterized by a fixed set of variable assignments. $s_{\mathcal{I}} \in \mathcal{S}$ represents the initial state of the system and $s_{\mathcal{G}} \subseteq \mathcal{S}$ defines the set of valid goal state conditions that must be held true. $\mathcal{A}$ is a collection of actions available to the planner. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function that determines the successor state $\mathcal{T}(s^i, a) = s^{i+1}$ when action $a \in \mathcal{A}$ is *applicable* in state s^i.

A solution to a planning problem $\mathcal{P}$ is a plan $\phi = \langle a_1, a_2, \ldots, a_n \rangle$ such that the preconditions of a_1 hold in the initial state $s_{\mathcal{I}}$. For each subsequent action a_k $(1 < k \leq n)$, its preconditions are satisfied in the state resulting from applying all previous actions. The final state after executing all actions in ϕ must satisfy the goal conditions $s_{\mathcal{G}}$.

3.2 PDDL

The Planning Domain Definition Language (PDDL) (McDermott et al. 1998) is among the most widely used methods for encoding Classical Planning problems. The PDDL representation consists of two key components:

1. **Domain File ($\mathbb{DF}$)** – Defines the universal aspects of a problem environment, including a fixed set of rules and constraints. In its simplest form, $\mathbb{DF}$ consists of: **Predicates** defining the state space $\mathcal{S}$, and set of **actions** $\mathcal{A}$. Each action $a \in \mathcal{A}$ is structured as: **Parameters** (a_{Par}) which specifies the types used in the action; **Preconditions** (a_{Pre}) are the conditions required for execution; and **Effects** (a_{Eff}) specifies the resulting state changes. Together, these define the transition function $\mathcal{T}$.
2. **Problem File ($\mathbb{PF}$)** – Grounds the domain with specific instances, including: A list of **objects** that ground the domain with the **initial state** $s_{\mathcal{I}}$ and **goal conditions** $s_{\mathcal{G}}$.

Using $\mathbb{DF}$ and $\mathbb{PF}$ as input to an external planner, domain-independent search algorithms are used to find a valid sequence of actions (i.e., a **plan**) that transitions the system from the initial states to states that satisfy goal conditions according to the dynamics defined by the domain.

4 Methodology

4.1 Problem Formulation

Our work focuses on the extraction of PDDL domains ($\mathbb{DF}$) from natural language using LLMs. Other components of the planning model, such as problem instance ($\mathbb{PF}$) generation, are considered out of scope. Formally, given a natural language description D_{NL}, an LLM is tasked to generate, at minimum, the corresponding PDDL domain model:

$$D_{\text{PDDL}} = \langle \mathcal{D}_R, \mathcal{D}_P, \mathcal{A} \rangle,$$

- $\mathcal{D}_R$: the set of :REQUIREMENTS declarations that specify the language features used in the domain.
- $\mathcal{D}_P$: the set of :PREDICATES encoding the logical relationships and properties used to describe world states.
- $\mathcal{A}$: the set of :ACTIONS, each specifying the action's parameter variables as well as its operators—preconditions and effects—defined using the available predicates.

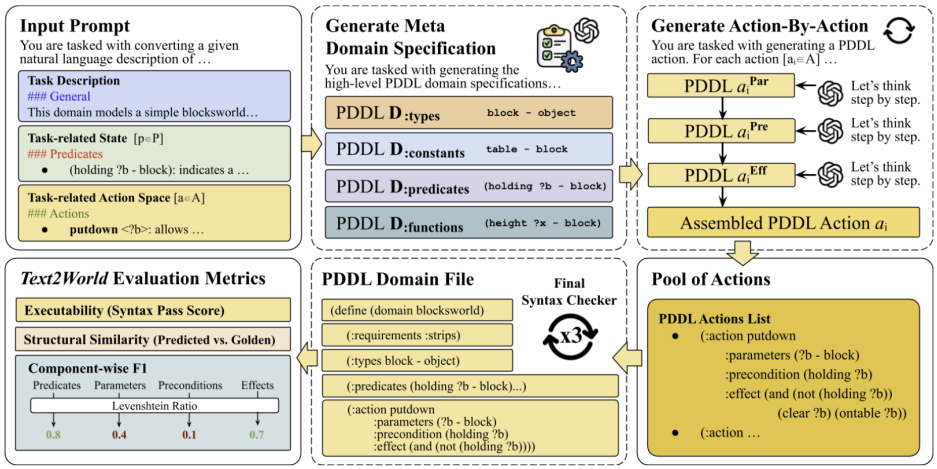

Figure 2: Overview of the *GRAN2PLAN* pipeline: LLM incrementally generates each action schema at a fine-grained level.

4.2 NL-PDDL Pipelines

A key contribution of our work is identifying the pipeline that best balances efficiency and PDDL quality. For reproducibility, we use **L2P** (Tantakoun, Muise, and Zhu 2025), an open-source Python tool that standardizes each pipeline from its original implementation:

- **TEXT2WORLD**: Hu et al. (2025) take a holistic approach, generating the full PDDL domain in a single pass. As this method tasks the LLM with producing a complete PDDL file directly, it *does not* leverage L2P's standardized pipeline implementation.
- **NL2PDDL**: Oswald et al. (2024) employ an action-by-action strategy in which the LLM incrementally constructs each action schema one at a time, without conditioning on previously generated actions.
- **GRAN2PLAN**: Our framework (Figure 2) adopts a similar strategy as NL2PDDL, but decomposes each action into sequential subtasks (i.e., parameters, preconditions, and effects) where each subtask is conditioned on the previous. This aims to reduce LLM cognitive load by restricting generation to context-specific scopes.

To ensure a fair comparison to TEXT2WORLD, we additionally tasked the LLM to generate the meta-domain specifications (e.g., :TYPES)—where necessary—for both NL2PDDL and GRAN2PLAN. Further information on the specific pipeline algorithms can be found in Appendix C.

4.3 Prompting

Each pipeline is prompted with the general description of a domain, denoted as d, along with a predefined set of predicate descriptions N_P and action descriptions N_A corresponding to the domain. Hu et al. (2025) demonstrates the effectiveness of incorporating few-shot chain-of-thought (CoT) prompting in enhancing domain modelling. Therefore, we use a two-shot CoT strategy across all pipelines, with in-context examples from the BLOCKSWORLD and GRIPPERS domains. Explicit examples of our prompt setup can be found in Appendix E.

4.4 Feedback Correction

Prior work has highlighted the importance of feedback correction driven by external PDDL syntax validators (Guan et al. 2023). Instead of the previously used TARSKI[1] checker leveraged by Hu et al. (2025), we opted for the PDDL[2] library due to its active maintenance, richer semantic messages, and greater flexibility in code structure manipulation. For structural issues (e.g., missing brackets), checkers often return only a line number without a clear mapping to the faulty code. To mitigate this, we modified our correction prompt to explicitly number each line of PDDL code as an effective signal to the root of errors. All pipelines enforce a final maximum correction loop of three iterations (EC_3); failure to produce executable output is deemed unsuccessful. Figure 3 shows a side-by-side example comparison of the error messages produced by PDDL and TARSKI.

PDDL CHECKER

```
PDDLValidationError: Type mismatch in argument 1 of
predicate 'clear' found in :precondition of action
'Lift': got '?y' assigned type 'crate', expected:
    * 'surface' (or its subtype(s))

Parsed error line: (clear ?y)
```

TARSKI CHECKER

```
SortMismatch: Sort mismatch on element ?y. Expected
was "Sort(surface)", element has sort "Sort(crate)"
```

Figure 3: Example of the same error comparing the PDDL and Tarski validator output message.

4.5 Evaluated Models

Our study focuses on frontier models, leaving the evaluation of smaller open-source alternatives for future work. We

[1]https://github.com/aig-upf/tarski

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

evaluated a range of 5 LLMs and 4 large reasoning models (LRMs) from 5 families: O4-MINI, O3-MINI (OpenAI 2025), GPT-4.1, GPT-4.1-MINI (OpenAI 2023); GEMINI-2.5-FLASH [3] (Google 2025); DEEPSEEK-R1, DEEPSEEK-V3 (DeepSeek-AI 2024); CLAUDE-SONNET-4 (Anthropic 2025); and MISTRAL-MEDIUM-LATEST (Mistral 2025). All models were accessed through their respective APIs. For consistency and reproducibility, we set the decoding temperature to $t = 0.0$. For the reasoning models, we set the reasoning effort to MEDIUM (thinking token budget: 8192).

5 Experiment Settings

5.1 Benchmark Dataset and Evaluation Metrics

We conduct our evaluations using the *TEXT2WORLD* benchmark (Hu et al. 2025), which comprises approximately 100 PDDL domains. Each domain is paired with manually annotated NL descriptions that exhibit low LLM training contamination rates using n-gram matching (Touvron et al. 2023). To enable a fine-grained evaluation, we also follow their multi-metric gating schema:

1. ***Executability* (EXEC.):** The primary gating metric. Evaluates whether the specification is syntactically valid (EXEC=1) or not (EXEC=0). We opted for the PDDL parser for checks. Only domains with EXEC=1 proceed to component-wise F1 scoring.
2. ***Component-wise F1*:** Applied exclusively to executable outputs, this metric reports macro-averaged F1 scores across predicates ($\mathbf{F1_{PRED}}$) and, for each action, evaluates accuracy over parameters ($\mathbf{F1_{PARAM}}$), preconditions ($\mathbf{F1_{PREC}}$), and effects ($\mathbf{F1_{EFF}}$).
3. ***Structural Similarity* (SIM.):** Applied to all domains regardless of executability. Captures textual similarity between predicted and ground truth PDDL via normalized Levenshtein ratio scoring, serving as a last-resort signal for domains that fail executability.

Detailed metric formulas can be found in Appendix B.

6 Experimental Results

6.1 Key Findings and Insights

Based on Table 1, we identified several insights. Firstly, **TEXT2WORLD performs the worst among the NL-PDDL pipelines**. As shown in Figure 4, in the zero-shot error correction setting (EC_0), *TEXT2WORLD* performs poorly across all metrics, with low executability (32.4%) and overall F1-scoring. In contrast, both *GRAN2PLAN* (EXEC. 61.1%) and *NL2PDDL* (EXEC. 64.9%) achieve significantly better results. This suggests that incremental generation helps mitigate errors, potentially by reducing the LLMs' cognitive tasks during inference. Even with error correction refinement, *TEXT2WORLD* continues to lag behind, reinforcing this observation. Applying the error correction setting (EC_3), all pipelines show that LLMs struggle with accurately modelling action operators—especially preconditions, exhibiting higher failure rates than effects. This likely

[3] GEMINI-2.5-FLASH reasoning feature disabled due to token budget overruns during internal reasoning, which led to incomplete or missing outputs in preliminary experiments.

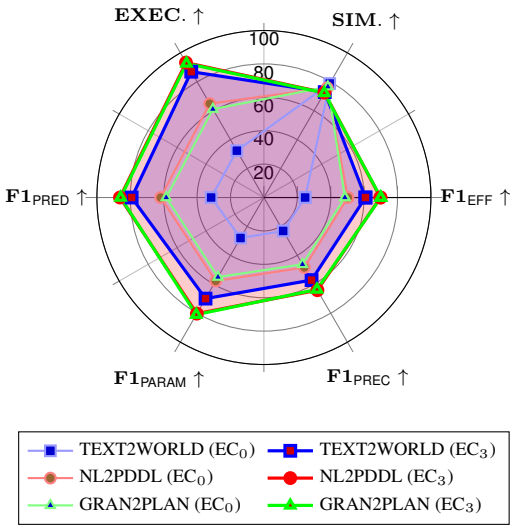

Figure 4: Aggregate comparison of pipeline performance based on average model scores across key accuracy metrics.

stems from natural language tending to state effects explicitly, while preconditions often involve implicit constraints.

More informative error messages significantly improve code executability. Replacing the TARSKI validator with the PDDL library—paired with explicit line-level error reporting—led to markedly better error recovery. In our updated evaluation, *TEXT2WORLD* paired with O3-MINI and DEEPSEEK-R1 achieved improvements of 91.0% and 93.0% (EC_3), respectively, surpassing the original results of 84.2% and 89.1% reported by Hu et al. (2025). Despite beginning with lower zero-shot scores, the enhanced error feedback produced substantial gains in executability, indicating that our detailed messages provide valuable context for LLMs to generate executable models. Surprisingly, textual similarity scoring drops after corrections despite an increased number of valid domains.

Smaller models see substantial improvements when using multi-step generation. Our results show that DEEPSEEK-R1 and CLAUDE-SONNET-4 are the best-performing models across all pipelines. However, incremental strategies allow *smaller* models to better compete with larger reasoning models. For instance, in the zero-shot correction setting, GPT-4.1-MINI achieves only 24% executability when generating full domains, but reaches 62% and 67% on *GRAN2PLAN* and *NL2PDDL*, respectively.

State-of-the-art LLMs struggle with generating advanced action schemas. While LLM-based pipelines can reliably generate executable domains, we found that higher-order constructs[4] (e.g., ADL features such as FORALL and WHEN) were only generated when explicitly included in the prompt. This suggests that (i) prompts must be carefully engineered to elicit advanced PDDL features, or (ii) further training on complex planning data is needed to enable LLMs to model such constructs autonomously.

[4] Advanced PDDL details available in supplementary material.

GRAN2PLAN

Family	Version	EXEC. ↑		SIM. ↑		$F1_{PRED}$ ↑		$F1_{PARAM}$ ↑		$F1_{PREC}$ ↑		$F1_{EFF}$ ↑	
		EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3
OpenAI	O4-MINI	52.0	93.0	77.4	72.2	50.9	84.2	50.0	77.2	41.8	61.9	43.4	65.5
	O3-MINI	73.0	92.0	75.1	70.7	70.1	87.0	67.7	84.5	54.8	63.8	57.9	70.7
	GPT-4.1	65.0	88.0	76.3	72.1	62.7	82.5	59.0	77.5	50.9	62.9	52.0	67.3
	GPT-4.1-MINI	62.0	94.0	76.4	72.4	59.7	86.7	55.0	79.6	45.0	59.7	48.4	68.4
DeepSeek	DEEPSEEK-R1	44.0	97.0	80.1	73.6	42.6	89.7	40.0	87.8	33.4	70.5	36.3	77.3
	DEEPSEEK-V3	44.0	86.0	79.0	73.7	41.8	79.9	38.0	76.1	33.2	59.4	33.6	65.1
Gemini	GEMINI-2.5-FLASH	70.0	95.0	75.0	70.8	67.3	88.3	64.0	83.5	51.5	63.4	57.0	74.5
Anthropic	CLAUDE-SONNET-4	73.0	97.0	74.7	72.5	67.1	88.2	63.0	84.6	54.7	67.6	57.3	75.4
Mistral	MISTRAL-MEDIUM	67.0	92.0	75.2	72.4	64.5	85.6	59.0	78.4	50.4	63.2	50.5	66.2
	Average	61.1	92.7	76.6	72.3	58.5	85.8	55.0	81.0	46.2	63.6	48.5	70.0

NL2PDDL

Family	Version	EXEC. ↑		SIM. ↑		$F1_{PRED}$ ↑		$F1_{PARAM}$ ↑		$F1_{PREC}$ ↑		$F1_{EFF}$ ↑	
		EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3
OpenAI	O4-MINI	63.0	91.0	76.0	72.8	59.7	83.7	55.2	76.3	46.1	60.9	47.8	67.1
	O3-MINI	71.0	95.0	75.1	72.5	67.9	88.4	62.1	81.7	51.7	65.5	54.5	71.1
	GPT-4.1	63.0	94.0	75.6	71.1	58.7	87.1	56.8	81.3	48.1	64.5	49.0	70.7
	GPT-4.1-MINI	67.0	93.0	76.3	71.7	64.9	85.5	61.4	80.0	51.5	61.7	53.6	69.2
DeepSeek	DEEPSEEK-R1	52.0	96.0	78.2	73.0	50.4	88.2	47.9	82.3	39.0	65.8	43.0	73.2
	DEEPSEEK-V3	43.0	88.0	78.4	73.7	42.8	82.3	39.0	75.4	33.9	61.0	34.0	64.1
Gemini	GEMINI-2.5-FLASH	76.0	95.0	74.1	72.5	71.3	87.4	67.5	84.1	55.6	65.9	59.4	74.2
Anthropic	CLAUDE-SONNET-4	82.0	99.0	74.4	72.5	77.1	91.4	71.3	85.9	59.7	69.4	62.1	75.8
Mistral	MISTRAL-MEDIUM	67.0	89.0	75.0	72.5	63.5	81.7	58.0	75.8	49.7	62.0	48.9	63.1
	Average	64.9	93.3	75.9	72.5	61.8	86.2	57.7	80.3	48.4	64.1	50.3	69.8

TEXT2WORLD

Family	Version	EXEC. ↑		SIM. ↑		$F1_{PRED}$ ↑		$F1_{PARAM}$ ↑		$F1_{PREC}$ ↑		$F1_{EFF}$ ↑	
		EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3	EC_0	EC_3
OpenAI	O4-MINI	55.0	84.0	65.5	65.8	52.1	70.6	39.6	51.2	33.5	42.3	36.4	45.3
	O3-MINI	34.0	91.0	69.5	67.3	33.0	76.6	31.5	64.1	26.6	52.5	28.2	55.4
	GPT-4.1	36.0	83.0	81.5	74.9	36.0	78.9	31.7	69.8	26.6	56.0	27.7	60.7
	GPT-4.1-MINI	24.0	79.0	82.1	74.2	24.0	74.8	20.2	61.4	17.1	49.2	17.5	51.7
DeepSeek	DEEPSEEK-R1	42.0	93.0	79.4	74.5	39.9	84.8	36.4	77.1	29.3	62.0	32.8	68.4
	DEEPSEEK-V3	13.0	84.0	84.0	75.6	13.0	76.8	11.0	69.6	8.5	56.1	8.4	61.0
Gemini	GEMINI-2.5-FLASH	10.0	80.0	84.2	73.7	10.0	73.3	10.0	72.3	7.5	59.8	8.9	62.9
Anthropic	CLAUDE-SONNET-4	43.0	95.0	81.5	75.6	43.0	90.4	39.0	83.2	31.5	70.1	35.5	73.6
Mistral	MISTRAL-MEDIUM	35.0	94.0	81.3	74.4	33.8	86.8	32.8	80.4	27.5	65.8	27.0	68.5
	Average	32.4	87.0	78.8	72.9	31.6	79.2	28.0	69.9	23.1	57.1	24.7	60.8

Table 1: LLM performance comparison across different pipelines. EC_k denotes a setting permitting K correction steps by the model. (EC_0: zero-shot w/o correction, EC_3: w/ 3 correction steps). Underlined values indicate the best performance within each metric category. Grey-highlighted values indicate the best overall across all pipelines.

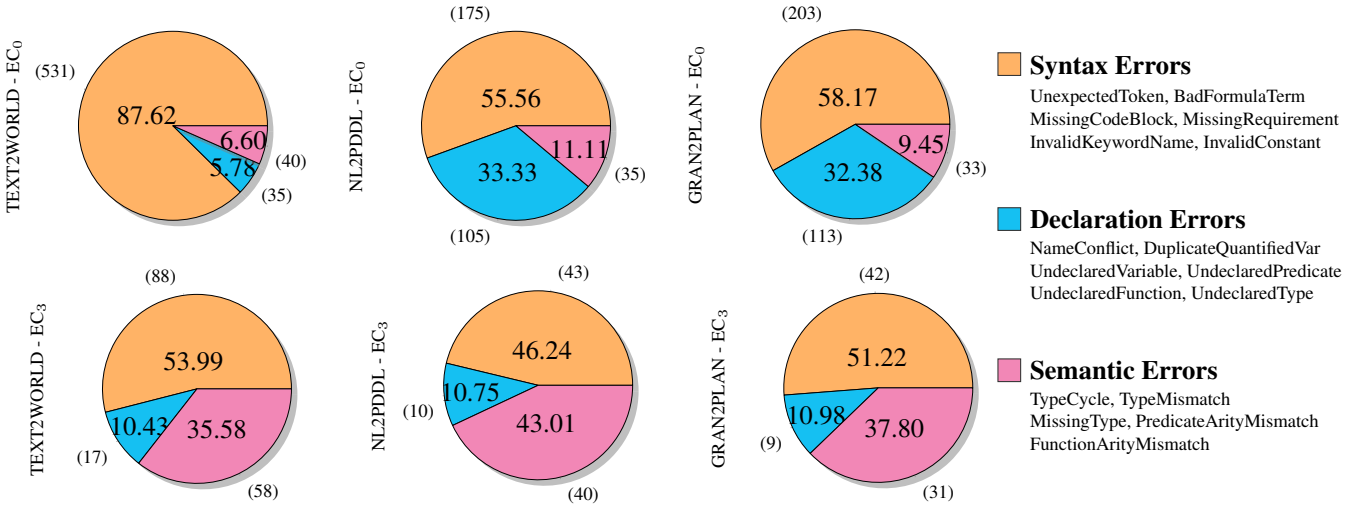

Figure 5: Pipeline error comparison between zero-shot and last error correction. *Note*: Numbers outside the plot show total errors across all LLMs.

6.2 Analysis

To reveal where LLMs fail in modelling planning domains, we categorize PDDL-related errors into the following failure modes (refer to Table 4 for further details):

- ***(i) Syntax errors***: parsing errors such as malformed formulas, unexpected tokens, or missing code blocks.
- ***(ii) Declaration errors***: undefined or improperly typed elements found outside the scope of defined constructs.
- ***(iii) Semantic errors***: violations of planning-specific logic, such as mismatched argument types for actions.

Figure 5 reveals three key findings: **(i) Single-shot domain generation results in frequent syntax errors**. *TEXT2WORLD* exhibited significantly more syntax errors—particularly frequent omission of necessary :REQUIREMENTS—per domain compared to the other pipelines. Because error messages reveal only one issue at a time, the fixed number of correction loops often fails to resolve all errors. Even in a scenario where we set a higher correction limit, our results show that repeated correction loops degrade overall PDDL quality. **(ii) LLMs recover well from syntax and declaration errors, but semantic issues persist**. Even after error correction is enabled, LLMs still heavily suffer from repairing semantic errors. In particular, models often fail to resolve type hierarchy cycle violations and mismatched types in predicate and function usage. **(iii) LLMs interpret natural language too literally**, often introducing name conflicts despite clear, unique naming in the description. Although absent in some ground-truth domains, LLMs frequently attempt to incorporate :TYPES definitions—reflecting a learned bias that treats them as necessary PDDL components. As a result, models often invent new terms to satisfy type constraints rather than removing them in the case of name conflicts.

Table 2: Categorization of PDDL Errors with Descriptions

	Error Type	Description
Syntax Errors	UNEXPECTEDTOKEN	Unexpected token that breaks parser
	BADFORMULATERM	Malformed term in atomic formula
	MISSINGCODEBLOCK	No PDDL/Lisp code block was found
	MISSINGREQUIREMENT	Feature not declared for formula usage
	INVALIDKEYWORDNAME	Invalid use keyword OBJECT as a name
	INVALIDCONSTANT	Malformed or invalid constant term
Declaration Errors	NAMECONFLICT	Two or more identifiers conflict
	DUPLICATEQUANTIFIEDVAR	Duplicate quantified var. in same scope
	UNDECLAREDVARIABLE	Variable used w/o declaration in scope
	UNDECLAREDPREDICATE	Predicate used but not defined in domain
	UNDECLAREDFUNCTION	Function used but not defined in domain
	UNDECLAREDTYPE	Type used but not defined in domain
Semantic Errors	TYPECYCLE	Cycle detected in type hierarchy
	TYPEMISMATCH	Arg. has incorrect/incompatible type
	MISSINGTYPE	Missing type for one or more arguments
	PREDICATEARITYMISMATCH	Predicate contains incorrect # of args.
	FUNCTIONARITYMISMATCH	Function contains incorrect # of args.

6.3 Exploration

Per-step Syntax Correction We examined whether introducing a step-wise syntax-checking feedback loop—applied after each generated action in *NL2PDDL* and *GRAN2PLAN*—would enhance the quality of the PDDL domain models. The goal is to catch errors early to prevent cascading syntax violations that could exceed the correction limit. Figure 6 and 7 presents the results of this comparison, highlighting the impact of step-wise validation on final generation accuracy.

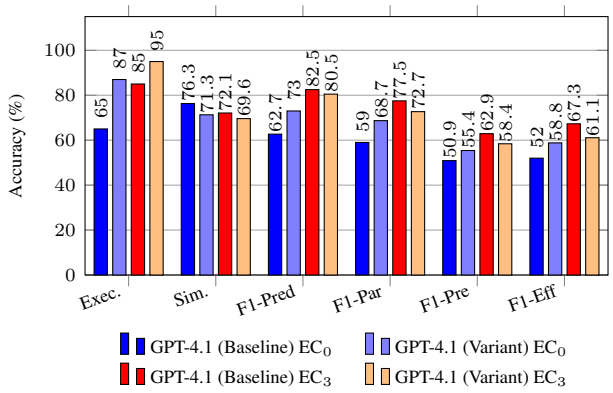

Figure 6: GRAN2PLAN pipeline performance b/w baseline error correction and per-step syntax checking variant

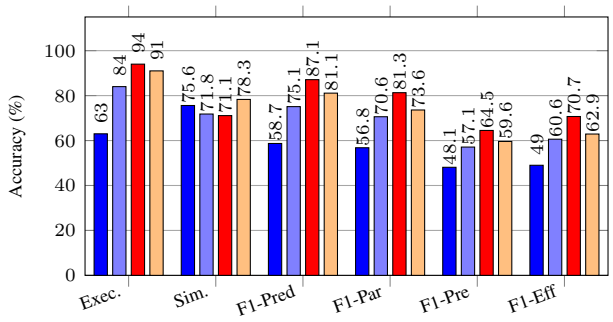

Figure 7: Comparison of NL2PDDL pipeline performance

Based on our results, **fine-grained syntax checkers significantly improved executability** upon arriving at final syntax checks (EC_0). However, at the expense of per-step error correction, they often produced incomplete or oversimplified actions by forcing syntactic constraints over original semantic quality. This ultimately led to lower F1-predicate and F1-action scores. We found that just using the final domain-level syntax check correction was sufficient for error recovery.

LLM-Feedback To further improve performance, we explore *LLM-Enhanced Feedback*, inspired by *NL2PLAN* (Gestrin, Kuhlmann, and Seipp 2024): an additional LLM evaluates each generated action using a checklist-guided reasoning process, either validating or suggesting revisions to enhance semantic accuracy. Figure 9 is the base prompt we used to elicit feedback. We revisit the *NL2PDDL* framework with few-shot CoT prompting, using GEMINI-2.5-FLASH for PDDL generation and feedback, and additionally evaluate GEMINI-2.5-PRO as an alternative feedback model. Generation is deterministic ($t = 0.0$), while feedback is tested at $t = 0.0$ and $t = 1.0$ to evaluate whether increased temperature settings encourage more diverse outputs.

As shown in Figure 8, **relying solely on LLMs' internal knowledge to revise PDDL domains often degrades their quality**. Specifically, LLMs frequently suggested removing

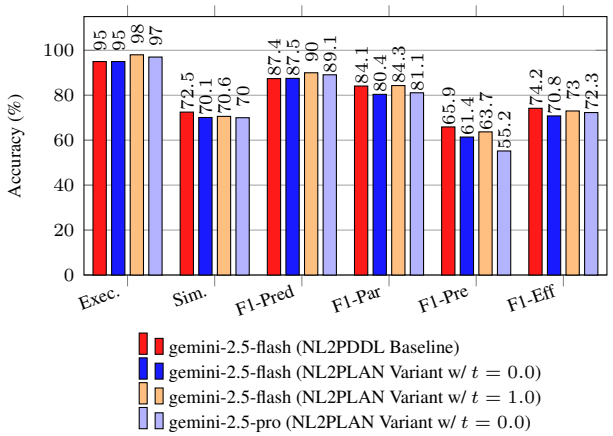

Figure 8: Comparison of pipeline performance with LLM-enhanced feedback.

necessary predicates, likely overfitting to surface-level descriptions, despite explicitly being instructed to infer implicit constraints. Using the Levenshtein ratio, we measured the average textual difference between the original and revised actions: GEMINI-2.5-PRO ($t = 0$) suggested edits for 81% of actions with a resulting 13% average change, while GEMINI-2.5-FLASH ($t = 0$) and its higher-temperature variant ($t = 1.0$) recommended changes about 50% of the time with 9–10% differences. Despite suggesting the most revisions, GEMINI-2.5-PRO performed the worst. These findings, consistent with Gragera and Pozanco (2023), where they suggest that current LLMs are poorly equipped to repair planning models in isolation. Incorporating external knowledge via domain documentation API's or retrieval-augmented generation (RAG) appears necessary to extract meaningful domain insight.

```
Review the following PDDL action and identify issues
in its structure and semantics. Go beyond surface-
level checks | infer what should be present based on
the intended meaning of the action. Provide detailed
corrections if needed.

Start your response with a "## Response" header follo-
wed by your step-by-step thinking using the checklist:
1. Are any necessary precondition checks missing?
2. Are any unnecessary preconditions checked?
3. Are any necessary effects missing?
4. Are any unnecessary effects included?
5. Can the used predicates be replaced by other
    defined predicates? For example, [...]
6. Should any predicate be used in a symmetrical
    manner? For example, [...]

If the action is well defined, respond with
"no feedback" under header '### JUDGEMENT'.
```

Figure 9: Prompt used in NL2PLAN to elicit corrective feedback from the LLM.

Table 3: API usage details for LLM rollouts over all domains

	Models	API $ (USD)	# Calls	Avg. IP	Avg. OT
GRAN2PLAN	o4-mini	$12.72	1495	2897.14	1209.24
	o3-mini	$12.50	1478	2923.28	1191.38
	gpt-4.1	$13.76	1495	2926.88	419.03
	gpt-4.1-mini	$2.88	1675	2806.22	373.74
	deepseek-r1	$2.33	1506	2868.04	2631.27
	deepseek-v3	$0.46	1521	2883.53	360.94
	gemini-2.5-flash	$3.22	1488	3095.89	492.97
	claude-sonnet-4	$39.78	1459	3280.31	1161.74
	mistral-medium	$2.78	1495	2979.60	334.01
NL2PDDL	o4-mini	$7.53	603	3752.62	1898.07
	o3-mini	$7.82	591	3805.22	2055.51
	gpt-4.1	$8.14	609	3805.47	719.07
	gpt-4.1-mini	$1.60	599	3809.34	715.34
	deepseek-r1	$1.34	603	3731.04	3796.82
	deepseek-v3	$0.33	643	3648.43	706.84
	gemini-2.5-flash	$2.36	586	4181.46	1111.82
	claude-sonnet-4	$22.48	566	4375.10	1772.73
	mistral-medium	$1.76	613	3871.45	660.38
TEXT2WORLD	o4-mini	$2.42	194	2371.56	2238.34
	o3-mini	$2.68	218	2231.20	2238.08
	gpt-4.1	$2.56	216	2550.69	846.42
	gpt-4.1-mini	$0.61	237	2470.79	995.20
	deepseek-r1	$0.54	189	2630.14	4995.81
	deepseek-v3	$0.13	264	2388.94	742.21
	gemini-2.5-flash	$0.98	264	2605.30	1167.29
	claude-sonnet-4	$8.12	174	3066.65	2498.44
	mistral-medium	$0.44	190	2497.92	670.47

Note. IP=Input Tokens, OT=Output Tokens. API calls varies on final error correction occurrences.

7 Discussion

7.1 Comparative Insights Across Pipelines

Findings on our framework: Table 1 shows that while *GRAN2PLAN* performed better than *TEXT2WORLD*, it did not yield significant improvements compared to the more holistic, action-level approach of *NL2PDDL*. While we hypothesized that finer granularity would reduce cognitive load and increase action-level accuracy, the results suggest that such excessive decomposition may not be necessary for fully automated pipelines. That being said, in human-in-the-loop settings, users gain a substantial advantage from *GRAN2PLAN*'s structured step-by-step process, which delivers the same model quality as state-of-the-art systems.

Striking the Right Balance: Finding the optimal generation granularity requires balancing efficiency with output quality. Generating the entire domain in a single pass is the most efficient, but consistently produces the lowest executability and accuracy F1-scores. In contrast, incremental generation yields higher-quality domains. However, Table 3 shows that *GRAN2PLAN*'s finer-grained method demands $\sim$2.5x more inference calls than *NL2PDDL* with subpar gains; and *NL2PDDL* yields $\sim$2.7x more inference calls than *TEXT2WORLD*. These findings suggest that generating PDDL domains on an "action-by-action" basis—as demonstrated by *NL2PDDL*—and equipping it with a syntax validator at the end for error correction, strikes the best balance between accuracy and efficiency. Additionally, our results show that smaller models have the potential to perform competitively using such pipelines. Therefore, users should consider using lightweight, open-source alternatives within our framework (or NL2PDDL) for practicality.

Pipeline Transparency: Considering broader applications, *TEXT2WORLD* lacks fine-grained traceability and transparency in the failure modes linked to LLM decision-making. This becomes particularly critical when considering not just syntax violations, but the semantic inconsistencies that can directly lead to plan failure. For interactive human-AI systems, *GRAN2PLAN*'s subtask action decomposition offers greater directions for interpretability. Furthermore, generating intermediate outputs paired with explanations enhances human understanding and debugging, which are crucial for the design of real-world planning models.

7.2 Limitations and Future Work

Our work presents three key limitations that inform future research directions. First, limited context is provided to the LLM during error correction such that only the faulty PDDL and its associated error message are supplied. This lack of context leads to narrow syntax fixes, often at the expense of semantic quality, causing the corrected domain to drift from the original intent. Future work should reintroduce the original description during correction to evaluate whether syntax can be resolved while preserving intent. Second, errors in the PDDL :REQUIREMENTS field are common, yet unlike other syntax issues, they can often be computed automatically from the rest of the domain. Instead, *TEXT2WORLD* relied on LLMs to generate these specifications. Incorporating automatic requirement inference would reduce trivial errors. Finally, the Levenshtein ratio is a poor indicator of domain quality in this setting. For instance, a domain can score high in structural similarity yet be semantically incorrect or entirely unsolvable. Future work should adopt more rigorous, semantic evaluation measures such as the Heuristic Domain Equivalence (HDE) metric proposed by Oswald et al. (2024) that better reflect true planning correctness.

8 Conclusion

We provide a comprehensive study benchmarking state-of-the-art natural language to planning language specification (PDDL) pipelines, featuring an extensive assessment across 9 state-of-the-art LLMs spanning 5 model families. We contribute *GRAN2PLAN*, a novel LLM-based pipeline for generating PDDL domains and include it in our evaluation. Our results reveal that, while holistic generation offers the most efficient approach, alternative pipelines show stronger potential for producing semantically richer and accurate domains. In addition, we highlight the critical importance of robust syntax validation using external checkers—and more importantly, more informative error correction feedback messages. However, excessive correction often yields diminishing returns in quality. All code to our pipelines and resources will be publicly released, supporting standardized evaluation and accelerating progress in LLM-driven planning specification research.

References

Anthropic. 2025. Introducing Claude 4.

Arora, D.; and Kambhampati, S. 2023. Learning and Leveraging Verifiers to Improve Planning Capabilities of Pretrained Language Models. *CoRR*, abs/2305.17077.

Behnke, G.; and Bercher, P. 2024. Envisioning a Domain Learning Track for the IPC.

Bengio, Y. 2020. Deep Learning for System 2 Processing. *AAAI 2020*.

Bohnet, B.; Nova, A.; Parisi, A. T.; Swersky, K.; Goshvadi, K.; Dai, H.; Schuurmans, D.; Fiedel, N.; and Sedghi, H. 2024. Exploring and Benchmarking the Planning Capabilities of Large Language Models. *CoRR*, abs/2406.13094.

Daniel, K. 2017. *Thinking, fast and slow*. Macmillan.

DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *CoRR*, abs/2412.19437.

Gestrin, E.; Kuhlmann, M.; and Seipp, J. 2024. NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions. *CoRR*, abs/2405.04215.

Google. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities.

Gragera, A.; and Pozanco, A. 2023. Exploring the Limitations of using Large Language Models to Fix Planning Tasks. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Gundawar, A.; Verma, M.; Guan, L.; Valmeekam, K.; Bhambri, S.; and Kambhampati, S. 2024. Robust Planning with LLM-Modulo Framework: Case Study in Travel Planning. *CoRR*, abs/2405.20625.

Hu, M.; Chen, T.; Zou, Y.; Lei, Y.; Chen, Q.; Li, M.; Mu, Y.; Zhang, H.; Shao, W.; and Luo, P. 2025. Text2World: Benchmarking Large Language Models for Symbolic World Model Generation.

Katz, M.; Kokel, H.; Muise, C.; Sohrabi, S.; and Sreedharan, S. 2025. Make Planning Research Rigorous Again! *CoRR*, abs/2505.21674.

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

McDermott, D.; M. Ghallab, A. H.; Knoblock, C.; A. Ram, M. V.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Mistral. 2025. Medium is the new large.

OpenAI. 2023. GPT-4 Technical Report. *CoRR*, abs/2303.08774.

OpenAI. 2025. OpenAI o3 and o4-mini System Card.

Oswald, J. T.; Srinivas, K.; Kokel, H.; Lee, J.; Katz, M.; and Sohrabi, S. 2024. Large Language Models as Planning Domain Generators. In Bernardini, S.; and Muise, C., eds., *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2024, Banff, Alberta, Canada, June 1-6, 2024*, 423–431. AAAI Press.

Plaat, A.; van Duijn, M. J.; van Stein, N.; Preuss, M.; van der Putten, P.; and Batenburg, K. J. 2025. Agentic Large Language Models, a survey. *CoRR*, abs/2503.23037.

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson. ISBN 9780134610993.

Shojaee, P.; Mirzadeh, I.; Alizadeh, K.; Horton, M.; Bengio, S.; and Farajtabar, M. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941*.

Smirnov, P.; Joublin, F.; Ceravola, A.; and Gienger, M. 2024. Generating consistent PDDL domains with Large Language Models. *CoRR*, abs/2404.07751.

Tantakoun, M.; Muise, C.; and Zhu, X. 2025. LLMs as Planning Formalizers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, 25167–25188. Association for Computational Linguistics.

Touvron, H.; et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR*, abs/2307.09288.

Valmeekam, K.; Stechly, K.; Gundawar, A.; and Kambhampati, S. 2024. Planning in Strawberry Fields: Evaluating and Improving the Planning and Scheduling Capabilities of LRM o1. *arXiv preprint arXiv:2410.02162*.

Valmeekam, K.; Stechly, K.; and Kambhampati, S. 2024. LLMs Still Can't Plan; Can LRMs? A Preliminary Evaluation of OpenAI's o1 on PlanBench. *arXiv preprint arXiv:2409.13373*.

Zhang, X.; Altaweel, Z.; Hayamizu, Y.; Ding, Y.; Amiri, S.; Yang, H.; Kaminski, A.; Esselink, C.; and Zhang, S. 2024. DKPROMPT: Domain Knowledge Prompting Vision-Language Models for Open-World Planning. *CoRR*, abs/2406.17659.

A Supplementary Information on PDDL

PDDL's standardization has facilitated benchmarking, data sharing, and tool development for validation and refinement. Its clear, declarative syntax and flexibility align well with LLMs' capabilities, particularly given the likelihood that such code appears in their training data.

A.1 PDDL Example Usage

Automated Planning, a distinct subfield of AI, often poses difficulties for newcomers given its roots in an earlier era than modern machine learning. This concept of action sequence decision-making is formally specified by symbolic models in specification languages such as the Planning Domain Definition Language (PDDL) to encode classical planning problems. To clarify these ideas, we use the classic Blocksworld example, where the task involves rearranging blocks into a target configuration through actions such as picking up, unstacking, and placing blocks. These actions must adhere to rules like moving only one block at a time and preserving the order of stacked blocks. The following example shows the Blocksworld PDDL domain file $\mathbb{DF}$:

```
(define (domain blocksworld)
   (:requirements :strips)
   (:predicates (clear ?x) (on-table ?x) (arm-empty)
        (holding ?x) (on ?x ?y))
   (:action pickup
      :parameters (?ob)
      :precondition (and (clear ?ob) (on-table ?ob)
           (arm-empty))
      :effect (and (holding ?ob) (not (clear ?ob))
           (not (on-table ?ob)) (not (arm-empty)))
   ))
```

Predicates define properties that can either be true or false, such as (ON ?X ?Y) for block ?X on ?Y, (ONTABLE ?X) for ?X on the table, (CLEAR ?X) for ?X having nothing on top, and (HOLDING ?X) for the agent holding ?X. **Actions** define possible state changes. For example, (PICK-UP) contains the *parameter(s)* block ?OB, *preconditions* requiring (CLEAR ?OB) and (ONTABLE ?OB), and *effects* updating the state to reflect the agent holding ?OB. The following is the PDDL problem file $\mathbb{PF}$:

```
(define (problem blocksworld-problem)
  (:domain blocksworld)
  (:objects A B C) ; Blocks
  (:init (ontable A) (ontable B) (on C A) (clear B)
       (clear C)) ; Initial state
  (:goal (and (on A B) (on B C)))) ; Goal state
```

Here, **Objects** represent the entities involved, such as blocks A, B, and C. The **initial state** defines the start configuration, where blocks A and B are on the table, block C is on A, and both B and C are have no blocks on top of them. The **goal state** specifies the desired end state, where block A is stacked on B, and block B is stacked on C. Given the previous PDDL domain and problem, an external classical planner might generate the following plan:

```
Unstack C from A
Put C on table
Pick up A
Stack A on B
Pick up B
Stack B on C
```

B TEXT2WORLD Detailed Metrics

We evaluate LLM-generated PDDL domains using the *TEXT2WORLD* multi-metric system: (i) PDDL file executability; (ii) domain text similarity with ground truth; (iii) F1-metrics with ground truth.

Levenshtein Ratio . To assess the overall structural similarity between a generated domain and its reference counterpart, we compute the Levenshtein ratio:

$$\text{Levenshtein Ratio} = 1 - \frac{d(a,b)}{|a| + |b|} \tag{1}$$

where $d(a,b)$ is the Levenshtein distance between strings a and b, and $|a|$, $|b|$ are their respective lengths. This normalized measure ranges from 0 to 1, where higher values indicate greater string similarity.

If the file is executable, that is, does not contain any syntax errors, the following metrics are used to conduct further evaluation on the semantics of the domain:

Predicate-Level F1 Score . To evaluate the correctness of the generated predicate definitions, we compute an F1 score between the predicted predicate set P and the reference set R:

$$F1_{\text{pred}} = \begin{cases} 1 & \text{if } |P| = 0 \text{ and } |R| = 0 \\ 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} & \text{otherwise} \end{cases} \tag{2}$$

Action Component-Level F1 Score . To capture the accuracy of individual components within each action schema—namely, parameters, preconditions, and effects—we compute component-specific F1 scores. For each component $c \in \{\text{params}, \text{preconds}, \text{effect}\}$, the average F1 score over all actions is:

$$F1_c = \frac{1}{N} \sum_{i=1}^{N} F1_{c,i} \tag{3}$$

where N is the number of actions, and $F1_{c,i}$ denotes the F1 score for component c in action i, computed as:

$$F1_{c,i} = 2 \cdot \frac{\text{precision}_{c,i} \cdot \text{recall}_{c,i}}{\text{precision}_{c,i} + \text{recall}_{c,i}} \tag{4}$$

with precision and recall defined over the predicted and reference component sets $P_{c,i}$ and $R_{c,i}$, respectively:

$$\text{precision}_{c,i} = \frac{|P_{c,i} \cap R_{c,i}|}{|P_{c,i}|}, \quad \text{recall}_{c,i} = \frac{|P_{c,i} \cap R_{c,i}|}{|R_{c,i}|}$$

C Details of Pipeline Variants

We present details of each pipeline's PDDL domain generation methodology. We opted to run the syntax validation loop only once at the end of the pipeline, as implemented in TEXT2WORLD. The following is TEXT2WORLD's single-pass generation of the domain via LLM.

Algorithm 1: TEXT2WORLD Generation Method

Input: Natural language domain description D_{NL}
Output: Generated PDDL domain D_{PDDL}

// Extract whole domain
$D_{\text{PDDL}} \leftarrow \langle \mathcal{D}_R, \mathcal{D}_T, \mathcal{D}_C, \mathcal{D}_P, \mathcal{D}_F, \mathcal{A} \rangle \leftarrow \text{LLM}(D_{\text{NL}})$
while pddl code block fails parsing **do**
    $D_{\text{PDDL}} \leftarrow \text{LLM}(\text{retry, error msg.})$
end while
// Final domain validation loop
while D_{PDDL} fails syntax validation **do**
    $D_{\text{PDDL}} \leftarrow \text{LLM}(\text{retry, error msg.})$
end while
return D_{PDDL}

Instead of generating whole domains at once, *NL2PDDL* decomposes the domain into action-step generation. First, we task the LLM to generate the high-level domain definitions (i.e., :types) before action generation. When all actions are iterated, we automatically infer the required PDDL features from the domain content and apply syntax-feedback correction loops.

Algorithm 2: NL2PDDL Generation Method

Input: Natural language domain description D_{NL}
Output: Generated PDDL domain D_{PDDL}

// Extract core domain components
$\langle \mathcal{D}_T, \mathcal{D}_C, \mathcal{D}_P, \mathcal{D}_F \rangle \leftarrow \text{LLM}(D_{\text{NL}})$
Initialize action list: $\mathcal{A} \leftarrow \emptyset$
for each action description a_i^{NL} in D_{NL} **do**
    // Construct Action
    $\langle \text{Par}_i^{\text{PDDL}}, \text{Pre}_i^{\text{PDDL}}, \text{Eff}_i^{\text{PDDL}} \rangle \leftarrow \text{LLM}(\text{a}_i^{\text{NL}})$
    $a_i \leftarrow \langle \text{Par}_i^{\text{PDDL}}, \text{Pre}_i^{\text{PDDL}}, \text{Eff}_i^{\text{PDDL}} \rangle$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{a_i\}$
end for
// Final domain assembly and validation
$D_{\text{PDDL}} \leftarrow \langle \mathcal{D}_T, \mathcal{D}_C, \mathcal{D}_P, \mathcal{D}_F, \mathcal{A} \rangle$
$D_{\text{PDDL}} \leftarrow \mathcal{D}_R$ *// Automatic :requirement fill-in*
// Final domain validation loop
while D_{PDDL} fails syntax validation **do**
    $D_{\text{PDDL}} \leftarrow \text{LLM}(\text{retry, error msg.})$
end while
return D_{PDDL}

Finally, our framework, *GRAN2PLAN* follows the same action-by-action generation method; however, we further decompose the task of each action by its parameters, precondition, and effect generation separately.

Algorithm 3: GRAN2PLAN Generation Method

Input: Natural language domain description D_{NL}
Output: Generated PDDL domain D_{PDDL}

// Extract core domain components
$\langle \mathcal{D}_T, \mathcal{D}_C, \mathcal{D}_P, \mathcal{D}_F \rangle \leftarrow \text{LLM}(D_{\text{NL}})$
Initialize action list: $\mathcal{A} \leftarrow \emptyset$
for each action description a_i^{NL} in D_{NL} **do**
    // Construct Parameters
    $\text{Par}_i^{\text{NL}} \leftarrow \text{LLM}(a_i^{\text{NL}})$
    $\text{Par}_i^{\text{PDDL}} \leftarrow \text{LLM}(\text{Par}_i^{\text{NL}})$
    // Construct Preconditions
    $\text{Pre}_i^{\text{NL}} \leftarrow \text{LLM}(D_{\text{NL}}, \text{Par}_i^{\text{PDDL}})$
    $\text{Pre}_i^{\text{PDDL}} \leftarrow \text{LLM}(\text{Pre}_i^{\text{NL}})$
    // Construct Effects
    $\text{Eff}_i^{\text{NL}} \leftarrow \text{LLM}(D_{\text{NL}}, \text{Par}_i^{\text{PDDL}}, \text{Pre}_i^{\text{PDDL}})$
    $\text{Eff}_i^{\text{PDDL}} \leftarrow \text{LLM}(\text{Eff}_i^{\text{NL}})$
    // Compose and append action
    $a_i \leftarrow \langle \text{Par}_i^{\text{PDDL}}, \text{Pre}_i^{\text{PDDL}}, \text{Eff}_i^{\text{PDDL}} \rangle$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{a_i\}$
end for
// Final domain assembly and validation
$D_{\text{PDDL}} \leftarrow \langle \mathcal{D}_T, \mathcal{D}_C, \mathcal{D}_P, \mathcal{D}_F, \mathcal{A} \rangle$
$D_{\text{PDDL}} \leftarrow \mathcal{D}_R$ *// Automatic :requirement fill-in*
// Final domain validation loop
while D_{PDDL} fails syntax validation **do**
    $D_{\text{PDDL}} \leftarrow \text{LLM}(\text{retry, error msg.})$
end while
return D_{PDDL}

D Prompting

We aim to keep prompts as consistent as possible. The prompt structure below was used to guide the extraction of PDDL action preconditions (substep in GRAN2PLAN). *Note*: original prompt has been slightly adapted for conciseness, while preserving its core structure to ensure faithful representation of the methodology.

```
[ROLE]: You are tasked with converting a given
Planning Domain Definition Language (PDDL)
domain description into its corresponding formal
PDDL domain action precondition. The description
will outline the essential components of the domains.
Your output should be a well-structured PDDL domain
action precondition that accurately represents the
given description, adhering to the syntax and
semantics of PDDL.

You may only use the types, constants, predicates,
and functions given. Do not introduce any new types.
-----------------------------------------------
[FORMAT]: Your output must strictly adhere to the
format exemplified below. Start your response with
'## RESPONSE', followed by your rationale and
thought process behind the PDDL precondition you
created. Keep your reasoning steps output concise.
End your final answer underneath the header (once)
'### Action Preconditions' followed by its content
```

````
that is enclosed by (```) comment blocks as so:

## RESPONSE
### Action Preconditions
```
(and
    (predicate_name ?t1 ?t2)
    (function_name_1 ?t1 - type_1 ?t2 - type_2)
)
```
------------------------------------------------
[EXAMPLE(S)]:
Example 1: [Blocksworld domain...]
Example 2: [Grippers domain...]
------------------------------------------------
[TASK]:
Here is the task to solve:
You need to generate the corresponding domain pddl
action  precondition for the following description.

## PDDL Domain Description: [domain_desc]
### Types, Constants, Predicates, Functions: [...]

Action to model:
### Action name: [action_name]
### Action description: [action_desc]
## Given Action Parameters: [action_parameters]

## RESPONSE
### Action Preconditions
```
[INSERT HERE]
```
````

The following is the prompt structure used to extract whole PDDL domains (TEXT2WORLD):

````
[ROLE]: You are tasked with converting a given
Planning Domain Definition Language (PDDL) domain
description into its corresponding formal PDDL
domain. The description will outline the essential
components of the domains. Your output should be a
well-structured PDDL domain that accurately represents
the given description, adhering to the syntax and
semantics of PDDL.

All actions MUST be modeled. You may only use the
types, constants, predicates, and functions given.
Do not introduce any new types.
------------------------------------------------
[FORMAT]: Your output must strictly adhere to the
format exemplified below with (```) enclosing blocks.

You MUST enclose the COMPLETE corrected PDDL within
```pddl```. Do not declare 'object' as a standalone
type in :types. It is a reserved keyword and should
only be used as a parent type for other types.

## PDDL Domain:
```pddl
(define (domain <domain_name>)
    (:requirements (...))
    (:predicates (...))
    (:action <action_name_1>
        :parameters (...)
        :precondition (...)
        :effect (...)
    )
    (:action <action_name_2>
        :parameters (...)
        :precondition (...)
        :effect (...)
    )
    (:action <action_name_n>
        :parameters (...)
        :precondition (...)
        :effect (...)
    )
)
```
------------------------------------------------
[EXAMPLE(S)]:
Example 1: [Blocksworld domain...]
Example 2: [Grippers domain...]
------------------------------------------------
[TASK]:
Here is the task to solve:
You need to generate the corresponding domain pddl
for the following description.

## PDDL Domain Description: [domain_desc]

Let's think step by step.

## PDDL Domain
```pddl
[INSERT HERE]
```
````

The following prompt was used to correct errors based on the syntax error message returned by the PDDL checker:

````
I would like you to serve as an expert in PDDL,
assisting me in correcting erroneous PDDL code. I will
provide you with the incorrect PDDL along with the
error messages returned by the system. You should
output your thought process firstly. You MUST enclose
the COMPLETE corrected PDDL within ```pddl```.

Hints to help you debug the pddl domain file:
1. You should start by checking if all the essential
domain constructs or domain definition constructs are
present. Commonly included domains comprise:
    a. :domain declaration to name the domain.
    b. :requirements to specify the PDDL features
        used in the domain.
    c. :types to define any object types for
        categorizing entities in the planning problem.
    d. :constants (if necessary) to declare any
        objects that remain static throughout.
    e. :predicates to define the properties and
        relations between objects that can change.
    f. :functions (if necessary) to define numeric
        functions for more complex domains.
    g. :action definitions for each action that
        agents can perform, including parameters,
````
````

```
        preconditions, and effects.
2. You need to check the number of parameters
    of each actions.
3. Having :typing in the domain indicates that it
    uses a hierarchy to organize objects. Therefore,
    it's crucial to clearly list all object types
    related to the planning task in a :types section.
4. '-' should not appear in :types.

Incorrect PDDL:
```pddl
  1 | (define (domain logistics-strips)
  2 |     (:requirements:strips :typing)
  3 | ...
 23 |     (:action LOAD-TRUCK
 24 |        :parameters (
 25 |           ?obj - obj ?truck - truck
                ?loc - location
 26 |        )
 27 |        :precondition
 28 |           (and
 29 |               (OBJ ?obj)
 30 |               (TRUCK ?truck)
 31 |               (LOCATION ?loc)
 32 |               (at ?obj ?loc)
 33 |               (at ?truck ?loc)
 34 |           )
 35 | ...
```
```

```
Error Information:
PDDLValidationError: Type mismatch in argument 1 of
predicate 'at' found in :precondition of action 'LOAD-
TRUCK': got `?truck` assigned type `truck`, expected:
    * `obj` (or its subtype(s))
Parsed error line: (at ?truck ?loc)
```

D.1 Error Correction

As discussed in Section 4.4, we chose to use the PDDL library for syntax validation and error messaging, in place of the TARSKI. Our preliminary experiments revealed that the error logs produced by TARSKI often lacked informative and interpretable content—even for human readers. While the PDDL library effectively detects most *syntactic* issues (e.g., missing brackets) and *declarative* issues (e.g., undefined predicates within actions), it falls short in identifying higher-level semantic errors when used in isolation. In particular, the base PDDL validator fails to detect:

- **Name Conflicts** across defined variables, types, constants, predicates, and numeric functions.
- **Undefined variable** in action scope for predicate and function usage. For instance, ?CAR was not defined in :PARAMETERS scope but used in an action schema.
- **Undefined expressions** used in action body. For instance, HOLDING(?ARM,?BLOCK1) was not defined in :PREDICATES but was used in action schema.
- **Invalid arity** of variables used in expressions in action body. For instance, original predicate definition was ON(?X,?Y), but ON(?X) was used in action schema.
- **Type mismatch** for expression arguments used in action body. For instance, LOAD-TRUCK(?T,?PACKAGE) was defined with types (TRUCK, PACKAGE) declared in :TYPES, but LOAD-TRUCK(?P,?PACKAGE1) was used in action schema where ?P is of type PLANE.

An advantage of using PDDL is its ease in code structure manipulation. That is, we were able to extract individual formulas from the domain in a recursive tree and apply our own custom validation checks on top of the original validator.

E Domain Prompt Structure

Each gold-standard domain is paired with a NL description from the TEXT2WORLD benchmark. We found that some instances also include domain-level details like :TYPES, :CONSTANTS, and :FUNCTIONS in the original dataset, though inconsistently. We applied minor fixes to the benchmark[5], such as correcting action name formatting to ensure proper parsing. The following is an example of an NL task description (not found in Text2World dataset):

```
### General
This domain models a robot navigating a grid environ-
ment with the objective of unlocking doors and moving
through the grid. The robot can carry keys that match
the shape of locks to unlock doors. The environment
includes places, keys with specific shapes, and doors
with corresponding shapes that need to be unlocked.

### Predicates
- (conn ?x ?y): Indicates a connection between two
    places ?x and ?y, allowing movement between them.
- (key-shape ?k ?s): Indicates key ?k has shape ?s.
- (lock-shape ?x ?s): Indicates lock (door) at
    place ?x has shape ?s.
- (at ?r ?x): Indicates key ?r is at place ?x.
- (at-robot ?x): Indicates the robot is at place ?x.
- (place ?p): Indicates ?p is a place in the grid.
- (key ?k): Indicates that ?k is a key.
- (shape ?s): Indicates that ?s is a shape.
- (locked ?x): Indicates place ?x is locked.
- (holding ?k): Indicates robot is holding key ?k.
- (open ?x): Indicates place ?x is open.
- (arm-empty): Indicates robot's arm is empty.

### Actions
- unlock <?curpos> <?lockpos> <?key> <?shape>: Allows
    the robot to unlock a door at place <?lockpos>
    using a key of a specific shape.
- move <?curpos> <?nextpos>: Allows the robot to move
    from place <?curpos> to place <?nextpos>.
- pickup <?curpos> <?key>: Allows the robot to pick
    up a key at its current location.
- pickup-and-loose <?curpos> <?newkey> <?oldkey>:
    Allows the robot to pick up a new key while
    dropping the one it was holding.
- putdown <?curpos> <?key>: Allows the robot to put
    down a key it is holding.
```

[5] *Note*: we removed the *rleap-project_d2l_domain_fstrips.pddl* domain from the dataset, as the abundance of keyword OBJECT usage did not align well with our current validator.

Table 4: Categorization of PDDL errors with descriptions and observed occurrence rates. Error Frequency (EF) reflects the number of times each error type was observed across LLM generation logs.

Category	Error Type	Description	EF
Syntax Errors	UNEXPECTEDTOKEN	Unexpected token that breaks parser expectations	**High**
	BADFORMULATERM	Malformed or invalid term in atomic formula	**None**
	MISSINGCODEBLOCK	No PDDL or Lisp code block was found in the input	**Low**
	MISSINGREQUIREMENT	Required PDDL feature not declared for formula usage	**Medium**
	INVALIDKEYWORDNAME	Invalid use of reserved keyword OBJECT as a name	**Medium**
	INVALIDCONSTANT	Malformed or invalid constant term	**Low**
Declaration Errors	NAMECONFLICT	Two or more identifiers conflict	**High**
	DUPLICATEQUANTIFIEDVAR	Duplicate quantified variable declared in same scope	**None**
	UNDECLAREDVARIABLE	Variable used without declaration in scope	**Low**
	UNDECLAREDPREDICATE	Predicate used but not defined in domain	**Low**
	UNDECLAREDFUNCTION	Function used but not defined in domain	**Low**
	UNDECLAREDTYPE	Type used but not defined in domain	**Medium**
Semantic Errors	TYPECYCLE	Cycle detected in type hierarchy	**High**
	TYPEMISMATCH	Argument has an incorrect or incompatible type	**High**
	MISSINGTYPE	Missing type specification for one or more arguments	**Low**
	PREDICATEARITYMISMATCH	Predicate usage contains incorrect number of arguments	**Low**
	FUNCTIONARITYMISMATCH	Function usage contains incorrect number of arguments	**Low**