Benchmarking and Improving PDDL Formalization Ability of Large Language Models with Planner-in-the-Loop Feedback

Anonymous ACL submission

Abstract

Planning depends on symbolic specifications that are both executable and verifiable. However, large language models often generate Planning Domain Definition Language (PDDL) problem descriptions that appear syntactically well formed yet fail under strict precondition, effect-consistency, and reachability constraints. Even minor specification errors can render a task unsolvable, motivating benchmarks and learning signals grounded in planner-based verification rather than surface plausibility.

We present NL-PDDL-Bench, a multi-domain benchmark for natural-language-to-PDDL specification construction, with planner-verified executability and object-count difficulty scaling. We also propose a planner-in-the-loop framework that uses validator and planner diagnostics to revise non-executable specifications via localized edits. Building on this toolchain, we present a planner-grounded optimization recipe combining parameter-efficient Low-Rank Adaptation supervised fine-tuning, offline planner-derived preference pairs for Direct Preference Optimization, and inference-time planner-in-the-loop repair, without online planner calls during training. We further provide a unified evaluation suite for parseability, solvability, specification similarity, and outcome-aware plan-level consistency against planner references. Experiments on eight representative model families show higher planner success and plan-level agreement, improved robustness under difficulty scaling and cross-domain transfer. Code and data are available at: https://anonymous.4open.science/r/NL-PDDL-Bench-BF76

1 Introduction

Planning is a core capability in artificial intelligence. Given an initial state and a set of goal conditions, a planner could synthesize an executable sequence of actions. This requirement underlies many applications, including robot control, autonomous driving, and logistics scheduling. Planning is intrinsically difficult because it demands a faithful model of action preconditions and effects, and it must satisfies reachability constraints over long horizons and large action spaces (Bercher et al., 2025; Pozo and Seipp, 2025).

Classical symbolic planners address these requirements through explicit, checkable specifications. PDDL is a widely adopted formalism that represents planning domains and problem instances in a structured, planner-verifiable form (Aeronautiques et al., 1998). When specifications are correct, state transitions and goal satisfaction can be validated by established planners and verification tools, yielding reproducible executability guarantees (Helmert, 2006; Howey et al., 2004). However, constructing such specifications is costly and typically requires expert knowledge, which limits scalability and practical deployment.

Recent large language models (LLMs) offer strong language understanding and structured generation abilities, motivating their use to reduce modeling effort (Valmeekam et al., 2023b). In practice, existing approaches largely follow two paradigms. The first paradigm directly generates action sequences from a task description. The second paradigm produces a symbolic specification that is subsequently solved by a symbolic planner (Huang et al., 2025; Mahdavi et al., 2024). Although both paradigms can produce outputs that appear plausible, reliability under planner verification remains the key obstacle.

For direct plan generation, actions that look reasonable often violate preconditions, break state consistency, or miss necessary constraints, and they fail under established planners (Valmeekam et al., 2023a; Kokel and Katz, 2025). More broadly, current LLMs remain far behind strong symbolic planners in terms of correctness and reliability on classical planning tasks, especially under strict reachability and long-horizon constraints. For symbolic

specification generation, outputs may be syntactically well formed yet still unsolvable due to subtle modeling errors such as missing objects, incorrect predicate usage, or incomplete initial facts (Silver et al., 2024). These failure modes show that planning requires consistency, executability, and verifiability that cannot be ensured by surface plausibility alone.

We address this challenge by releasing NL-PDDL-Bench and a unified planner-grounded pipeline that supports executable specification construction, optimization, and evaluation at scale. It couples large-scale planner-verified NL–PDDL pairs with a planner-in-the-loop feedback mechanism and planner-grounded training signals, enabling reproducible analysis and robust performance comparisons across domains, difficulty levels, and transfer settings.

Contributions:

- We propose a planner-in-the-loop feedback framework that turns validator and planner diagnostics into localized guidance for minimal edits toward executability.
- We open-source NL-PDDL-Bench, a multi-domain benchmark with planner-verified executable instances and controlled scaling by instance size.
- We introduce a planner-grounded optimization recipe for specification construction. It combines LoRA-based supervised fine-tuning, planner-derived preference signals via DPO, and inference-time planner-in-the-loop repair to systematically improve executability and plan-quality alignment.
- We develop a planner-grounded evaluation suite that measures solvability and plan-level agreement against reference solutions. Using this suite, we show consistent gains in planner success and plan-level agreement across model families, with improved robustness under difficulty scaling, cross-domain generalization, and transfer to PlanBench.

2 Definitions and Background

2.1 Planning Formulation

We focus on deterministic classical planning. A planning domain is denoted by $D = (V, A)$, where V is a finite set of predicate symbols (which induce ground atomic facts), and A is a set of action schemas with preconditions and effects. Given D, a problem instance specifies an object set O, an initial state s_0 (a set of ground atoms), and a goal condition g_{true} (Volkema, 1983):

$$P = (D, O, s_0, g_{\text{true}}) \tag{1}$$

A classical planner searches for a finite plan $\pi = (a_0, \ldots, a_n)$ such that each a_t is applicable in s_t and induces a valid transition

$$s_{t+1} = \gamma(s_t, a_t) \tag{2}$$

with the terminal state satisfying $s_n \models g_{\text{true}}$. Under the PDDL convention, the domain file declares predicates, types, and action schemas, while the problem file instantiates O, s_0, and g_{true}.

2.2 Introduction to PDDL

PDDL is a widely used formalism for classical planning (Gerevini, 2020). A specification is separated into a domain file and a problem file: the domain file defines predicate symbols, object types, and action schemas with explicit preconditions and effects, while the problem file instantiates a concrete task by declaring objects, initial facts, and goal conditions. This decomposition yields a normalized state–action representation that can be parsed and solved by symbolic planners, enabling reproducible verification of executability and solution quality.

In this work, PDDL serves as the formal interface for connecting natural-language task descriptions with symbolic planning toolchains. A large language model is required to produce problem specifications that respect the predicate signatures and action constraints defined by the domain. Deviations such as missing object declarations, inconsistent predicate usage, or incomplete initial facts can render an instance invalid or unsolvable, and these failures are reliably exposed by planner-based validation and solving, providing grounded signals for diagnosis and correction.

3 Method

3.1 Overall Framework

We present a planner-in-the-loop feedback framework for natural language planning that couples large language models with a standard planning toolchain to produce specifications that are parseable, solvable, and verifiable, together with executable plans. The key principle is to place the planner in the generation loop. The model focuses on extracting task entities and constraints from text and expressing them as a symbolic specification,

while the validator and planner provide rigorous checks of well-formedness and solvability. Their diagnostics are treated as reproducible evidence that supports subsequent targeted revisions. This design separates semantic interpretation from state-space search, enabling evidence-driven minimal edits that improve executability under fixed symbolic semantics, rather than relying on surface-form plausibility.

To enable systematic evaluation and optimization, we build NL-PDDL-Bench and use a unified symbolic toolchain throughout dataset construction, verification, training, and evaluation. Each instance includes an aligned natural-language description, its corresponding specification, and planner-derived reference signals, which support consistent measurement of validity, solvability, and plan-level alignment. Figure 1 summarizes the workflow, including benchmark construction, diagnosis-guided refinement, and unified symbolic evaluation.

On the learning side, we combine parameter-efficient supervised fine-tuning(SFT) with planner-grounded preference optimization to improve executability and solution-quality alignment. At inference time, when parsing or planning fails, the system converts diagnostic logs into structured feedback and prompts the model to perform localized repairs until verification succeeds or a fixed budget is reached. Formal definitions, training objectives, and evaluation metrics are specified in the subsequent subsections.

3.2 NL-PDDL-Bench

NL-PDDL-Bench is designed to provide a reproducible and planner-verifiable foundation for training and evaluating natural-language specification construction. Rather than relying on offline text-level alignment or manual rewriting, we build an end-to-end automated pipeline that combines top-down task design with bottom-up symbolic verification, following the general benchmark-construction spirit of automatically deriving LLM planning evaluations from PDDL resources (Stein et al., 2023). This pipeline enforces executable alignment: only instances that satisfy symbolic constraints and pass planner verification are included in the executable subset used for learning and evaluation, while failed instances yield structured diagnostic evidence under a unified toolchain. To make verification reproducible and diagnostically useful, we rely on planner-backed validation as a systematic check of semantic executability and inconsistency exposure, consistent with classical-planner-based verification and counterexample generation practices (Goldman et al., 2012), and we adopt a strong IPC-tested planning system as the underlying solver backbone (Corrêa et al., 2023).

Starting from classical IPC domains and their problem generators, we generate multi-domain planning instances with controlled scaling. We use

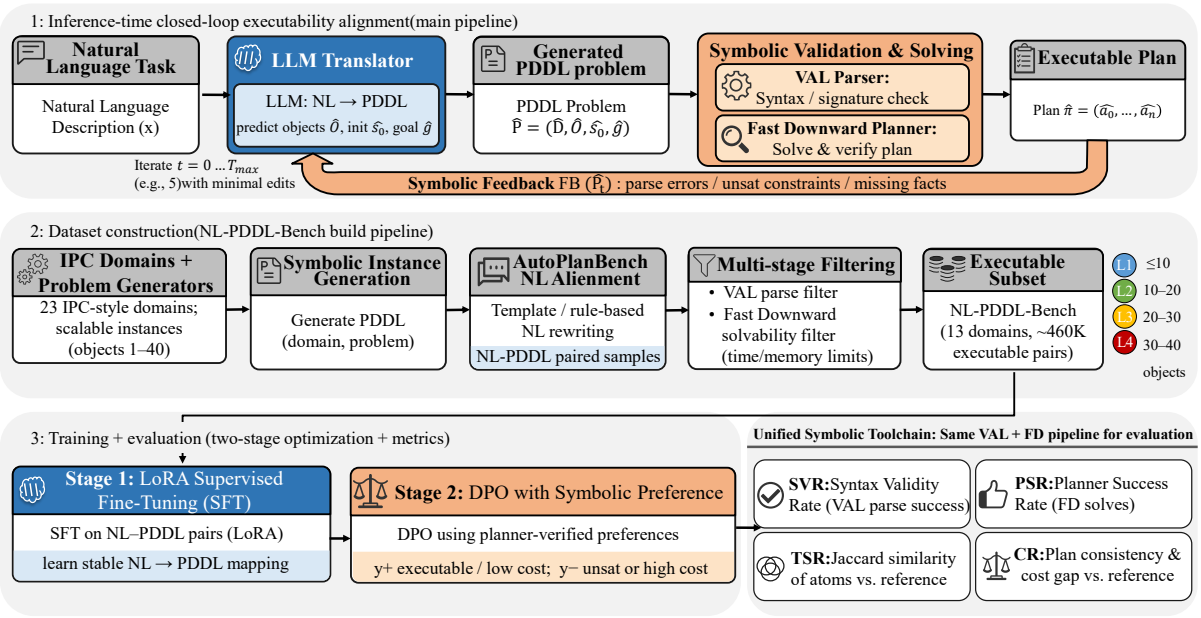

Figure 1: Overview of the planner-in-the-loop feedback framework, including benchmark construction, planner-in-the-loop feedback, and unified symbolic evaluation.

the number of objects as a unified difficulty surrogate and stratify problems into four levels (L1–L4), covering instances from 1 to 40 objects. For each symbolic instance, we then produce an aligned natural-language description via rule- and template-based, structure-preserving rewriting. The resulting descriptions explicitly encode initial conditions, goal conditions, and domain-critical constraints such as connectivity, capacities, and mutual exclusions, ensuring traceability between text and symbolic structure.

Table 1: Dataset distribution and planning statistics of NL-PDDL-Bench across difficulty levels.

Level	Objects	Solvable pairs	Plan length (median [Q1, Q3])
L1	1–10	154219	15.0 [12.8, 17.5]
L2	11–20	112003	39.0 [21.5, 91.5]
L3	21–30	93909	56.0 [19.0, 144.0]
L4	31–40	96918	86.5 [30.2, 208.5]

To guarantee executability and support planner-in-the-loop use cases, we apply multi-stage verification. We first run a validator to filter specifications that violate syntax, predicate signatures, typing, or arity constraints. We then invoke Fast Downward (FD) under a fixed resource budget and retain only planner-solvable instances as the executable subset, recording planner-derived reference signals including plans and plan cost/length, as well as failure logs for diagnosis. This pipeline starts from approximately 1.707M aligned NL–PDDL pairs and yields about 460K planner-solvable high-quality instances. Table 1 summarizes the resulting dataset distribution across difficulty levels and reports basic planning statistics.

3.3 LoRA-Based SFT and Planner-Grounded Preference Optimization

We adopt a two-stage optimization scheme to improve executability and plan-quality alignment of generated specifications. The first stage performs parameter-efficient SFT to learn a stable mapping from text to problem specifications, emphasizing object declarations, type consistency, predicate signatures, and faithful encoding of initial and goal facts. The second stage introduces planner-grounded preference optimization: we construct reusable preference signals via offline validation and planning, pushing the model toward specifications that are solvable and quality-aligned with reference solutions rather than merely well-formed.

For SFT, we use LoRA (Hu et al., 2022) and learn a low-rank update on top of frozen base weights,

$$W = W_0 + \Delta W = W_0 + \alpha B_{\text{LoRA}} A_{\text{LoRA}} \quad (3)$$

and minimize the standard negative log-likelihood,

$$\mathcal{L}_{\text{NLL}} = -\frac{1}{T}\sum_{t=1}^{T} \log P_\theta(y_t \mid y_{<t}, x) \quad (4)$$

Following common practice under constrained compute, we adopt QLoRA (Dettmers et al., 2023), which fine-tunes LoRA adapters on a frozen 4-bit quantized base model to reduce memory footprint while maintaining performance.

Supervision alone does not guarantee planner solvability. We therefore build an offline preference dataset: for each input x, the reference specification is treated as a positive sample y^+; candidate negatives are generated by programmatic perturbations and filtered by a validator and a planner to obtain y^- (unsolvable, or solvable but substantially worse than the reference in plan quality). We then apply Direct Preference Optimization (DPO) (Rafailov et al., 2023) by optimizing the policy f_θ against a frozen reference f_{θ_0}:

$$\begin{aligned}\mathcal{L}_{\text{DPO}} &= \mathbb{E}[-\log \sigma(\beta\Delta)]\,, \\ \Delta &= \Big(\log f_\theta(y^+ \mid x) - \log f_\theta(y^- \mid x)\Big) \\ &\quad - \Big(\log f_{\theta_0}(y^+ \mid x) - \log f_{\theta_0}(y^- \mid x)\Big)\end{aligned} \quad (5)$$

Since preference labels are produced entirely by offline planner verification, this stage avoids planner calls during backpropagation while injecting stable symbolic signals for executability and quality alignment. Details of perturbations and thresholds are provided in the appendix.

3.4 Planner-in-the-Loop Feedback and Multi-Dimensional Evaluation

We build a verifiable neuro-symbolic loop by using a symbolic planner as a unified interface for parsing, solving, and diagnosis, and by placing it in the generation process at inference time. Due to space constraints, the full feedback workflow is illustrated in Appendix C.2 (Figure 5). The model first produces a candidate problem specification; a validator and a planner then attempt to parse and solve it, returning structured diagnostics. When parsing or planning fails, we convert diagnostics into localized constraints and revision directives, and prompt the

model to apply minimal edits while preserving the intended task semantics. The loop terminates once the specification becomes executable or the budget is exhausted. This design turns non-executable outputs into attributable symbolic defects and provides formally grounded evidence for both evaluation and optimization.

At test time, we create an isolated workspace per instance and record the domain file, the reference specification, intermediate model outputs, and planner logs. We first run a one-shot generation and invoke validation and planning. If parsing fails, we trigger localized repairs guided by validator errors; if parsing succeeds but planning fails, we provide feedback targeting structural omissions and consistency conflicts and continue iterative repairs. The final prediction is the best verified specification produced within the evaluation budget.

Let $\mathcal{D}_{\text{test}} = \{(x_i, y_i)\}_{i=1}^{N}$, where x_i is the input description, y_i is the reference specification, and $\hat{y}_i$ is the model output (optionally after planner-in-the-loop refinement). We report four complementary metrics that emphasize executable alignment. Syntax Validity Rate (SVR) measures whether $\hat{y}_i$ is parseable:

$$\text{SVR} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\text{Parse}(\hat{y}_i) = \text{OK}] \tag{6}$$

Planner Success Rate (PSR) measures solvability conditional on parseability. Define $I_{\text{parse}} = \{\, i \mid \text{Parse}(\hat{y}_i) = \text{OK} \,\}$ and $I_{\text{solve}} = \{\, i \mid \text{Plan}(\hat{y}_i) = \text{SUCCESS} \,\}$, then

$$\text{PSR} = \frac{\left|I_{\text{parse}} \cap I_{\text{solve}}\right|}{\left|I_{\text{parse}}\right|} \tag{7}$$

Text Similarity Rate (TSR) measures atom-level overlap with Jaccard similarity:

$$\text{TSR} = \frac{1}{N}\sum_{i=1}^{N} \frac{|\text{Atoms}(\hat{y}_i) \cap \text{Atoms}(y_i)|}{|\text{Atoms}(\hat{y}_i) \cup \text{Atoms}(y_i)|} \tag{8}$$

where $\text{Atoms}(y)$ extracts the set of atomic facts from a specification y.

For plan-level alignment, we solve both the generated and reference problems under a fixed planner configuration and obtain $(\text{status}_i^{\text{gen}}, \pi_i^{\text{gen}}, c_i^{\text{gen}})$ and $(\text{status}_i^{\text{ref}}, \pi_i^{\text{ref}}, c_i^{\text{ref}})$, where c denotes plan cost or length. Consistency Rate (CR) is outcome-aware: it measures whether the planner outcomes agree (SUCCESS vs. FAILURE); when the reference instance is solvable, it further requires agreement in plan quality and structure:

$$\begin{aligned} \text{CR} &= \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\Big[\text{status}_i^{\text{gen}} = \text{status}_i^{\text{ref}} \ \wedge \Psi_i\Big], \\ \Psi_i &= \Big(\text{status}_i^{\text{ref}} = \text{FAILURE}\Big) \\ &\quad \vee \Big(\big|c_i^{\text{gen}} - c_i^{\text{ref}}\big| \le \epsilon_i \ \wedge \ \pi_i^{\text{gen}} \approx \pi_i^{\text{ref}}\Big) \end{aligned} \tag{9}$$

We detail the operational definition of $\pi_i^{\text{gen}} \approx \pi_i^{\text{ref}}$ and the choice of ϵ in Appendix C.3.

4 Experiments

4.1 Experimental Setup

Experiments are conducted on NL-PDDL-Bench to validate executability alignment and feedback-driven optimization across multiple model families, thirteen planning domains, and stratified difficulty levels. All evaluations rely on a unified symbolic toolchain: a validator checks well-formedness, and a symbolic planner assesses solvability and produces reproducible plans and failure diagnostics. Fast Downward (FD) is used throughout under a fixed configuration, yielding deterministic behavior and logs for consistent verification and attribution (Helmert, 2006). All reported numbers are computed with the same tool versions and resource limits to ensure comparability.

The test set is drawn from the executable subset of NL-PDDL-Bench. From approximately four hundred and sixty thousand planner-verified instances, we sample evenly across the thirteen domains and difficulty strata to obtain one thousand test instances. The test set is disjoint from the data used during model fine-tuning. For planner-in-the-loop evaluation, the maximum number of feedback iterations is fixed to $T_{\max} = 5$.

Evaluation targets a common failure mode in language-based specification generation: outputs may be syntactically well formed yet violate preconditions, effect consistency, or reachability constraints under planner verification. We therefore report executability metrics and plan-level alignment against planner-derived references, and conduct three analyses: (i) comparison with direct action-sequence generation, (ii) transfer experiments on PlanBench, and (iii) difficulty- and domain-wise analyses for robustness under scaling and cross-domain variation.

4.2 Comparative Results

4.2.1 Comparing Against Mainstream LLMs for Direct Plan Generation

This paper compare mainstream large language models that directly generate complete action sequences, including GPT-4o, Claude 3.5, and DeepSeek-R1, against our final method. Our method uses a low-rank-adapted Llama 3.1-8B model with symbolic feedback to produce a planning specification, and then applies the FD planner to synthesize an executable plan. We report planner success rate on three classic domains. For Logistics and Zenotravel, we evaluate 100 instances per domain sampled from our benchmark; for Blocksworld, we evaluate 600 instances from the PlanBench Mystery variant.

Table 2: Planner success rate for direct plan generation vs. our method.

Domain	GPT-4o	Claude 3.5	DeepSeek-R1	**Ours**
Logistics	22.4%	22.6%	56.8%	**78.0%**
Zenotravel	33.7%	57.9%	98.9%	**99.0%**
Blocksworld	0.0%	0.0%	43.3%	**87.3%**

We include plan generation as a baseline to test a central claim: when correctness is governed by preconditions, effect consistency, and reachability, fluent action sequences are not reliable evidence of executability. Direct generation must satisfy long chains of implicit constraints, so a single local omission can invalidate downstream transitions. This brittleness is pronounced in Blocksworld, where validity is tightly coupled to long-horizon state changes, whereas our approach leverages a verifiable symbolic interface that delegates search to a planner and uses planner-in-the-loop diagnostics for localized repair. Although direct generation can be strong in some domains (e.g., Zenotravel), it is less stable across domains. The gap is driven less by surface form than by missing constraint closure; planner-in-the-loop feedback turns global plan failures into localized, formally grounded corrections.

4.2.2 Overall Results of Mainstream Models

We evaluate eight representative model families on the NL-PDDL-Bench test set under two settings: Baseline, where each open-source base model follows the same prompt to directly generate a PDDL problem that is then solved under the same Fast Downward configuration, and our full pipeline that combines LoRA-based supervised fine-tuning, planner-grounded preference optimization, and inference-time planner-in-the-loop repair. We test a central claim: when correctness is jointly governed by preconditions, effect consistency, and reachability, failures are dominated by missing constraint closure rather than surface formatting.

Table 3 shows that this behavior holds broadly across model families. Averaged over all eight models, planner-in-the-loop raises planner success from 17.8% to 54.8% and plan-level agreement from 19.9% to 55.7%, while syntax validity increases more modestly from 66.8% to 74.7%. Except for GPT-OSS-20B, most baselines exhibit low PSR/CR, leading to larger absolute gains once planner feedback is introduced: PSR and CR increase (or remain unchanged) for every model, and TSR improves for seven out of eight models, consistent with planner-grounded signals correcting structural omissions and constraint conflicts.

The per-model results illustrate the same trend under diverse base capabilities. For instance, Llama 3.1-8B and Qwen3-8B both exhibit large gains in PSR/CR (e.g., for Llama 3.1-8B, PSR increases from 9.7% to 80.7% and CR increases from 10.9% to 81.2%; for Qwen3-8B, PSR increases from 24.8% to 78.9% and CR increases from 25.6% to 79.7%), indicating that planner diagnostics effectively convert global failures into localized, correctable revisions. In contrast, GPT-OSS-20B shows essentially flat PSR/CR with decreased TSR, suggesting deeper semantic mismatches that are less amenable to conservative local repair; we therefore conduct a domain-wise analysis of failure patterns. This analysis further localizes the anomaly to specific domains, indicating that the lack of improvement is driven by concentrated domain-level failures rather than uniform degradation across the benchmark. Overall, the results support our thesis that planner-in-the-loop feedback is most effective when failures are diagnosable and local, and that syntax compliance is neither necessary nor sufficient for executable alignment.

4.2.3 Comparison and Transferability on PlanBench

Table 4 contrasts NL-PDDL-Bench and PlanBench along task focus, scale, and verification. Both benchmarks are grounded in PDDL and rely on planner-based checking, but they target different bottlenecks. PlanBench evaluates planning and state-change reasoning from natural-language prompts, with a limited set of domains and small

Table 3: Overall results on the NL-PDDL-Bench test set. Parentheses report the absolute change of Planner-in-the-loop over Baseline for each metric.

Model	Setting	SVR	PSR	TSR	CR
Gemma3-4B	Baseline	37.2%	0.8%	6.8%	10.1%
	Planner-in-the-loop	60.6% (↑**23.4%**)	25.5% (↑**24.7%**)	32.3% (↑**25.5%**)	29.4% (↑**19.3%**)
GLM-4.1V-9B	Baseline	75.6%	9.9%	21.1%	11.1%
	Planner-in-the-loop	80.0% (↑**4.4%**)	63.3% (↑**53.4%**)	75.9% (↑**54.8%**)	64.2% (↑**53.1%**)
Llama3.1-8B	Baseline	76.3%	9.7%	52.9%	10.9%
	Planner-in-the-loop	84.9% (↑**8.6%**)	80.7% (↑**71.0%**)	82.8% (↑**29.9%**)	81.2% (↑**70.3%**)
Qwen2.5-7B	Baseline	76.8%	21.7%	61.2%	23.3%
	Planner-in-the-loop	81.8% (↑**5.0%**)	62.5% (↑**40.8%**)	78.8% (↑**17.6%**)	61.9% (↑**38.6%**)
Qwen3-8B	Baseline	82.1%	24.8%	61.5%	25.6%
	Planner-in-the-loop	82.7% (↑**0.6%**)	78.9% (↑**54.1%**)	80.8% (↑**19.3%**)	79.7% (↑**54.1%**)
GPT-OSS-20B	Baseline	69.9%	37.5%	39.9%	38.3%
	Planner-in-the-loop	79.0% (↑**9.1%**)	37.6% (↑**0.1%**)	31.5% (↓8.4%)	38.4% (↑**0.1%**)
DeepSeek-R1-7B	Baseline	84.6%	32.4%	34.1%	33.7%
	Planner-in-the-loop	90.2% (↑**5.6%**)	58.3% (↑**25.9%**)	57.0% (↑**22.9%**)	58.1% (↑**24.4%**)
DeepSeek-Coder-V2-16B	Baseline	32.2%	5.3%	13.1%	6.1%
	Planner-in-the-loop	38.5% (↑**6.3%**)	32.0% (↑**26.7%**)	32.8% (↑**19.7%**)	32.8% (↑**26.7%**)

problem sizes. NL-PDDL-Bench emphasizes verifiable specification construction under a unified symbolic toolchain, enabling systematic measurement of syntactic validity, planner solvability, and plan-level agreement against reference solutions. This design shifts evaluation from surface plausibility to executable correctness, and makes failures reproducible and diagnosable through planner logs.

Table 4: Key differences between Ours and PlanBench.

Aspect	NL-PDDL-Bench	PlanBench
Primary target	Spec construction and executability / plan-level alignment	Plan generation + cost-oriented evaluation
Domain coverage	**13** planner-verified domains; cross-domain by design	**2** IPC domains: Blocksworld, Logistics.
Scale	1.7M pairs (**460K** executable)	**26.3K** prompts
Instance size	Difficulty-stratified; **1–40 objects**	No explicit strata; typically **1–5 objects**
Verification	Unified parsing/solving and log-level diagnostics	Planner/validator checks; less standardized diagnostics
Training support	Large executable subset enables training + feedback optimization	Limited training-oriented support

Compared with PlanBench, NL-PDDL-Bench provides broader coverage and stronger training support. It spans thirteen planner-verified domains and controls difficulty through object-scale stratification, while PlanBench covers two domains with substantially smaller instances. NL-PDDL-Bench also offers a substantially larger collection of aligned pairs with an executable subset, which supports supervised training and feedback-driven optimization. Finally, NL-PDDL-Bench standardizes verification through consistent parsing and solving procedures and retains log-level diagnostics, enabling fine-grained attribution of errors such as signature mismatches, missing initial facts, and constraint conflicts. These properties make NL-PDDL-Bench complementary to PlanBench, and better suited for studying executability alignment and robustness under scaling and cross-domain transfer.

Table 5: Results on PlanBench (Baseline vs. Ours).

Metric	Qwen3-8B	Llama3.1-8B	GLM4.1V-9B
SVR	43.3%	99.8%	96.2%
	99.8% (↑**56.5%**)	100.0% (↑**0.2%**)	93.2% (↓3.0%)
PSR	0.0%	7.2%	21.7%
	32.2% (↑**32.2%**)	49.2% (↑**42.0%**)	69.0% (↑**47.3%**)
TSR	0.0%	0.0%	2.8%
	0.0%	0.0%	4.9% (↑**2.1%**)
CR	14.3%	7.2%	44.7%
	44.3% (↑**30.0%**)	49.3% (↑**42.1%**)	64.3% (↑**19.6%**)

To assess transferability, we evaluate our models on PlanBench using its original verification protocol. Table 5 shows that the full training and planner-guided refinement pipeline substantially improves planner success and plan-level agreement across

model families, even though the benchmark differs in domain scope and instance characteristics. The gains indicate that planner-grounded training signals and refinement generalize beyond the source benchmark, improving reliability under a different evaluation setting rather than merely overfitting to the construction pipeline.

4.3 Analytical Experiments

4.3.1 Performance under Difficulty Scaling

To test our central claim that planning reliability is governed by executable constraint satisfaction rather than surface-form fluency, we conduct a difficulty-scaling analysis. We stratify instances by object scale and evaluate how specification generation degrades as horizons and constraint coupling intensify, and whether planner-in-the-loop feedback can arrest such degradation.

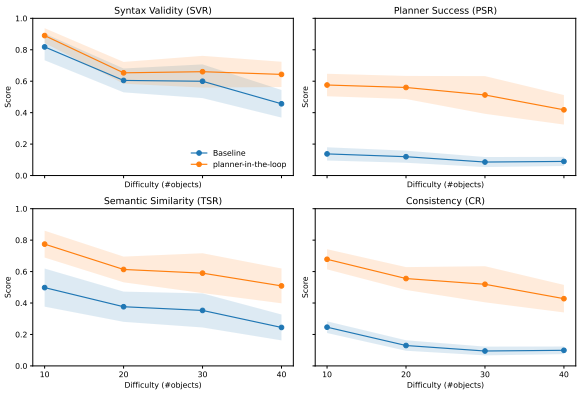

Figure 2: Performance under difficulty scaling.

The results support this diagnosis. As the object scale increases from 10 to 40, the baseline planner success drops from about 13% to about 9%, plan-level agreement drops from about 25% to about 9%, and specification similarity drops from about 50% to about 26%, with the overall score decreasing from about 42% to about 22%. In contrast, with planner-in-the-loop feedback, performance remains substantially more stable at the hardest scale: syntax validity is about 64%, planner success and plan-level agreement are both about 42%, and the overall score stays near 50%, suggesting improved robustness under scaling.

4.3.2 Cross-Domain Generalization

We further study cross-domain generalization to assess whether gains reflect domain-specific memorization or a general improvement in executable alignment. Using the same evaluation protocol, we compare baseline and planner-in-the-loop performance by domain and interpret metric patterns under different constraint structures. Because domains vary in their dominant constraints, executable alignment should primarily improve planner success and plan-level agreement across domains rather than only syntax.

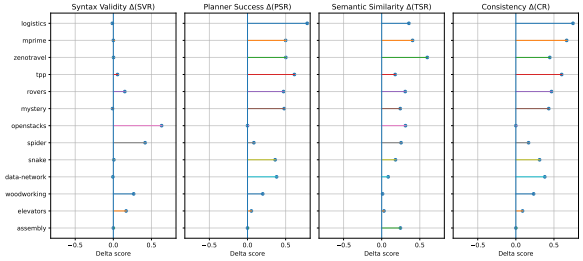

Figure 3: Cross-domain generalization results on NL-PDDL-Bench.

The observed trends support this expectation. In many domains, syntax validity is already relatively high and changes modestly, whereas planner success and plan-level agreement improve substantially, indicating that the main benefit comes from constraint-level alignment. For example, in Logistics, planner success increases from about 13% to about 78%, and plan-level agreement increases from about 20% to about 82%, consistent with repairing parseable but structurally invalid specifications such as missing connectivity facts or capacity conflicts. In contrast, in resource-intensive domains such as Openstacks, syntax improvements do not yield comparable gains in executability, reinforcing that syntax compliance is not a reliable proxy for executable alignment.

5 Conclusion

We introduce NL-PDDL-Bench, a multi-domain benchmark for natural-language-to-PDDL specification generation, with planner-verified executability and object-count scaling. We present a planner-in-the-loop framework that maps tasks to executable PDDL and applies localized repairs from validator and planner diagnostics. We further propose a planner-grounded optimization recipe, together with a unified evaluation suite for parseability, solvability, similarity, and outcome-aware plan consistency. Across eight model families and thirteen domains, we substantially improve planner success and plan-level agreement with only modest syntax changes, and transfer robustly to harder scales, new domains, and PlanBench.

6 Limitations

Although NL-PDDL-Bench and our planner-in-the-loop feedback framework provide a unified, planner-verifiable pipeline for executable alignment, our current study still has limitations along several dimensions:

- **Broader settings.** We evaluate on a fixed set of classical PDDL domains under a standardized validation and planning setup; extending to richer PDDL fragments (e.g., numeric fluents, temporals, derived predicates), alternative planners, and heterogeneous resource conditions is a natural next step.
- **Richer language.** Our benchmark uses structure-preserving, rule/template-based generation to maintain traceability between text and symbols; incorporating more diverse, naturally occurring instructions would broaden linguistic coverage.
- **Computational cost and configuration generality.** Our plan-level metrics and verification loop run under a fixed planner configuration and a bounded iteration budget, which improves reproducibility but incurs additional inference-time overhead and limits systematic evaluation across planners and heuristics. Future work may reduce cost via caching, early stopping, and adaptive budgeting, and further study stability across planner/heuristic configurations to strengthen scalability and deployment practicality.

We are actively exploring these directions in ongoing experiments, aiming to broaden coverage and improve practicality without changing the core executable-alignment protocol.

Acknowledgments

We thank the anonymous reviewers for their valuable comments.

References

Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, and 1 others. 1998. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*

Pascal Bercher, Sarath Sreedharan, and Mauro Vallati. 2025. A survey on model repair in ai planning. In *34th International Joint Conference on Artificial Intelligence*, page 26. IJCAI Organization.

Augusto B Corrêa, Guillem Frances, Markus Hecher, Davide Mario Longo, and Jendrik Seipp. 2023. The powerlifted planning system in the ipc 2023. *Tenth International Planning Competition (IPC-10): Planner Abstracts*.

Augusto B Corrêa, André G Pereira, and Jendrik Seipp. 2025. Classical planning with llm-generated heuristics: Challenging the state of the art with python code. *arXiv preprint arXiv:2503.18809*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115.

Alfonso Emilio Gerevini. 2020. An introduction to the planning domain definition language (pddl): Book review. *Artificial Intelligence*, 280:103221.

Robert P Goldman, Ugur Kuter, and A Schneider. 2012. Using classical planners for plan verification and counterexample generation. In *Proceedings of AAAI Workshop on Problem Solving Using Classical Planning*.

Malte Helmert. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Richard Howey, Derek Long, and Maria Fox. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Cassie Huang and Li Zhang. 2025. On the limit of language models as planning formalizers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4880–4904.

Sukai Huang, Nir Lipovetzky, and Trevor Cohn. 2025. Planning in the dark: Llm-symbolic planning pipeline without experts. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26542–26550.

Harsh Kokel and Michael Katz. 2025. Acpbench: Reasoning about action, change, and planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26559–26568.

Sadegh Mahdavi, Raquel Aoki, Keyi Tang, and Yanshuai Cao. 2024. Leveraging environment interaction for automated pddl translation and planning with large language models. *Advances in Neural Information Processing Systems*, 37:38960–39008.

Martın Pozo and Jendrik Seipp. 2025. Abstraction heuristics for classical planning tasks with conditional effects. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 8608–8616.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. 2024. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 20256–20264.

Pavel Smirnov, Frank Joublin, Antonello Ceravola, and Michael Gienger. 2024. Generating consistent pddl domains with large language models. *arXiv preprint arXiv:2404.07751*.

Katharina Stein, Daniel Fišer, Jörg Hoffmann, and Alexander Koller. 2023. Autoplanbench: Automatically generating benchmarks for llm planners from pddl. *arXiv preprint arXiv:2311.09830*.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023a. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36:38975–38987.

Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. 2023b. On the planning abilities of large language models (a critical investigation with a proposed benchmark). *arXiv preprint arXiv:2302.06706*.

Roger J Volkema. 1983. Problem formulation in planning and design. *Management science*, 29(6):639–652.

Xiaopan Zhang, Hao Qin, Fuquan Wang, Yue Dong, and Jiachen Li. 2025. Lamma-p: Generalizable multi-agent long-horizon task allocation and planning with lm-driven pddl planner. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10221–10221. IEEE.

Max Zuo, Francisco Piedrahita Velez, Xiaochen Li, Michael Littman, and Stephen Bach. 2025. Planetarium: A rigorous benchmark for translating text to structured planning languages. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11223–11240.

A Appendix: Introduction to PDDL

Planning Domain Definition Language (PDDL) is a standard formalism for representing deterministic classical planning tasks. A specification is decomposed into a *domain* and a *problem*. The domain defines predicate symbols, object types, and action schemas with explicit preconditions and effects. The problem instantiates a concrete instance by declaring objects, initial facts, and goal conditions. This separation yields a normalized state–action model that can be parsed and solved by symbolic planners, enabling reproducible verification of executability.

Figure 4 illustrates the semantics using a simplified Logistics example that contains a minimal but complete closed-world task: one truck must deliver one package from Location A to Location B. The example is split into three parts: the domain file `logistics.pddl` (left), the problem file `task.pddl` (top-right), and a schematic state-transition trace consistent with a valid plan (bottom-right).

Domain file (`logistics.pddl`). The domain declares `:requirements :strips :typing`, indicating that the task uses STRIPS-style add/delete effects with typed objects. It introduces three types—`truck`, `package`, and `location`—and three predicates: `(at ?x ?l)` encodes the location of either a truck or a package, `(in ?p ?t)` encodes that a package is loaded in a truck, and `(connected ?from ?to)` encodes reachability between locations. These predicates define the state vocabulary, i.e., the set of ground atoms that can be true in any state.

The domain further defines three action schemas. Action `drive` moves a truck from `from` to `to` when the truck is currently at `from` and the connectivity relation holds; its effect deletes `(at t from)` and adds `(at t to)`. Action `load` requires a package and the truck to co-locate, and updates the state by removing the package location fact `(at p l)` and adding `(in p t)`. Action `unload` is the inverse: it requires `(in p t)` and the truck to be at a location, then removes `(in p t)` and adds `(at p l)`. Together, these schemas constrain which transitions are legal and make executability checkable via precondition satisfaction and effect application.

Problem file (`task.pddl`). The problem instantiates a specific task under the above domain. It declares a truck `t1`, a package `p1`, and two locations

A and B, each with the correct type. The initial state sets t1 and p1 at A, and provides connectivity facts (connected A B) and (connected B A). The goal requires the package to be at B, i.e., (at p1 B). Importantly, this problem illustrates a common source of failure in language-to-PDDL generation: if connectivity is omitted or mis-specified, the instance can remain syntactically valid yet become unsolvable due to broken reachability constraints.

State-transition trace and an executable plan. Given the domain dynamics and the instantiated facts, a planner can synthesize an executable plan that achieves the goal. One minimal plan is: load(p1,t1,A), then drive(t1,A,B), then unload(p1,t1,B). Starting from the initial state, load is applicable because both t1 and p1 are at A; after applying its effects, the state contains (in p1 t1) and no longer contains (at p1 A). Next, drive is applicable because t1 is at A and (connected A B) holds; its effects move the truck to B. Finally, unload is applicable because p1 is in t1 and t1 is at B; after unloading, the state satisfies the goal atom (at p1 B). The schematic diagram in Figure 4 visualizes these intermediate states and highlights how PDDL makes the underlying constraints explicit and mechanically verifiable.

Relevance to NL→PDDL generation. When a large language model produces a problem specification from a natural-language description, it must preserve (i) type-correct object declarations, (ii) predicate signatures and arities, and (iii) domain-critical initial facts that ensure reachability and resource consistency (e.g., connected relations). Violations of (i)–(ii) typically yield parser/type errors, whereas violations of (iii) often produce instances that parse but are unsolvable. This example therefore captures the key distinction between surface well-formedness and true executability that motivates planner-based verification and feedback.

B Appendix: Related Work

This appendix situates our study within three closely related threads: language-model-based planning, symbolic interfaces with planner verification, and feedback-driven refinement. Across these threads, a common theme is the gap between surface plausibility and executable correctness under formal planning semantics. Our work addresses this gap by unifying benchmark construction, verification, and optimization under a single symbolic toolchain, while explicitly measuring plan-level alignment in addition to syntactic well-formedness and solvability.

B.1 Large Language Models for Planning

Large language models exhibit strong semantic parsing and structured generation capabilities, which has spurred growing interest in applying them to planning problems. Nevertheless, classical planning imposes a stringent notion of correctness: action applicability must satisfy preconditions, effects must preserve state consistency, and goals must be reachable through a valid transition sequence (Valmeekam et al., 2023a). Empirical benchmarks consistently show that models can generate plans that read as coherent yet fail under symbolic execution or validation, indicating that natural-language fluency is not a reliable proxy for constraint satisfaction or long-horizon state-transition correctness (Valmeekam et al., 2023a; Kokel and Katz, 2025). In contrast, established symbolic planners remain strong on well-specified domains, owing to mature heuristic search, soundness guarantees under the given semantics, and predictable performance profiles when resources are controlled (Corrêa et al., 2025).

Recent analyses further characterize the limits of LLMs as planning formalizers, showing that seemingly minor specification errors (e.g., in signatures, initial facts, or reachability-relevant constraints) can systematically break executability under strict symbolic semantics (Huang and Zhang, 2025). Our study is consistent with these observations, but goes beyond diagnosing failure modes by providing a unified benchmark and a planner-in-the-loop optimization pipeline that turns validator/planner outcomes into reproducible supervision signals and localized repairs, enabling systematic improvements in executable alignment rather than surface plausibility.

B.2 Symbolic Interfaces and Planner Verification

A complementary line of work adopts symbolic task specifications as the interface between language understanding and planning, enabling parsing, verification, and diagnosis through standard planning toolchains (Mahdavi et al., 2024). This interface has two practical advantages. First, it separates semantic interpretation from search, since the planner operates on a formal specification whose semantics are fixed by the domain model. Second, it makes failure modes more interpretable, because

(1) Domain: logistics.pddl

```
(define (domain logistics)
    (:requirements :strips :typing)
    (:types truck package location)
    (:predicates (at x - (either truck package) l - location)
                 (in p - package t - truck)
                 (connected from - location to - location))

    (:action drive
        :parameters (t - truck from - location to - location)
        :precondition (and (at t from) (connected from to))
        :effect (and (not (at t from)) (at t to)))

    (:action load
        :parameters (p - package t - truck l - location)
        :precondition (and (at p l) (at t l))
        :effect (and (not (at p l)) (in p t)))

    (:action unload
        :parameters (p - package t - truck l - location)
        :precondition (and (in p t) (at t l))
        :effect (and (not (in p t)) (at p l)))
)
```

(2) Problem: task.pddl

```
(define (problem task)
  (:domain logistics)
  (:objects t1 - truck p1 - package A B - location)
  (:init
    (at t1 A)
    (at p1 A)
    (connected A B)
    (connected B A))
  (:goal (and (at p1 B)))
)
```

(3) Schematic Diagram

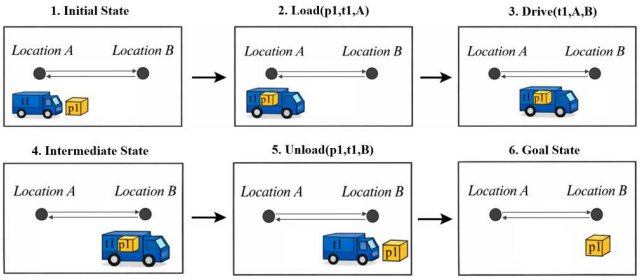

Figure 4: A simple example of the PDDL language.

validators and planners can return concrete error messages, unsatisfied conditions, or evidence of unreachable goals.

Benchmark efforts for text-to-structured planning languages provide an important foundation for positioning NL→PDDL as a rigorous translation-and-verification problem (Zuo et al., 2025). While such benchmarks emphasize careful evaluation of structured outputs, our focus is on *executable alignment at scale*: NL-PDDL-Bench provides a large planner-verifiable executable subset with controlled difficulty scaling, and our evaluation explicitly measures plan-level agreement under a fixed symbolic configuration. In addition, we treat planner logs and diagnostics as first-class artifacts that are reused consistently for dataset construction, evaluation, and optimization.

Related work also studies consistency checking for LLM-generated PDDL artifacts. For example, Smirnov et al. (2024) targets domain-level consistency when generating PDDL domains with LLMs, reducing contradictions in action/predicate definitions via automated checks. Our setting is complementary: we focus on problem-level specifications and emphasize failures that remain syntactically well-formed yet are unsolvable due to missing reachability facts or constraint closure. Accordingly, we rely on a validator-plus-planner pipeline that exposes not only static consistency issues but also planning-time reachability and resource conflicts, and we convert these reproducible diagnostics into localized repair signals.

B.3 Feedback-Driven Refinement

To address brittleness, recent studies incorporate external feedback to iteratively refine model outputs. One approach leverages environment or simulator interaction to produce executability signals and guide progressive correction (Mahdavi et al., 2024). Another approach couples generation with planner-based checking and replanning to recover feasible solutions in long-horizon settings (Huang et al., 2025; Zhang et al., 2025). These methods share the intuition that correctness should be anchored in externally verifiable signals rather than purely in model likelihood.

Despite clear progress, two limitations are recurrent in the literature. First, evaluations often focus primarily on feasibility, while providing less emphasis on whether the generated solutions match reference plan structure or achieve comparable solution quality under a fixed planning configuration. Second, analyses of robustness under controlled scaling and cross-domain variation are comparatively limited, making it difficult to characterize when feedback leads to general improvements versus domain-specific gains.

Our work addresses these gaps by introducing plan-level consistency and quality-alignment measurements grounded in planner-derived reference solutions, and by evaluating behavior under explicit difficulty stratification and multi-domain coverage.

This enables a more fine-grained view of what feedback corrects, and under what conditions improvements persist as constraints become more tightly coupled.

B.4 Our Position

Our study lies at the intersection of symbolic interfaces and feedback-driven refinement, with a primary focus on executable alignment and reproducibility. We contribute NL-PDDL-Bench, a multi-domain benchmark with planner-verified executability and controlled scaling by instance size, which supports systematic analysis across domains and difficulty levels. We further provide a unified symbolic toolchain that is used consistently for benchmark construction, verification, training-signal derivation, and evaluation, ensuring that correctness claims are grounded in the same planner semantics and are directly comparable across experimental settings.

Building on this foundation, we develop an end-to-end pipeline for specification construction that combines parameter-efficient supervised training, planner-grounded preference optimization based on offline planner-derived preference pairs, and inference-time planner-in-the-loop repair guided by validator and planner diagnostics. The pipeline is evaluated with complementary metrics that separate syntactic well-formedness from planner solvability and from outcome-aware plan-level consistency, thereby measuring executable correctness beyond surface plausibility.

In contrast to benchmarks that mainly frame natural-language-to-structured planning as a translation and evaluation task (Zuo et al., 2025), our work unifies verification and optimization under the same symbolic infrastructure. The toolchain that certifies executability also produces reusable diagnostics that support both preference construction and localized repair, enabling scalable and reproducible improvements in executable alignment under controlled difficulty scaling, cross-domain generalization, and transfer to external benchmarks.

C Appendix: Method Details

C.1 NL-PDDL-Bench

NL-PDDL-Bench is built from an initial pool of 23 IPC-style planning domains and their problem generators collected in AutoPlanBench, and processed under a unified symbolic toolchain for instance generation, natural-language rewriting, and verification. The released executable subset contains 13 fully verified domains: *Assembly, Data Network, Elevators, Logistics, MPrime, Mystery, Openstacks, Rovers, Snake, Spider, Traveling Purchaser Problem (TPP), Woodworking,* and *ZenoTravel*. These domains provide complementary constraint structures, spanning transportation and navigation tasks (e.g., Logistics, Rovers, ZenoTravel), resource- and capacity-intensive settings (e.g., Openstacks, TPP, Woodworking), and domains with stronger combinatorial structure in objects and relations (e.g., Assembly, Data Network, Mystery), which supports controlled cross-domain comparison and robustness analyses.

The reduction from 23 initial domains to the final 13 prioritizes reproducible executability and consistent NL–symbol alignment. Concretely, we retain domains whose generators produce stable multi-scale instances, whose domain/problem specifications are reliably parseable and solvable under a fixed validation and planning configuration, whose rule/template-based rewriting preserves traceability between text and symbolic structure, and whose verification pipeline yields a sufficient volume of executable instances for training and evaluation. The remaining domains are excluded when they (i) rely on PDDL features outside the fragment supported in our standardized toolchain, (ii) produce too few solvable instances under the fixed planner configuration and resource budget, or (iii) exhibit instability in generation-to-rewriting consistency that harms reproducibility. For transparency, we record the full initial domain pool and the excluded domains with their exclusion reasons in the released benchmark metadata.

Action costs and cost/length convention. To keep metrics comparable across domains, we use a uniform plan-cost convention in all experiments. By default, we assume unit action costs and treat plan cost as plan length; correspondingly, plan-level agreement is computed using length-based cost and structure under a fixed planner configuration. When an original IPC domain encodes action costs, we still report results with length as the cost definition under the same standardized configuration, so that cross-domain trends reflect executable alignment rather than heterogeneous cost modeling choices.

C.2 Planner-in-the-Loop Feedback Workflow

Figure 5 illustrates the prompting and verification loop used in our planner-in-the-loop workflow. The

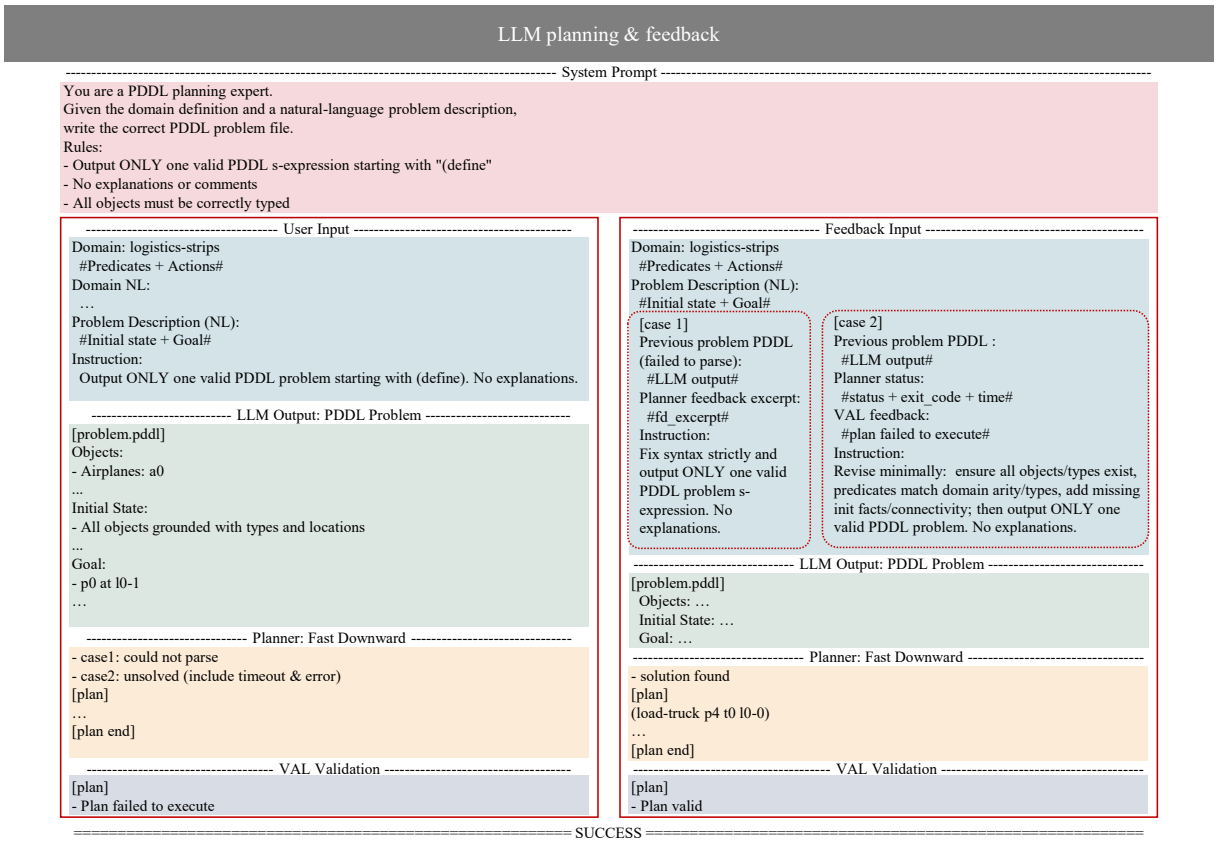

Figure 5: Prompting and planner-in-the-loop feedback process for iterative specification repair and verification.

goal is to convert a natural-language planning request into an executable PDDL problem by iteratively repairing errors revealed by a symbolic toolchain. Concretely, given a domain definition D and a natural-language problem description x, the model is instructed to output exactly one PDDL problem s-expression starting with `(define)`, with no additional commentary. This strict output contract minimizes formatting drift and makes downstream parsing and checking deterministic.

Step 1: One-shot specification generation. We first provide the model with (i) the domain name, (ii) a concise domain summary containing predicates and actions (including arities and types), and (iii) the problem description containing the initial state and goal conditions. The model outputs a candidate PDDL problem instance P. We emphasize type correctness and schema consistency by requiring that all objects be declared with valid domain types, and that each predicate instance matches the declared arity and argument types in D.

Step 2: Symbolic checking and plan synthesis. The generated problem P is fed into a standardized symbolic pipeline. We run a parser/validator to check well-formedness (e.g., balanced parentheses, well-typed objects, and legal predicate usage). If parsing succeeds, we invoke FD under a fixed configuration to assess solvability and, when possible, to produce a plan π. We then validate π using VAL to ensure that the plan is executable with respect to D and P (i.e., all preconditions hold along the execution trace and the goal is achieved).

Step 3: Feedback construction (two failure cases). When the pipeline fails, we construct feedback that is concise yet actionable and feed it back to the model for minimal repair.

- **Case 1 (Parsing/format failure).** If P cannot be parsed or fails basic well-formedness checks, we return a compact excerpt of the parsing error (e.g., unexpected tokens, missing parentheses, unknown symbols, or type declaration issues). The instruction is to fix syntax strictly and output only one valid PDDL problem.
- **Case 2 (Executable failure).** If P parses but FD reports `UNSOLVABLE`, times out, or produces a plan that fails VAL validation, we return (a) FD status metadata (e.g., status, exit

code, runtime) and a short log excerpt (e.g., unsatisfied preconditions, unreachable goals), and (b) VAL failure messages when available (e.g., the first violated precondition). The instruction is to revise minimally: ensure all referenced objects/types exist, predicates match domain arities/types, and add missing initial facts (e.g., connectivity) required by reachability constraints.

Step 4: Iterative repair and termination. The model revises the previous problem instance conditioned on the feedback and outputs a new candidate $P^{(t+1)}$. We repeat the symbolic checking and feedback steps until either (i) FD returns a plan and VAL confirms it as valid, or (ii) a maximum number of iterations $T_{\max}$ is reached. In all iterations, we preserve the same strict output contract (single `(define)` expression; no explanations) to keep the loop machine-checkable and to avoid introducing non-PDDL artifacts.

Rationale. This workflow operationalizes executability alignment by converting opaque generation failures into localized, verifiable repair signals. Parsing feedback targets syntactic correctness, while FD/VAL feedback targets semantic executability (preconditions, effects, reachability, and goal satisfaction). By delegating long-horizon search to a symbolic planner and using diagnostic logs for targeted revisions, the loop turns global plan failure into incremental, formally grounded corrections.

C.3 Consistency Rate Details

This appendix specifies the operational definition of Consistency Rate (CR), including the planner configuration, the cost/length convention, and the plan-structure agreement criterion. CR measures planner-grounded consistency between a generated specification and its reference under a fixed Fast Downward (FD) configuration. It first checks whether the planner outcomes agree (SUCCESS vs. FAILURE); when both sides are solvable, it further checks agreement in plan quality and action-level structure.

C.3.1 Planner configuration and cost definition

For each instance, we solve both the reference specification y_i and the generated specification $\hat{y}_i$ using the same Fast Downward configuration as in the main experiments. The planner outputs are denoted as $(\text{status}_i^{\text{ref}}, \pi_i^{\text{ref}}, c_i^{\text{ref}})$ and $(\text{status}_i^{\text{gen}}, \pi_i^{\text{gen}}, c_i^{\text{gen}})$. Unless stated otherwise, c denotes the plan length under unit action cost.Thus, cost agreement reduces to length agreement under an identical planner configuration.

C.3.2 Outcome-aware Consistency Rate

Let $\mathcal{D}_{\text{test}} = \{(x_i, y_i)\}_{i=1}^{N}$ be the test set, and let $\hat{y}_i$ be the model-generated specification. We define Consistency Rate (CR) as

$$\text{CR} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\left[\text{status}_i^{\text{gen}} = \text{status}_i^{\text{ref}} \ \wedge \ \Psi_i\right]. \tag{10}$$

where Ψ_i enforces additional requirements only when the reference instance is solvable.

$$\Psi_i = \begin{cases} \text{True}, & \text{if status}_i^{\text{ref}} = \text{FAILURE}, \\ \Phi_i, & \text{if status}_i^{\text{ref}} = \text{SUCCESS}. \end{cases} \tag{11}$$

The constraint Φ_i requires agreement in both plan cost and plan structure:

$$\Phi_i = \begin{array}{l} \left|c_i^{\text{gen}} - c_i^{\text{ref}}\right| \le \epsilon \\ \qquad \wedge \ \pi_i^{\text{gen}} \approx \pi_i^{\text{ref}} \end{array} \tag{12}$$

Thus, failed instances contribute to CR if and only if the planner outcomes agree, while plan-level comparisons are applied only when both specifications are solvable.

C.3.3 Plan-structure agreement $\pi^{\text{gen}} \approx \pi^{\text{ref}}$

Plans are compared at the level of grounded actions. Let $\text{act}(\pi)$ denote the sequence obtained by serializing each grounded action (e.g., `drive(truck1,locA,locB)`) in execution order.

Sequence similarity. We compute a normalized edit similarity between action-token sequences:

$$\text{Sim}_{\text{edit}} = 1 - \frac{\text{ED}\big(\text{act}(\pi^{\text{gen}}), \text{act}(\pi^{\text{ref}})\big)}{\max(|\pi^{\text{gen}}|, |\pi^{\text{ref}}|)} \tag{13}$$

where $\text{ED}(\cdot,\cdot)$ is the Levenshtein edit distance.

Bag-of-actions similarity. To mitigate non-uniqueness arising from commutative or weakly ordered actions, we additionally compute an order-insensitive multiset similarity. Let $\mathcal{M}(\pi)$ be the multiset of grounded actions in π. The multiset Jaccard similarity is defined as

$$\text{Sim}_{\text{bag}} = \frac{\sum_a \min\big(\mathcal{M}^{\text{gen}}(a), \mathcal{M}^{\text{ref}}(a)\big)}{\sum_a \max(\mathcal{M}^{\text{gen}}(a), \mathcal{M}^{\text{ref}}(a))} \tag{14}$$

Combined criterion (Scheme A). We define a commutativity-robust plan-structure similarity by

$$\mathrm{Sim} = \max\left(\mathrm{Sim}_{\mathrm{edit}},\ \mathrm{Sim}_{\mathrm{bag}}\right). \tag{15}$$

We consider $\pi^{\mathrm{gen}} \approx \pi^{\mathrm{ref}}$ if

$$\mathrm{Sim} \geq \tau \tag{16}$$

In all experiments, we set $\tau = 0.8$.

C.3.4 Tolerance ϵ for cost/length agreement

We define cost (length) agreement by an absolute tolerance ϵ applied uniformly across domains. Plan costs are considered consistent if

$$\left|c_i^{\mathrm{gen}} - c_i^{\mathrm{ref}}\right| \leq \epsilon_i \tag{17}$$

The tolerance ϵ is defined relative to the reference plan length as

$$\epsilon_i = \max\left(1,\ \lceil \rho\, c_i^{\mathrm{ref}} \rceil\right) \tag{18}$$

where $\rho = 0.05$ in all experiments.

This design allows minor plan-length deviations while still penalizing substantial departures from the reference plan quality under the same planner configuration.

D Appendix: Experiments Result

D.1 Training Environment

We ran our main experiments on a server equipped with six NVIDIA A100 80GB PCIe GPUs. Table 6 summarizes the compute environment.

Table 6: Compute environment for our main experiments.

Component	Specification
GPU	6× NVIDIA A100 80GB PCIe
CPU	2× AMD EPYC 9684X (96 cores per socket; 384 logical CPUs in total)
Memory	1.0 TiB RAM
OS	Ubuntu 24.04.1 LTS
Kernel	Linux 6.8.0-88-generic
CUDA	12.8

D.2 Training Dynamics of LoRA Fine-Tuning

As shown in Figure 6, we track the training dynamics of LoRA-based supervised fine-tuning for Qwen3-8B on NL-PDDL-Bench, including the loss curve, learning-rate schedule, and gradient-norm trajectory. The loss exhibits a clear fast-convergence pattern: it starts at approximately 0.18, drops rapidly within a small number of iterations, and falls below 0.01 at around 0.2 epochs. It then enters a stable plateau, fluctuating mildly within roughly 0.004–0.006. This suggests that the model acquires the core structural mapping for NL→PDDL early in training, while later updates primarily refine local constraint details and improve generation stability, yielding diminishing returns from additional epochs.

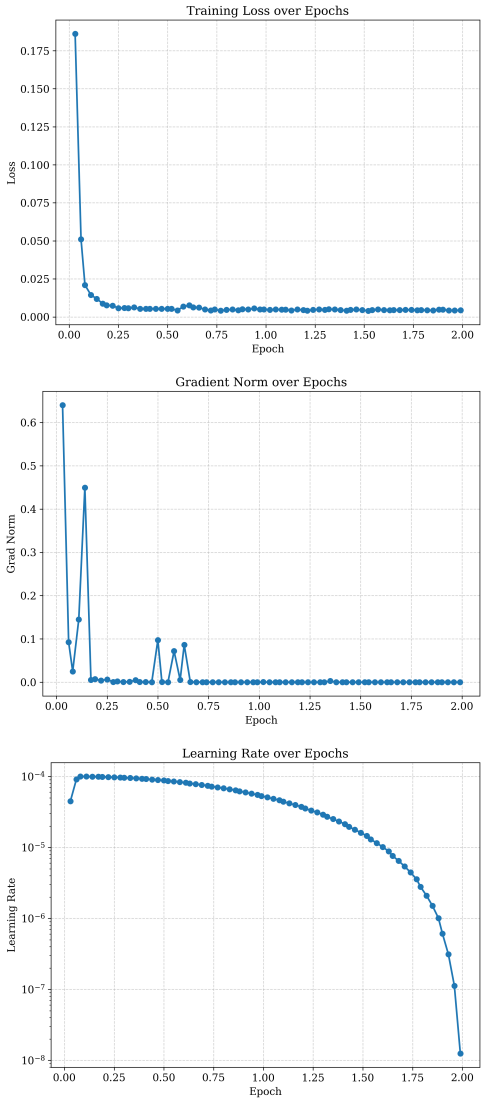

Figure 6: Training dynamics of LoRA fine-tuning for Qwen3-8B on NL-PDDL-Bench.

The learning-rate schedule and gradient-norm evolution further support this interpretation. The learning rate quickly warms up to around 1×10^{-4}, stays relatively high in the early-to-mid stage to encourage effective exploration, and then gradually decays to the order of 1×10^{-7} toward the end to stabilize convergence. Correspondingly, the gra-

dient norm peaks near the beginning and rapidly decays toward zero, with only occasional small impulses and no sustained growth or strong oscillations. Overall, these trends indicate stable and well-controlled optimization, providing a solid foundation for subsequently incorporating planner feedback for verification and refinement. Accordingly, our experimental focus moves beyond learning surface syntax templates to evaluating whether symbolic validation and feedback loops can reliably improve executability and plan consistency.

D.3 Case Study: Qwen2.5-7B

Figure 7 presents a case study in the Rovers domain, comparing Qwen2.5-7B under Baseline and our full framework across four object-count difficulty levels (10, 20, 30, and 40 objects) and four evaluation metrics (SVR, PSR, TSR, and CR). Rovers is representative of domains where correctness is determined by the closure of interacting constraints, including typed object declarations, consistent predicate grounding, and reachability-supporting facts that enable long-horizon state transitions. As the instance size grows, small omissions in the specification tend to propagate into global failures, because later actions require precise causal support and resource feasibility that cannot be recovered by local plausibility alone.

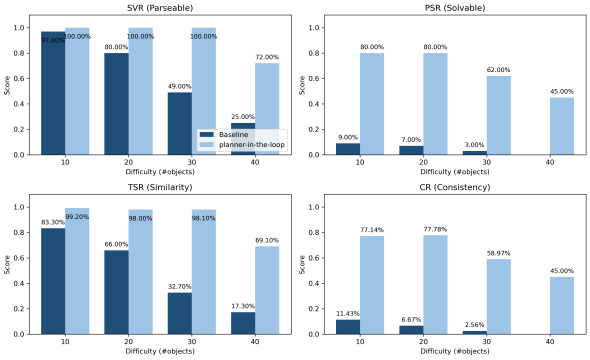

Figure 7: Case study: Qwen2.5-7B on Rovers across difficulty levels.

Under the Baseline setting, performance deteriorates sharply with scaling. While many outputs remain syntactically parseable at smaller scales, the gap between surface well-formedness and planner-grounded executability widens as constraints accumulate. This pattern suggests that baseline errors increasingly concentrate on semantic structure rather than formatting, such as missing enabling facts required for navigation and communication, inconsistent use of predicate arguments and types, or incomplete initialization of resources and locations. Such defects typically do not prevent parsing, yet they break reachability or precondition satisfaction, leading the planner to return UNSOLVABLE instances and reducing plan-level agreement with the reference specification. At the largest scale, the baseline often fails to maintain global constraint closure, which manifests as a collapse in solvability and consistency even when fragments of the specification appear locally reasonable.

In contrast, our framework mitigates this scaling-induced brittleness by coupling generation with verifiable symbolic feedback. The validator and planner expose concrete failure evidence, such as type or signature mismatches and unsatisfied preconditions, which we convert into targeted revision directives. This forces the model to repair missing causal links and reconcile constraint conflicts, rather than iterating on surface templates. As a result, the framework maintains substantially stronger executable alignment as difficulty increases, preserving solvability and plan-level consistency under tighter constraint coupling. The remaining degradation at the largest scale is consistent with the increased density of interacting constraints, where a single missed support fact can invalidate multiple downstream action chains. Overall, the case study indicates that the gains arise from improved semantic closure under fixed domain dynamics, not merely from better formatting, and that planner-grounded diagnostics provide an effective mechanism for turning global planning failures into localized, correctable specification edits.

D.4 Cross-Domain Generalization

We further study cross-domain generalization to determine whether the observed gains reflect domain-specific memorization or a transferable improvement in executable alignment. Using the same evaluation protocol as in the main experiments, we compare the Baseline and the planner-in-the-loop setting on each domain and examine how improvements distribute across the four complementary metrics. Because domains differ in their dominant constraint structures, a robust method should primarily improve planner-grounded criteria, namely planner success and outcome-aware plan-level consistency, rather than only increasing surface well-formedness.

For space reasons, the main paper presents the domain-wise delta plot in a single-column figure to fit the layout constraints. Here we include an en-

larged cross-column version (Figure 8) to improve readability and enable more careful inspection of domain-level patterns.

The results show that improvements are concentrated on planner-grounded correctness. Across most domains, changes in syntax validity are comparatively modest, while planner success and plan-level consistency increase substantially, indicating that the main benefit comes from repairing semantic defects that remain invisible to purely syntactic checks. This pattern is consistent with the role of planner-in-the-loop feedback, which targets constraint closure, reachability prerequisites, and resource consistency, and therefore reduces failures caused by missing initial facts, incompatible predicate usage, or latent precondition violations.

We also observe meaningful domain variation that helps interpret when planner feedback is most effective. Transportation-style domains, where failures often arise from omitted connectivity facts or incomplete state constraints, typically exhibit large gains in planner success and consistency once diagnostics are converted into localized revisions. In contrast, in domains with tighter resource coupling or more brittle modeling requirements, improvements in syntax validity do not necessarily translate into comparable gains in solvability, which suggests that the remaining errors are less local and may require richer semantic modeling or stronger priors to resolve. Overall, the domain-wise analysis supports the conclusion that planner-in-the-loop feedback improves executable alignment in a transferable manner, while also revealing constraint regimes where conservative local repair is insufficient.

D.5 Results of Mainstream Models on NL-PDDL-Bench

Tables 7 and 8 summarize domain-wise and difficulty-wise SVR/PSR/TSR/CR for mainstream models under the Baseline setting and our planner-in-the-loop framework. The results highlight a persistent gap between surface well-formedness and planner-grounded correctness: many baselines achieve non-trivial SVR yet frequently yield unsolvable instances, especially in domains with tight structural constraints and long-range dependencies, and the gap widens as object scale increases. Importantly, our CR is outcome-aware: it first measures whether the planner outcomes of the generated and reference instances match (SUCCESS vs. FAILURE); plan-cost and plan-structure agreement are only evaluated when the reference instance is solvable (and thus both sides succeed). Therefore, it is possible to observe high CR together with low PSR in subsets where the references are predominantly unsolvable, because consistent FAILURE outcomes contribute to CR while PSR only counts SUCCESS of the generated instances. By combining multi-stage symbolic validation (parsing, planning, and execution verification) with evidence-driven minimal repairs, planner-in-the-loop converts previously silent semantic inconsistencies into diagnosable error types and yields broad gains in executability-aligned metrics, with particularly large improvements in PSR and in CR on solvable references, indicating better satisfaction of domain-critical structures such as preconditions, reachability/connectivity facts, and resource constraints. Moreover, the smoother degradation of planner-in-the-loop under scaling suggests that planner-in-the-loop feedback provides an explicit correctness signal that helps preserve global consistency as constraints accumulate, improving reliability beyond what can be obtained by syntax compliance or template memorization alone.

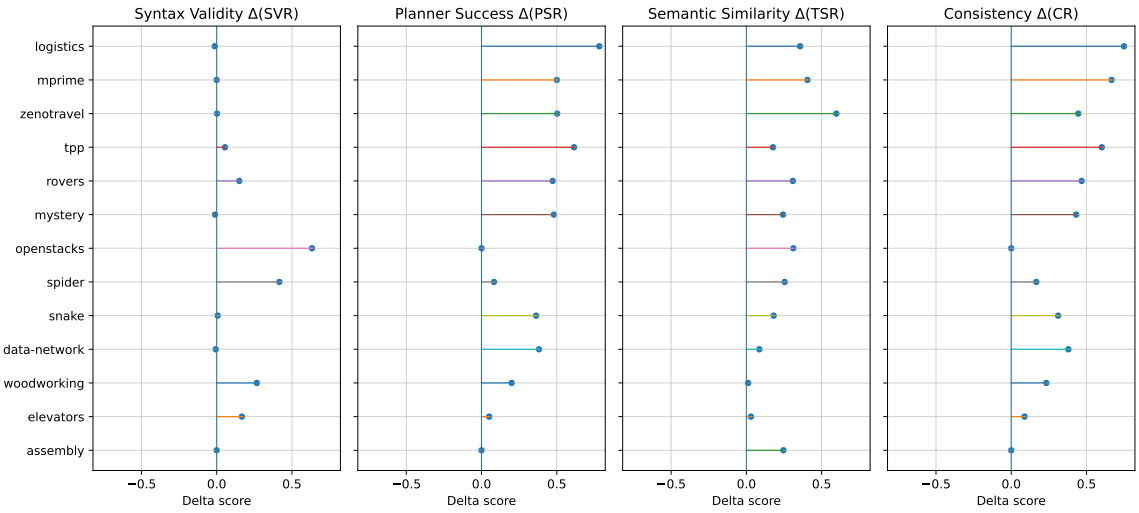

Figure 8: Cross-domain generalization results on NL-PDDL-Bench. This figure provides an enlarged cross-column view to facilitate readability and detailed comparison across planning domains.

Table 7: Domain-wise results on NL-PDDL-Bench. We report SVR/PSR/TSR/CR for each domain under Baseline and planner-in-the-loop.

Domain	Baseline				planner-in-the-loop			
	SVR	PSR	TSR	CR	SVR	PSR	TSR	CR
DeepSeek-Coder-V2-16B								
Assembly	100.0%	0.0%	23.5%	100.0%	100.0%	0.0%	37.3%	100.0%
Data Network	0.0%	0.0%	0.0%	0.0%	10.0%	0.0%	0.0%	0.0%
Elevators	2.0%	0.0%	1.9%	0.0%	12.0%	0.0%	2.3%	0.0%
Logistics	40.0%	6.0%	34.4%	5.6%	38.0%	38.0%	37.6%	37.6%
MPrime	100.0%	0.0%	56.7%	0.0%	100.0%	100.0%	100.0%	100.0%
Mystery	100.0%	84.0%	93.6%	84.2%	100.0%	100.0%	100.0%	100.0%
Openstacks	0.0%	0.0%	0.0%	0.0%	60.0%	0.0%	0.0%	0.0%
Rovers	1.0%	0.0%	0.4%	0.0%	14.0%	11.0%	11.8%	11.3%
Snake	41.0%	0.0%	35.3%	0.0%	46.0%	44.0%	43.6%	43.6%
Spider	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Traveling Purchaser Problem	13.0%	0.0%	9.6%	0.0%	18.0%	11.0%	10.7%	10.7%
Woodworking	0.0%	0.0%	0.0%	0.0%	80.0%	0.0%	0.0%	0.0%
ZenoTravel	100.0%	15.0%	15.5%	15.2%	100.0%	100.0%	100.0%	100.0%
GPT-OSS-20B								
Assembly	100.0%	0.0%	38.0%	100.0%	100.0%	0.0%	10.3%	100.0%
Data Network	88.0%	0.0%	57.5%	0.0%	69.0%	5.0%	17.3%	5.1%
Elevators	10.0%	2.0%	3.6%	1.6%	66.0%	2.0%	3.2%	2.2%
Logistics	90.0%	60.0%	72.7%	60.0%	89.0%	58.0%	63.2%	59.2%
MPrime	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%
Mystery	95.0%	89.0%	88.9%	89.5%	95.0%	74.0%	72.2%	73.7%
Openstacks	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	39.9%	0.0%
Rovers	52.0%	18.0%	19.9%	17.6%	59.0%	22.0%	14.4%	22.0%
Snake	46.0%	5.0%	28.0%	5.1%	54.0%	8.0%	12.6%	7.7%
Spider	100.0%	0.0%	46.3%	0.0%	0.0%	0.0%	0.0%	0.0%
Traveling Purchaser Problem	97.0%	30.0%	53.0%	30.0%	86.0%	38.0%	43.6%	37.6%
Woodworking	80.0%	0.0%	50.7%	0.0%	60.0%	0.0%	25.7%	0.0%
ZenoTravel	97.0%	96.0%	48.8%	96.5%	100.0%	86.0%	44.8%	86.3%

Table 7: Domain-wise results on NL-PDDL-Bench (continued).

Domain	Baseline				planner-in-the-loop			
	SVR	PSR	TSR	CR	SVR	PSR	TSR	CR
GLM-4.1V-9B								
Assembly	100.0%	0.0%	9.5%	100.0%	100.0%	0.0%	62.4%	100.0%
Data Network	92.0%	0.0%	25.2%	0.0%	100.0%	0.0%	94.9%	0.0%
Elevators	10.0%	0.0%	1.4%	0.0%	21.0%	0.0%	7.8%	0.0%
Logistics	93.0%	1.0%	24.1%	0.8%	100.0%	100.0%	99.9%	100.0%
MPrime	100.0%	0.0%	0.0%	0.0%	100.0%	100.0%	100.0%	100.0%
Mystery	100.0%	21.0%	26.3%	21.1%	100.0%	100.0%	100.0%	100.0%
Openstacks	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	50.1%	0.0%
Rovers	74.0%	8.0%	18.5%	8.2%	75.0%	57.0%	72.4%	57.2%
Snake	90.0%	0.0%	12.2%	10.3%	97.0%	92.0%	97.3%	92.3%
Spider	100.0%	0.0%	0.0%	0.0%	100.0%	50.0%	86.5%	50.0%
Traveling Purchaser Problem	93.0%	3.0%	15.4%	2.5%	94.0%	86.0%	92.1%	85.8%
Woodworking	80.0%	0.0%	59.5%	0.0%	80.0%	0.0%	54.2%	0.0%
ZenoTravel	99.0%	39.0%	45.8%	38.6%	100.0%	96.0%	100.0%	97.5%
DeepSeek-R1-7B								
Assembly	100.0%	0.0%	28.4%	100.0%	100.0%	0.0%	41.7%	100.0%
Data Network	90.0%	8.0%	32.5%	7.8%	94.0%	38.0%	63.1%	37.6%
Elevators	18.0%	0.0%	6.3%	0.0%	24.0%	6.0%	12.1%	5.8%
Logistics	92.0%	35.0%	58.9%	36.4%	100.0%	92.0%	97.6%	91.4%
MPrime	100.0%	12.0%	41.8%	11.6%	100.0%	68.0%	88.9%	66.7%
Mystery	100.0%	54.0%	62.3%	53.2%	100.0%	96.0%	97.1%	95.8%
Openstacks	0.0%	0.0%	0.0%	0.0%	72.0%	0.0%	36.4%	0.0%
Rovers	78.0%	42.0%	47.9%	41.6%	96.0%	83.0%	88.7%	82.4%
Snake	84.0%	21.0%	35.4%	20.8%	93.0%	64.0%	72.8%	63.7%
Spider	100.0%	0.0%	19.6%	0.0%	100.0%	22.0%	51.2%	21.5%
Traveling Purchaser Problem	86.0%	19.0%	39.1%	18.7%	92.0%	61.0%	75.4%	60.3%
Woodworking	72.0%	4.0%	28.7%	3.9%	88.0%	32.0%	55.6%	31.8%
ZenoTravel	98.0%	79.0%	61.2%	78.4%	100.0%	97.0%	98.3%	96.8%
Llama3.1-8B								
Assembly	100.0%	0.0%	57.7%	100.0%	100.0%	0.0%	93.1%	100.0%
Data Network	95.0%	0.0%	89.2%	0.0%	100.0%	100.0%	100.0%	100.0%
Elevators	14.0%	0.0%	5.4%	0.0%	18.0%	8.0%	8.5%	8.2%
Logistics	88.0%	5.0%	47.0%	5.6%	100.0%	100.0%	100.0%	99.2%
MPrime	100.0%	0.0%	100.0%	0.0%	100.0%	100.0%	100.0%	100.0%
Mystery	100.0%	0.0%	91.1%	0.0%	100.0%	100.0%	100.0%	100.0%
Openstacks	20.0%	0.0%	0.0%	0.0%	100.0%	0.0%	57.4%	0.0%
Rovers	69.0%	21.0%	54.9%	21.4%	100.0%	99.0%	99.5%	98.7%
Snake	97.0%	0.0%	49.3%	5.1%	97.0%	92.0%	97.4%	92.3%
Spider	100.0%	0.0%	59.7%	0.0%	100.0%	0.0%	91.5%	0.0%
Traveling Purchaser Problem	97.0%	20.0%	90.2%	19.8%	100.0%	100.0%	100.0%	99.0%
Woodworking	100.0%	0.0%	84.6%	20.0%	100.0%	80.0%	98.9%	80.0%
ZenoTravel	100.0%	9.0%	48.2%	9.1%	100.0%	98.0%	100.0%	98.5%

Table 7: Domain-wise results on NL-PDDL-Bench (continued).

Domain	Baseline				planner-in-the-loop			
	SVR	PSR	TSR	CR	SVR	PSR	TSR	CR
Qwen2.5-7B								
Assembly	100.0%	0.0%	54.7%	100.0%	100.0%	0.0%	60.1%	100.0%
Data Network	100.0%	0.0%	86.8%	0.0%	93.0%	42.0%	92.0%	40.7%
Elevators	15.0%	0.0%	6.8%	0.5%	12.0%	0.0%	6.3%	2.7%
Logistics	94.0%	0.0%	76.4%	0.0%	100.0%	92.0%	99.9%	87.2%
MPrime	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Mystery	100.0%	79.0%	94.8%	78.9%	100.0%	95.0%	99.5%	94.7%
Openstacks	0.0%	0.0%	0.0%	0.0%	40.0%	0.0%	33.0%	0.0%
Rovers	62.0%	4.0%	49.4%	5.0%	93.0%	67.0%	91.0%	64.8%
Snake	97.0%	0.0%	76.7%	12.8%	97.0%	69.0%	89.7%	71.8%
Spider	100.0%	0.0%	80.8%	0.0%	100.0%	0.0%	70.4%	50.0%
Traveling Purchaser Problem	100.0%	3.0%	96.6%	3.0%	99.0%	88.0%	99.2%	83.8%
Woodworking	100.0%	0.0%	83.3%	0.0%	100.0%	0.0%	75.8%	20.0%
ZenoTravel	99.0%	96.0%	63.7%	95.9%	100.0%	81.0%	96.5%	79.2%
Qwen3-8B								
Assembly	100.0%	0.0%	61.6%	100.0%	100.0%	0.0%	75.0%	100.0%
Data Network	100.0%	0.0%	100.0%	0.0%	100.0%	100.0%	100.0%	100.0%
Elevators	11.0%	0.0%	9.6%	0.0%	16.0%	6.0%	8.2%	6.0%
Logistics	100.0%	0.0%	62.7%	0.0%	100.0%	100.0%	100.0%	100.0%
MPrime	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Mystery	100.0%	100.0%	98.4%	100.0%	89.0%	89.0%	89.5%	89.5%
Openstacks	0.0%	0.0%	0.0%	0.0%	80.0%	0.0%	46.7%	0.0%
Rovers	97.0%	20.0%	75.0%	20.1%	99.0%	97.0%	98.4%	97.5%
Snake	90.0%	0.0%	83.7%	0.0%	97.0%	97.0%	97.4%	97.4%
Spider	50.0%	0.0%	47.2%	0.0%	100.0%	0.0%	91.9%	0.0%
Traveling Purchaser Problem	99.0%	0.0%	86.1%	0.0%	93.0%	93.0%	93.4%	93.4%
Woodworking	100.0%	0.0%	94.3%	0.0%	100.0%	40.0%	100.0%	40.0%
ZenoTravel	100.0%	99.0%	54.9%	99.5%	100.0%	100.0%	100.0%	100.0%
Gemma3-4B								
Assembly	100.0%	0.0%	10.7%	100.0%	100.0%	0.0%	37.5%	100.0%
Data Network	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Elevators	4.0%	0.0%	0.0%	0.5%	96.0%	1.0%	1.1%	21.2%
Logistics	71.0%	2.0%	19.0%	8.0%	38.0%	34.0%	37.6%	30.4%
MPrime	100.0%	0.0%	0.0%	0.0%	100.0%	0.0%	100.0%	100.0%
Mystery	100.0%	32.0%	58.5%	42.1%	100.0%	84.0%	100.0%	63.2%
Openstacks	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	0.0%	0.0%
Rovers	0.0%	0.0%	0.0%	0.0%	11.0%	7.0%	9.7%	6.3%
Snake	56.0%	0.0%	0.0%	30.8%	44.0%	28.0%	43.6%	25.6%
Spider	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Traveling Purchaser Problem	16.0%	0.0%	5.8%	6.1%	58.0%	8.0%	11.0%	16.8%
Woodworking	0.0%	0.0%	0.0%	0.0%	80.0%	0.0%	0.0%	20.0%
ZenoTravel	99.0%	0.0%	10.6%	25.4%	100.0%	80.0%	100.0%	72.1%

Table 8: Difficulty-wise results on NL-PDDL-Bench. We report SVR/PSR/TSR/CR under Baseline and planner-in-the-loop for each difficulty level (10/20/30/40).

Difficulty Level	Baseline				planner-in-the-loop			
	SVR	PSR	TSR	CR	SVR	PSR	TSR	CR
Gemma3-4B								
Level 1 (10)	58.0%	3.0%	17.3%	22.3%	65.0%	41.0%	54.6%	40.2%
Level 2 (20)	28.0%	0.0%	3.8%	3.6%	49.0%	26.0%	26.8%	26.8%
Level 3 (30)	39.0%	0.0%	2.4%	8.0%	64.0%	22.0%	28.8%	28.4%
Level 4 (40)	23.0%	0.0%	3.6%	6.4%	64.0%	14.0%	18.9%	22.1%
GLM-4.1V-9B								
Level 1 (10)	92.0%	12.0%	24.8%	16.3%	97.0%	73.0%	93.5%	75.7%
Level 2 (20)	74.0%	10.0%	25.6%	9.6%	80.0%	65.0%	77.0%	65.2%
Level 3 (30)	69.0%	12.0%	22.2%	12.0%	77.0%	65.0%	72.2%	65.2%
Level 4 (40)	67.0%	6.0%	11.6%	6.4%	67.0%	51.0%	61.0%	50.6%
Llama3.1-8B								
Level 1 (10)	94.0%	13.0%	65.0%	17.1%	97.0%	91.0%	96.5%	93.6%
Level 2 (20)	70.0%	10.0%	52.2%	10.4%	81.0%	77.0%	78.0%	76.4%
Level 3 (30)	71.0%	7.0%	45.1%	6.8%	80.0%	78.0%	78.2%	77.6%
Level 4 (40)	70.0%	9.0%	49.2%	9.2%	81.0%	77.0%	78.6%	77.1%
Qwen2.5-7B								
Level 1 (10)	98.0%	24.0%	80.2%	30.7%	97.0%	69.0%	91.5%	72.5%
Level 2 (20)	76.0%	19.0%	61.4%	18.8%	77.0%	58.0%	75.6%	55.2%
Level 3 (30)	69.0%	26.0%	52.8%	26.0%	78.0%	62.0%	76.2%	60.8%
Level 4 (40)	63.0%	18.0%	50.4%	17.7%	75.0%	60.0%	71.9%	59.0%
Qwen3-8B								
Level 1 (10)	97.0%	28.0%	78.9%	31.1%	95.0%	88.0%	93.9%	91.2%
Level 2 (20)	77.0%	20.0%	59.1%	20.4%	82.0%	78.0%	77.7%	77.6%
Level 3 (30)	78.0%	30.0%	53.5%	29.6%	78.0%	77.0%	77.1%	76.8%
Level 4 (40)	77.0%	21.0%	54.4%	21.3%	76.0%	73.0%	74.4%	73.1%
DeepSeek-R1-7B								
Level 1 (10)	94.0%	45.0%	48.6%	46.3%	96.0%	78.0%	81.4%	79.6%
Level 2 (20)	86.0%	34.0%	38.9%	35.2%	91.0%	63.0%	65.7%	62.4%
Level 3 (30)	82.0%	27.0%	31.4%	26.8%	89.0%	51.0%	54.1%	50.8%
Level 4 (40)	76.0%	18.0%	24.6%	18.9%	85.0%	41.0%	45.3%	40.7%
DeepSeek-Coder-V2-16B								
Level 1 (10)	53.0%	8.0%	29.1%	10.8%	62.0%	53.0%	56.6%	56.6%
Level 2 (20)	27.0%	3.0%	10.3%	2.8%	31.0%	27.0%	26.8%	26.8%
Level 3 (30)	30.0%	4.0%	6.6%	4.4%	36.0%	29.0%	28.8%	28.8%
Level 4 (40)	19.0%	6.0%	6.3%	6.4%	24.0%	19.0%	18.9%	18.9%