TRACS: COMBINING LLM-GENERATED PLAN TRAJECTORIES WITH CLASSICAL SEARCH FOR NUMERIC PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Numeric planning requires long-horizon reasoning under logical and numerical constraints. Advances in model architecture and increases in model scale have strengthened the planning capabilities of frontier large language models (LLMs), but a single inapplicable action can invalidate an entire generated plan. We introduce **TRACS**, a solver-centered hybrid that exploits LLM planning capability without requiring the model to solve each problem independently. Stagnation signals trigger occasional requests for LLM continuations from selected states. Symbolic replay retains valid prefixes for integration into native eager or lazy search. Symbolic validation preserves plan validity, while bounded intervention preserves the backbone's conditional completeness. We instantiate TRACS with a 9B proposer trained on solver-verified residual continuations using supervised fine-tuning followed by policy optimization, rewarding executable progress and penalizing state revisits. Across DEPOTS-NUM and ZENOTRAVEL-NUM, the lazy-search variant of TRACS achieves full coverage and substantially improves solution quality over native planners under the reported deployments.

1 INTRODUCTION

Numeric planning requires generating long action sequences that satisfy logical and numerical constraints (Ghallab et al., 2004; Fox & Long, 2003). Although advances in architecture and scale have strengthened the planning capabilities of frontier large language models (LLMs) (Corrêa et al., 2025), these models remain unreliable as end-to-end planners: a single inapplicable action can invalidate an entire plan (Goebel & Zips, 2025; Huang et al., 2025). Classical planners ensure plan validity through symbolic checks, but their efficiency depends on heuristic quality: weak guidance can lead to extensive search (Hoffmann, 2003; Kuroiwa et al., 2022; Scala et al., 2020b). Some approaches using learned heuristics or LLMs also incur inference overhead from frequent model calls (Chen et al., 2024; Borelli et al., 2026; Zhou et al., 2024).

The challenge lies in harnessing delayed, potentially incomplete or invalid LLM trajectories to improve search while preserving plan validity and the underlying planner's completeness guarantees. We introduce **TRACS**—**TR**ajectory-**A**ugmented **C**lassical **S**earch—to guide native search with sparse LLM-generated trajectories. Figure 1 summarizes how three components assist search: **(A)** three inexpensive stagnation signals at different levels identify a query state s_q for sparse intervention; **(B)** s_q becomes the initial state of a residual problem, for which the LLM proposes a multi-step continuation toward the goal and symbolic replay retains its valid prefix; **(C)** an eager or lazy adapter integrates the validated trajectory into native search as additional guidance. The planner continues searching while generation is pending. Both complete residual solutions and valid partial continuations can contribute to search. Our focus on numeric planning targets the additional difficulty of numerical constraints; the framework also applies to purely propositional planning.

Training for this search-assistance role requires learning from complete solutions and drawing optimization signals from partial progress when the former are unavailable, while discouraging reward hacking. We construct a dataset and train a 9B proposer through a two-stage pipeline of supervised fine-tuning (SFT) followed by group-based policy optimization. Experiments show marked gains in solution coverage, plan quality, and matched-quality search effort on difficult instances.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

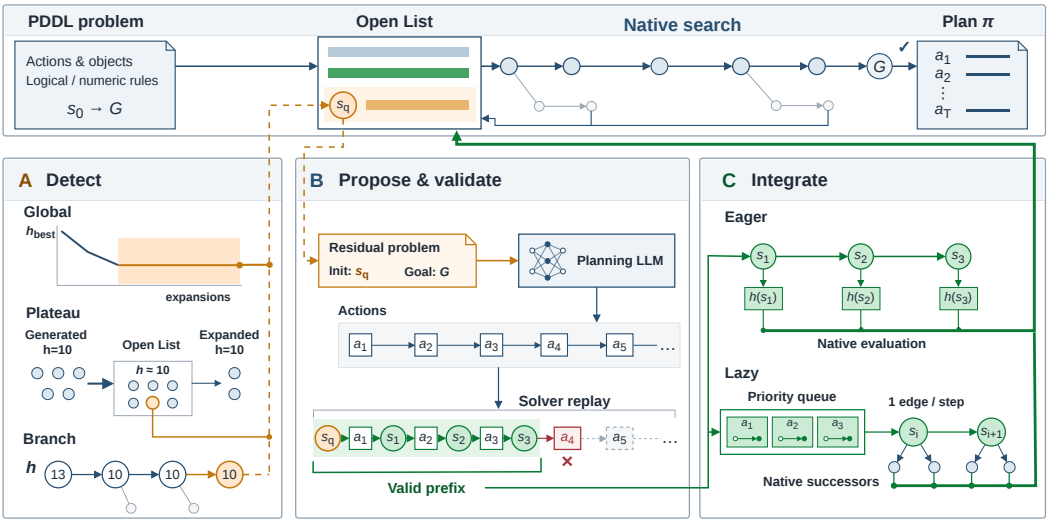

Figure 1: TRACS runtime overview. **Top:** PDDL input, native search, and plan output. **A:** Stagnation signals select a query state s_q for sparse intervention (orange). **B:** The LLM proposes actions for the residual problem; replay retains a valid prefix. **C:** Eager or lazy adapters integrate validated guidance (green). Native search continues while LLM responses are pending.

Our main contributions are as follows. **(1) Search integration.** We propose an asynchronous framework that integrates continuations from sparse LLM calls into native eager and lazy search, improving search performance while preserving plan validity and conditional completeness. **(2) Proposer training.** We develop a training pipeline for this search-assistance role that rewards goal attainment and usable partial progress while discouraging reward hacking, without domain-specific reward rules. Experiments show improved continuation generation and provide evidence of within-domain scale generalization.

2 Related Work

2.1 Classical Search and Macro-Actions

Classical planners use heuristics to prioritize symbolic state-space search (Hoffmann & Nebel, 2001; Helmert, 2006). Numeric planning methods include subgoaling and interval relaxations (Scala et al., 2020a; Aldinger & Nebel, 2017), LM-cut and operator-counting heuristics (Kuroiwa et al., 2021), and numeric pattern databases (Gnad et al., 2025; Fritzsche et al., 2026). Lazy evaluation offers another way to improve numeric search (Scala & Bonassi, 2025). These methods provide the symbolic search foundations for TRACS.

Macro-actions offer guidance by reusing action sequences. Marvin learns plateau-escaping macros online from paths found during search and reuses them on later plateaus (Coles & Smith, 2007); temporal macro-events are extracted from valid plans (La Farciola et al., 2025). TRACS instead generates candidate sequences with an LLM conditioned on the current residual problem.

2.2 End-to-End LLM Planning and Training

PlanGPT learns from solved planning instances (Rossetti et al., 2024), while PDDL-Instruct uses logical reasoning traces for instruction tuning (Verma et al., 2025). SafeGen-LLM combines SFT with verifier-grounded GRPO for safety-aware planning, including progress rewards for incomplete plans (Fan et al., 2026). These approaches ultimately require the LLM to produce a complete, valid plan, even when partial execution informs training or verification feedback guides revision. TRACS relaxes this requirement by using the LLM as a proposer: solving the residual problem is desirable, but an executable prefix of an incomplete proposal can also assist search.

2.3 Learned Guidance and LLM Integration

Learned heuristics use neural models to evaluate states (Ferber et al., 2022; Chen et al., 2024; Borelli et al., 2026). RAP, LATS, and SayCanPay rely on model-generated candidates for search expansion (Hao et al., 2023; Zhou et al., 2024; Hazra et al., 2024). TRACS instead requests trajectory continuations sparsely and asynchronously, while native symbolic search continues during inference.

Other hybrids connect LLMs to planning through symbolic problem and world-model construction (Liu et al., 2023; Guan et al., 2023), or through proposal refinement using external feedback (Kambhampati et al., 2024; Gonzalez-Pumariega et al., 2024). Generated plans can also provide a search structure: Tree-Planner organizes sampled plans into an action tree (Hu et al., 2024), while VOLTS combines symbolic action validation with branching during decoding (Massad et al., 2026). Silver et al. (2022) initialize the search frontier with states reached by LLM-generated plan prefixes. TRACS instead selects query states adaptively as search progresses.

3 Preliminaries

3.1 Numeric Planning and Residual Problems

A numeric planning problem is $\mathcal{P} = \langle F, X, A, s_0, G\rangle$, where F and X are finite sets of propositional and numeric variables, A is the action set, s_0 is the initial state, and G is the goal. A state $s = (s_F, s_X)$ assigns Boolean values to F and real values to X. Each action $a \in A$ has preconditions and effects. The effects specify changes to the truth values of propositions and the values of numeric variables; $\gamma(s, a)$ denotes the resulting state when $s \models \mathrm{pre}(a)$.

For a sequence $\pi = (a_1, \ldots, a_T)$, the prefix $\pi_{1:k} = (a_1, \ldots, a_k)$ is *valid* from s_0 if every transition $s_t = \gamma(s_{t-1}, a_t)$ is defined for $1 \le t \le k$. The complete sequence is a *valid plan* if all T transitions are defined and $s_T \models G$. The same definitions apply when execution starts from an intermediate state. The key insight behind TRACS is that even a valid prefix that does not reach the goal can advance search to a state farther along an executable trajectory, from which the planner can continue.

The remaining task from a reachable state s is the *residual problem* $\mathcal{P}_s = \langle F, X, A, s, G\rangle$, where $s \in \mathcal{S}_R$ and $\mathcal{S}_R$ denotes the states reachable from s_0. Only the initial state changes. A solution to $\mathcal{P}_s$ appended to a valid prefix ending at s yields a valid plan for the original problem.

3.2 Heuristic State-Space Search

Both eager and lazy search maintain an Open list of pending candidates and a Closed set of expanded states. Candidates are ranked using heuristic estimates $h(s)$ of remaining cost, optionally combined with accumulated path cost $g(s)$. Search repeatedly selects a candidate and obtains its state. Duplicate checks and pruning filter candidates during generation or selection. If a retained state satisfies the goal, predecessor pointers reconstruct the plan; otherwise, it is marked Closed and expanded through applicable actions, adding eligible candidates to Open.

Eager search constructs successor states and checks for duplicates before evaluating eligible successors and inserting them into Open. Each entry therefore refers to a state with its own heuristic value and recorded predecessor. **Lazy search** instead inserts predecessor–action edges (s, a), using the parent's $h(s)$ as the heuristic component of their priority. When an edge is selected, the planner constructs $s' = \gamma(s, a)$, checks for duplicates, and computes $h(s')$ if the state remains eligible for expansion. These differences motivate the distinct plateau signals in Section 4.1 and integration adapters in Section 4.3.

4 Our TRACS Approach

4.1 Stagnation Detection (A)

Since the best states for LLM intervention cannot be directly identified in advance, TRACS uses three inexpensive stagnation signals as proxies: global lack of improvement, persistent heuristic plateaus, and stalled branches (Figure 1). Global and branch stagnation suggest limited heuristic

progress, while plateaus suggest weak discrimination among states with similar h values; these conditions motivate additional guidance. A shared cooldown and request limits keep queries sparse. Detection intervals count ordinary base-search expansions, excluding LLM-origin rollout steps; evaluated rollout states may still update the best observed h. Plateau detection assigns each evaluated state to bucket $b = \mathrm{round}(h(s))$. We use separate plateau detectors for eager and lazy search.

Eager state-frontier plateau. For bucket b, let O_b and E_b count states generated into and expanded from that bucket within one counting window. Define

$$\rho_b^{\mathrm{eager}} = \frac{O_b}{\max(E_b, 1)}, \qquad G_b^{\mathrm{eager}} = \max(O_b - E_b, 0). \tag{1}$$

A bucket qualifies when $E_b \geq n_{\mathrm{e}}$, $\rho_b^{\mathrm{eager}} \geq \tau_\rho$, and $G_b^{\mathrm{eager}} \geq \tau_G$, and becomes active after K_{e} consecutive qualifying windows. This detects buckets where generation persistently outpaces expansion. When a query is permitted, TRACS samples an unexpanded state from an active bucket.

Lazy expansion-stream plateau. For expansion window $\mathcal{W}_t$, let $N_b^{(t)}$ count native evaluated states in bucket b. Define

$$\mathrm{active}_t(b) \iff \bigwedge_{j=0}^{K_{\mathrm{lazy}}-1} \left[N_b^{(t-j)} \geq n_{\min} \ \wedge \ \frac{N_b^{(t-j)}}{|\mathcal{W}_{t-j}|} \geq \alpha \right]. \tag{2}$$

This detects sustained concentration of native evaluations within a heuristic bucket. The next native expansion in the selected active bucket supplies the query state. Excluding LLM-origin rollouts prevents a proposal from generating its own plateau evidence.

Branch and global stagnation. At fixed expansion intervals, the branch detector follows d parent pointers from the current evaluated state s to an ancestor $a_d(s)$. It fires when $h(s) \geq h(a_d(s))$. The global detector counts eligible expansions since the best observed h last improved; it fires at a configured interval and rearms upon improvement. Both query the current evaluated state.

If several signals qualify, TRACS selects a request in the order plateau $>$ global $>$ branch. A candidate produces a request only when the shared cooldown, state deduplication, pending-request limit, and the budget of B state-level requests per native search phase permit it. Appendix A gives the full selection and bookkeeping algorithm.

4.2 Trajectory Proposal and Validation (B)

From the state s_q selected by A, TRACS generates and validates continuations for $\mathcal{P}_{s_q}$. The prompt uses Python-style documentation of the action signatures, logical and numeric preconditions and effects, objects, current state, and goal. The response contains a reasoning trace followed by a code block of grounded action calls; Appendix B illustrates the format used for training and inference. A single state-level request samples K independent continuations from this residual problem to reduce the impact of generation variability on overall search performance. Requests are asynchronous: the planner continues searching while generation is in progress, and responses remain tied to the recorded s_q.

TRACS parses the code block and maps its calls to grounded actions in the domain. The planner independently replays each candidate from s_q using its symbolic transition model. Replay stops before an unrecognized or inapplicable action. The retained action sequence and its replayed states supply the input to the integration adapter in Section 4.3; empty prefixes are discarded. Appendix A gives further replay and asynchronous request-handling details.

4.3 Integration into Native Search (C)

Validation determines which actions are executable; integration determines how they guide native search. **Eager integration.** For each state along the validated trajectory from Section 4.2, TRACS computes its heuristic value h, records its parent pointer and incoming action, and inserts it into Open after native duplicate and other checks. The inserted states therefore have the same representation as those generated by native search. **Lazy integration.** Lazy Open stores state–action pairs

whose successors await construction and evaluation. TRACS therefore uses a high-priority queue to favor the LLM-proposed actions for directed expansion through the planner's native lazy expansion procedure. When the queue empties, selection resumes from the base Open list.

Both methods supply additional guidance to native search through Open or the high-priority queue, including intermediate states along the proposed trajectories. For illustration, suppose a solution to a residual problem starts with 10 crate-rearrangement actions during which h remains 10. If many alternative arrangements share this value, the heuristic offers little guidance. A validated 15-action proposal reaching $h = 5$ or the goal can expose an exit before ordinary search discovers it: eager integration inserts eligible states into Open, whereas lazy integration prioritizes the action sequence. Even a 5-action prefix ending at a new state with $h = 10$ can advance exploration along this route, providing a continuation point within the plateau. Across all candidates from one state-level request, at most H validated transitions are integrated. The planner retains ordinary successor generation and existing search branches; only native expansion marks a state Closed.

4.4 Training the LLM Proposer

Dataset construction. For offline training of the proposer, we construct residual problems from solver-verified solutions. For each solution $\pi = (x_1, \ldots, x_n)$ to a problem $\mathcal{P}_{s_0}$, we uniformly sample at most p of its $n-1$ interior cut positions, subject to a minimum spacing between cuts. For each sampled position m, we replay the prefix from s_0:

$$s_i = \gamma(s_{i-1}, x_i), \quad i = 1, \ldots, m, \qquad y_m = (x_{m+1}, \ldots, x_n). \tag{3}$$

The intermediate state s_m serves as the initial state of the residual problem $\mathcal{P}_{s_m}$, with y_m as its corresponding solution. We use DeepSeek-V4-Pro (DeepSeek-AI, 2026a) as a teacher model to generate a reasoning trace for this problem–solution pair while keeping the actions in y_m unchanged.

Collecting verified continuations for stagnation-triggered queries would require additional runs for each backbone and heuristic and solving the resulting residual problems. We instead sample states along solver-verified solution paths, obtaining varied residual problems with valid continuation labels. Although this does not guarantee distribution matching, the trained proposers produce usable continuations on actual search-time queries and, within TRACS, improve coverage and quality (Q) while reducing matched-quality expansions (Table 3(b)).

Supervised fine-tuning. SFT teaches the reasoning–action format and solver-derived continuations. Errors in a few critical tokens can invalidate a sequence: for example, `crate0` and `crate8` refer to different objects. We upweight numerical object identifiers and arguments in the action block. For prompt x, target tokens $y = (y_1, \ldots, y_L)$, and critical-token positions $\mathcal{C}$,

$$\mathcal{L}_{\mathrm{SFT}} = -\frac{\sum_{t=1}^{L} w_t \log \pi_\theta(y_t \mid x, y_{<t})}{\sum_{t=1}^{L} w_t}, \qquad w_t = \begin{cases} \lambda, & t \in \mathcal{C}, \\ 1, & t \notin \mathcal{C}, \end{cases} \quad \lambda > 1. \tag{4}$$

Subsequent policy optimization starts from the SFT checkpoint, using symbolic replay to score newly sampled continuations for both goal attainment and usable partial progress.

Policy optimization. For each residual problem, GRPO samples a group of responses, normalizes their rewards into relative advantages, and optimizes a clipped policy objective (Shao et al., 2024). We use DAPO (Yu et al., 2025), which adds decoupled clipping, token-level loss, and dynamic resampling of zero-variance groups. Reward design must address two issues. *(i) Missing learning signals.* On difficult problems, goal-only rewards can assign the same failure reward to every response in a group, yielding zero relative advantages despite differences in partial progress. SafeGen-LLM uses verifier-derived progress rewards to alleviate this issue. *(ii) Reward hacking.* Such partial rewards can, however, favor easy-to-find, legal actions that inflate reward without helping search, as when raw executable length credits unproductive cycles. Appendix C.1.4 gives concrete examples of both failure modes.

Reward construction. We combine goal attainment, effective executable length, and bounded heuristic progress. Length provides a signal when h is flat, while heuristic progress adds information when h is informative. These remain proxies for search benefit: measuring actual savings would

require additional planner runs with and without each proposal. For a sampled trajectory τ, replay yields L executable actions and states $(s_0, \ldots, s_L)$, with s_0 denoting the query state. L_{eff} counts executable actions after excluding cyclic and repeated actions (Appendix C.1); C denotes plan cost for a successful trajectory. Non-goal responses need at least $L_{\min}$ effective actions for partial credit; shorter responses receive r_{fail}. We assign

$$R(\tau) = \begin{cases} R_{\text{goal}}(C), & s_L \models G, \\ \max(r_{\text{floor}}, R_{\text{len}} + R_h - P_{\text{loop}} - P_{\text{invalid}}), & s_L \not\models G,\ L_{\text{eff}} \geq L_{\min}, \\ r_{\text{fail}}, & \text{otherwise}, \end{cases} \tag{5}$$

Let $h_0 = h(s_0)$ and $h_{\text{best}} = \min_{0 \leq i \leq L} h(s_i)$. Then

$$R_{\text{len}} = w_{\text{len}} \frac{\log(1 + \min\{(L_{\text{eff}} - L_{\min})_+, L_{\text{cap}} - L_{\min}\})}{\log(1 + L_{\text{cap}} - L_{\min})}, \qquad R_h = w_{\text{best}} \Delta_h(h_0, h_{\text{best}}) + w_{\text{final}} \Delta_h(h_0, h(s_L)). \tag{6}$$

Here $(x)_+ = \max(x, 0)$ and $\Delta_h(a, b) = \text{clip}((a - b)/\max(|a|, \epsilon_h), 0, 1)$. To address *missing learning signals*, r_{floor} keeps usable partial rewards above r_{fail}, while differences in effective length and heuristic progress can distinguish proposals even when none reaches the goal. R_{len} grows logarithmically until $L_{\text{eff}} = L_{\text{cap}}$; R_h credits the best and final heuristic progress and is zero when h is unavailable. DAPO additionally resamples remaining zero-variance groups.

To curb *reward hacking*, L_{eff} excludes cyclic and repeated actions, while P_{loop} penalizes cyclic positions. Canonical state IDs detect revisits even when the action calls are distinct. $R_{\text{goal}}(C)$ remains higher than partial rewards and includes a bounded lower-cost bonus. P_{invalid} penalizes generation, parsing, or replay failure for non-goal responses; an executable sequence that misses the goal is not invalid. Appendix C gives constants and training settings.

4.5 Plan Validity and Conditional Completeness

Property 1: Plan validity. Every plan returned by TRACS is executable from s_0 and reaches a state satisfying G. LLM proposals may contain invalid actions, but TRACS validates each proposed transition before admitting it to search. The native planner also generates transitions according to the problem's symbolic rules. Once a goal state is found, tracing parent pointers back to s_0 reconstructs a valid plan because every transition along the path has been verified.

Property 2: Conditional completeness. Let $\mathcal{A}$ be a native search phase on $\mathcal{P}$. We retain the assumptions underlying its completeness: finitely many reachable native search states and duplicate/pruning rules that preserve solution reachability under reordered expansion. With these native conditions retained, TRACS preserves completeness through the integration rules in Section 4.3 and bounded intervention: request handling terminates, at most B state-level requests are issued, and each integrates at most H transitions. The guarantee holds for every LLM response sequence ω:

$$\forall \omega, \quad \text{Complete}(\mathcal{A}, \mathcal{P}) \Longrightarrow \text{Complete}(\text{TRACS}(\mathcal{A}; \omega), \mathcal{P}). \tag{7}$$

Proof. At most BH transitions are added, and request handling terminates, so asynchronous intervention cannot indefinitely delay ordinary search. TRACS retains ordinary Open entries; only native expansion closes a state, preserving the successors required by the native completeness argument. Under the stated duplicate/pruning conditions, guidance changes expansion order without removing solution reachability. On the finite native search-state set, the base search therefore still finds a solution whenever one exists. □

This guarantees eventual solution discovery, not optimality or success within a fixed resource budget. Appendix D details the native assumptions and contrasts this guarantee with repeated end-to-end generation; Section 5 evaluates planning performance and search effort.

5 Experiments

5.1 Experimental Setup

Data. We evaluate on the standard IPC domains Depots-Num and ZenoTravel-Num with native plan costs and four scales (10, 20, 30, and 40 entities per problem). For each domain, we

generate approximately 1,000 solvable problems at each of scales 10, 20, and 30, and approximately 250 at scale 40. We randomly select 5% of the resulting problem pool as the test set. From the remaining problems, we derive about 6,000 residual subproblems per domain using the method in Section 4.4. SFT uses these problems with their reference responses; DAPO uses only the problem prompts from a random subset of about 4,000 problems per domain.

Planners and protocol. We configure eager search in NLM-CutPlan (Kuroiwa et al., 2021; 2022) and lazy search in Count Downward (Scala & Bonassi, 2025), systems with IPC numeric-track wins (Taitler et al., 2024; IPC 2026 Numeric Track Organizers, 2026). Each native baseline uses its configured anytime search; TRACS retains the same search settings. All solvers have a search wall-clock limit of 1,800 seconds at scales 10–30 and 3,600 seconds at scale 40. CPU deployments differ; configurations and measured search throughput are documented in Appendices C.2.1 and C.3.3. We use Qwen3.5-9B (Qwen Team, 2026) as the TRACS proposer, trained separately per domain with SFT and DAPO.

The end-to-end API baselines are Kimi K2.6, DeepSeek-V4.1-Flash, and GLM-4.7 (Moonshot AI, 2026; DeepSeek-AI, 2026b; Z.ai, 2025). All use thinking mode. Each receives three complete-plan attempts without additional fine-tuning, using the prompt format in Section 4.2. Symbolic validation only checks whether returned plans achieve the goal. They assess direct planning capability without matching compute to TRACS. Appendix C specifies the API, training, and runtime settings.

5.2 Main Results

We use three main metrics to assess TRACS's overall performance and algorithmic benefits: **Coverage** is the fraction of problems solved within the wall-clock limit, measuring solving success. **Quality score** Q combines solving success and final plan quality at the same cutoff; higher is better (Table 1). **Expansion reduction** measures geometric-mean savings in native expansions needed to match the baseline's best cost at a paired expansion cutoff, within each backbone and over instances where both methods reach that cost (Table 2). It assesses native search efficiency with reduced dependence on hardware speed; counts exclude inference and integration overhead. API Pass@k counts problems solved within k attempts ($k = 1, 3$). Detailed configurations and evaluation protocols appear in Appendices C.2 and C.3. Despite differing hardware deployments, Depots throughput diagnostics show that TRACS achieves its gains with lower aggregate native-search throughput, supporting their attribution to search guidance (Appendix C.3.3).

Table 1: **Quality and coverage under the reported configurations.** Solver cells give Q/solved at wall-clock cutoffs; API cells give Pass@1/Pass@3 under attempt budgets. Columns show problem scale; n denotes the number of test instances at each scale. Bold marks the highest solver Q.

	Depots-Num				ZenoTravel-Num			
	Problem scale							
Method	10 (n=48)	20 (n=50)	30 (n=50)	40 (n=13)	10 (n=50)	20 (n=50)	30 (n=45)	40 (n=15)
Kimi K2.6	43/43	41/41	11/14	1/1	49/50	17/33	6/18	6/7
DeepSeek-V4.1-Flash	43/48	43/48	30/40	6/8	50/50	40/49	30/39	12/14
GLM-4.7	41/47	32/44	9/17	1/1	50/50	43/50	28/37	2/7
NLM-CutPlan	0.98/48	0.67/50	0.41/48	0.42/13	0.99/50	0.75/50	0.44/42	0.39/13
TRACS–Eager	0.98/48	0.67/50	0.42/49	0.38/11	0.99/50	0.76/50	0.45/43	0.40/13
Count Downward	**1.00/48**	0.99/50	0.71/39	0.25/4	**1.00/50**	0.96/50	0.69/37	0.24/5
TRACS–Lazy	**1.00/48**	**0.99/50**	**0.93/50**	**1.00/13**	**1.00/50**	**0.98/50**	**0.99/45**	**1.00/15**

Results and analysis. TRACS–Lazy achieves the strongest overall performance across both domains, with full coverage and the highest or tied-highest Q at every tested scale (Table 1). The native backbones exhibit a coverage–quality tradeoff: eager search more reliably finds feasible plans on difficult instances, whereas lazy search often returns lower-cost plans when successful but leaves more problems unsolved. LLM guidance makes TRACS–Lazy particularly effective at closing this coverage gap while further improving plan quality. Table 2 also shows fewer native expansions at matched quality on harder instances, supporting the algorithmic benefits of TRACS's guidance.

Table 2: **Geometric mean expansion reduction (%) at paired expansion cutoffs.** Brackets give 95% confidence intervals; n is the number of matched problems. D and Z denote DEPOTS-NUM and ZENOTRAVEL-NUM.

	TRACS–Eager vs. NLM-CutPlan		TRACS–Lazy vs. Count Downward	
Set	Reduction [95% CI]	n	Reduction [95% CI]	n
D-10	0.00 [0.00, 0.00]	48	0.00 [0.00, 0.00]	48
D-20	0.50 [−0.01, 1.41]	50	1.88 [−0.02, 4.53]	50
D-30	0.18 [−1.44, 1.95]	48	27.05 [9.65, 43.22]	31
D-40	4.53 [0.00, 12.98]	10	25.13 [−0.22, 48.31]	4
Z-10	0.00 [0.00, 0.00]	50	0.00 [0.00, 0.00]	50
Z-20	0.00 [0.00, 0.00]	50	1.10 [1.04, 2.19]	50
Z-30	1.62 [1.33, 4.46]	40	21.42 [10.94, 27.85]	32
Z-40	4.11 [1.17, 5.67]	11	25.10 [7.43, 44.78]	4

Larger gains on difficult instances are consistent with greater opportunities for intervention: easy searches can finish before LLM proposals arrive, whereas longer searches allow proposals to expose additional states. Hard lazy runs admit more states per request (Table 9), supporting this interpretation. Eager search may already have evaluated proposed states before responses arrive, consistent with its lower *Inj./req.* and smaller, mixed gains. Overall, TRACS–Eager achieves comparable or slightly better performance despite lower aggregate native-search throughput (Appendix C.3.3). Seeded Depots reruns retain the lazy variant's gains in coverage and Q (Appendix C.6).

End-to-end LLM baselines. Frontier LLMs often generate valid plans without additional fine-tuning, yet still fail on instances solved by every native configuration. This combination of planning ability and unreliable completion motivates using their validated proposals within symbolic search.

5.3 ABLATIONS AND MECHANISM ANALYSIS

To understand the sources and scope of TRACS's gains, we conduct three complementary evaluations. **Proposer training** evaluations test whether SFT and DAPO improve goal attainment and usable partial progress. Offline comparisons use fixed prompts and three attempts per problem: Goal@3 measures solving, Usable@3 and L_{eff} measure continuation usability and length, and TO/Loop track timeout/cycle failures. **Search performance** ablations test whether proposer training and query design improve solver outcomes, with TRACS hardware and native search settings fixed. Coverage, Q, and expansion reduction follow Section 5.2; Usable/call additionally measures the mean number of executable noncyclic prefixes per request. Offline comparisons also include Legacy GRPO, an earlier recipe that leaves zero-gradient groups on hard problems and reward hacking unaddressed (Appendix C.1.4). **Scale generalization** tests whether the final proposer can still supply usable partial progress beyond its training scales, using Goal@3 and Usable@3 for solving and continuation usability, and H-best/H-end for heuristic improvement along prefixes and at their endpoints.

Table 3 uses three DEPOTS-NUM problem sets: (a) residual problems derived from the test set (Section 4.4); (b) full test problems at scales 30–40; and (c) newly generated, held-out full problems at scales 45 and 50. Appendix C.4 details all three evaluation protocols.

Results. Proposer training. Continued DAPO training from SFT-1100steps strengthens goal attainment while usability remains high and effective length grows only modestly (Table 3(a)), consistent with more goal-directed proposals. Legacy GRPO underperforms SFT-1100steps, showing that policy optimization alone does not guarantee improvement. Its larger usability loss on hard problems motivates learning signals that remain informative when complete solutions are scarce.

Search performance. With either SFT-1100steps or DAPO, TRACS achieves full coverage, substantially improves Q, and reduces matched-quality expansions relative to Count Downward (Table 3(b)). In paired seed-101 runs, DAPO additionally yields higher plan quality averaged over paired expansion intervals (Appendix C.7). Together with its stronger offline goal attainment, this provides evidence of incremental gains from continued proposer training. Window-random inter-

Table 3: **Proposer training, search performance, and scale generalization on DEPOTS-NUM.** Rates in (a) and (c) are percentages; L_{eff} is the mean effective prefix length in actions. Arrows show preferred directions; bold marks unrounded column bests in (a–b), including ties.

(a) Proposer training: improved goal attainment and continuation usability.
Goal@3 and Usable@3 measure how often three attempts solve a problem or yield a usable continuation.

	Easy: scales 10–20 ($n = 134$)					Hard: scales 30–40 ($n = 185$)				
Checkpoint	Goal@3 ↑	Usable@3 ↑	L_{eff} ↑	TO ↓	Loop ↓	Goal@3 ↑	Usable@3 ↑	L_{eff} ↑	TO ↓	Loop ↓
Qwen3.5-9B Base	48.5	50.0	5.28	30.8	**1.5**	16.2	18.9	3.50	59.3	1.1
SFT-600steps	53.7	60.4	7.7	16.8	2.2	23.2	50.3	9.2	25.9	1.1
SFT-1100steps	77.6	**92.5**	11.07	**0.0**	3.0	33.0	85.4	16.02	**0.0**	1.3
DAPO-200steps	78.4	89.6	11.04	**0.0**	1.7	39.5	**86.5**	17.21	**0.0**	**0.9**
DAPO (ours)	**82.1**	**92.5**	**11.46**	**0.0**	1.7	**47.0**	85.9	**17.79**	**0.0**	**0.9**
Legacy GRPO	61.2	70.9	8.67	9.2	4.5	8.1	22.7	5.99	22.2	2.0

(b) Search performance: improved coverage and plan quality.
Coverage counts solved problems; Q rewards both solving success and lower plan costs. Expansion reduction measures search savings at matched quality.

Configuration	Samples	Coverage ↑	Q ↑	Usable/call ↑	Exp. reduction ↑
Count Downward	–	43/63	0.61	–	0.0%
Window-random + DAPO	3	53/63	0.75	2.30	15.1%
Adaptive + DAPO	1	50/63	0.69	0.76	13.4%
Adaptive + SFT-1100steps	3	**63/63**	0.91	2.39	**27.2%**
Adaptive + DAPO	3	**63/63**	**0.92**	**2.41**	26.8%

(c) Scale generalization: usable continuations beyond the training scales.
Despite zero Goal@3, Usable@3 remains substantial; H-best/H-end report heuristic improvement along prefixes/at endpoints over all problems.

Scale	Problems	Goal@3 ↑	Usable@3 ↑	L_{eff} ↑	H-best ↑	H-end ↑	TO ↓	Loop ↓
45	40	0.0	67.5	16.15	92.5	72.5	0.0	0.8
50	40	0.0	60.0	15.38	85.0	52.5	0.0	0.8

vention already benefits search; adaptive triggering improves further with the same sample count and comparable request counts. Under adaptive triggering, three samples outperform one but require more generations.

Scale generalization. Beyond the training scales, usable continuations and heuristic progress persist despite zero complete solutions (Table 3(c)). This indicates retained partial planning ability; the H-best/H-end gap further motivates using intermediate states as guidance.

6 CONCLUSION

We introduced TRACS, which uses sparse, asynchronous LLM proposals to guide native search while preserving plan validity and the base planner's conditional completeness. Training combines solver-verified suffix supervision with rewards for goal achievement and usable partial continuations, with penalties for state cycles.

Across both domains, TRACS–Lazy combines full coverage with the highest aggregate quality scores under the reported resource configurations. Eager effects are smaller and mixed. Matched-quality comparisons show fewer native expansions on difficult lazy-search problems, and Depots reruns retain gains in coverage and Q. SFT improves continuation usability, while DAPO strengthens offline goal attainment. Adaptive multi-sample triggering improves search performance, and usable continuations persist beyond the training scales.

The current implementation trains a separate proposer for each domain, although the integration mechanism is domain-independent. Future work will investigate cross-domain proposer training to improve generalization and reduce the need for domain-specific training.

AI USE STATEMENT

OpenAI Codex assisted with literature discovery, manuscript drafting and revision, the presentation of formal arguments, experimental protocol refinement, data analysis and interpretation, figure and table preparation, and implementation of author-specified methodological ideas. Google Gemini assisted with early engineering implementation.

For training-data construction, DeepSeek-V4-Pro generated reasoning traces around solver-verified action sequences (Section 4.4). Qwen3.5-9B was trained as the TRACS proposer, while DeepSeek-V4.1-Flash, GLM-4.7, and Kimi K2.6 were evaluated as direct planning baselines; their roles and experimental settings are described in Section 5.1 and Appendix C.

The authors reviewed AI-assisted text and analyses, checked figures and tables against their underlying data, checked formal claims against the native search semantics and stated assumptions, and decomposed AI-assisted code into modules for module-level functional testing. Action sequences in generated training examples were replay-validated by the symbolic solver before use. The authors take full responsibility for the final content of this work, including all text, claims, code, figures, tables, and data artifacts produced with generative-AI assistance.

REPRODUCIBILITY STATEMENT

Section 4 specifies the TRACS components and completeness assumptions. Appendices A and B provide the runtime algorithm and prompt format. Section 4.4 and Appendix C.1 document data construction, training, and rewards. Appendix C.2 collects planner, hardware, and inference settings. Appendix C.3 defines the evaluation cutoffs, quality score, reference pools, paired comparisons, and bootstrap procedures; Appendix C.4 specifies the ablation evaluations. Appendices C.5, C.6, and C.7 report solution-quality and intervention diagnostics, seeded reruns, and the paired checkpoint comparison, respectively.

REFERENCES

Johannes Aldinger and Bernhard Nebel. Interval based relaxation heuristics for numeric planning with action costs. In *KI 2017: Advances in Artificial Intelligence*, volume 10505 of *Lecture Notes in Computer Science*, pp. 15–28. Springer, 2017. doi: 10.1007/978-3-319-67190-1_2.

Valerio Borelli, Alfonso Gerevini, Enrico Scala, and Ivan Serina. Learning heuristic functions with graph neural networks for numeric planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 36172–36179, 2026. doi: 10.1609/aaai.v40i43.40935.

Dillon Z. Chen, Sylvie Thiébaux, and Felipe Trevizan. Learning domain-independent heuristics for grounded and lifted planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 20078–20086, 2024. doi: 10.1609/aaai.v38i18.29986.

Andrew I. Coles and Amanda J. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007. doi: 10.1613/jair.2077.

Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. Frontier large language models rival state-of-the-art planners. *arXiv preprint arXiv:2511.09378*, 2025. doi: 10.48550/arXiv.2511.09378.

DeepSeek-AI. DeepSeek V4: Technical documentation, 2026a. URL `https://fe-static.deepseek.com/chat/transparency/deepseek-V4-model-card-EN.pdf`. Technical documentation and model card.

DeepSeek-AI. DeepSeek-V4.1-Flash: Technical report, 2026b. URL `https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/blob/main/DeepSeek_V41_Tech_Report.pdf`.

Jialiang Fan, Weizhe Xu, Mengyu Liu, Oleg Sokolsky, Insup Lee, and Fanxin Kong. SafeGen-LLM: Enhancing safety generalization in task planning for robotic systems. *arXiv preprint arXiv:2602.24235*, 2026. doi: 10.48550/arXiv.2602.24235.

Patrick Ferber, Florian Geißer, Felipe Trevizan, Malte Helmert, and Jörg Hoffmann. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling*, volume 32, pp. 583–587, 2022. doi: 10.1609/icaps.v32i1.19845.

Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003. doi: 10.1613/jair.1129.

Markus Fritzsche, Daniel Gnad, Mikhail Gruntov, and Alexander Shleyfman. Managing infinite abstractions in numeric pattern database heuristics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 36227–36235, 2026. doi: 10.1609/aaai.v40i43.40941.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

Daniel Gnad, Lee-Or Alon, Eyal Weiss, and Alexander Shleyfman. PDBs go numeric: Pattern-database heuristics for simple numeric planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26507–26515, 2025. doi: 10.1609/aaai.v39i25.34851.

Kai Goebel and Patrik Zips. Can LLM-reasoning models replace classical planning? a benchmark study. *arXiv preprint arXiv:2507.23589*, 2025. doi: 10.48550/arXiv.2507.23589.

Gonzalo Gonzalez-Pumariega, Wayne Chen, Kushal Kedia, and Sanjiban Choudhury. Query-efficient planning with language models. *arXiv preprint arXiv:2412.06162*, 2024. doi: 10.48550/arXiv.2412.06162. URL https://arxiv.org/abs/2412.06162.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/f9f54762cbb4fe4dbffdd4f792c31221-Abstract.html.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023. doi: 10.18653/v1/2023.emnlp-main.507.

Rishi Hazra, Pedro Zuidberg Dos Martires, and Luc De Raedt. SayCanPay: Heuristic planning with large language models using learnable domain knowledge. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 20123–20133, 2024. doi: 10.1609/aaai.v38i18.29991.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006. doi: 10.1613/jair.1705.

Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003. doi: 10.1613/jair.1144.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001. doi: 10.1613/jair.855.

Mengkang Hu, Yao Mu, Xinmiao Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. Tree-planner: Efficient close-loop task planning with large language models. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/d53538ba21c05fa361d2b21704172753-Abstract-Conference.html.

Sukai Huang, Trevor Cohn, and Nir Lipovetzky. Chasing progress, not perfection: Revisiting strategies for end-to-end LLM plan generation. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling*, volume 35, pp. 204–212, 2025. doi: 10.1609/icaps.v35i1.36119.

IPC 2026 Numeric Track Organizers. International planning competition 2026: Numeric tracks, 2026. URL https://ipc2026-numeric.github.io/. Competition results.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B. Murthy. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 22895–22907, 2024. URL https://proceedings.mlr.press/v235/kambhampati24a.html.

Ryo Kuroiwa, Alexander Shleyfman, Chiara Piacentini, Margarita P. Castro, and J. Christopher Beck. LM-cut and operator counting heuristics for optimal numeric planning with simple conditions. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling*, volume 31, pp. 210–218, 2021. doi: 10.1609/icaps.v31i1.15964.

Ryo Kuroiwa, Alexander Shleyfman, and J. Christopher Beck. LM-cut heuristics for optimal linear numeric planning. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling*, volume 32, pp. 203–212, 2022. doi: 10.1609/icaps.v32i1.19803.

Alessandro La Farciola, Alessandro Valentini, and Andrea Micheli. Automatic selection of macro-events for heuristic-search temporal planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26579–26586, 2025. doi: 10.1609/aaai.v39i25.34859.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023. doi: 10.48550/arXiv.2304.11477.

Nicholas Massad, Amine Trabelsi, François Ferland, and Froduald Kabanza. VOLTS: Validated output through logit tree search for reliable PDDL planning with small language models. In *Proceedings of the 39th Canadian Conference on Artificial Intelligence*, volume 318 of *Proceedings of Machine Learning Research*, pp. 872–879. PMLR, 2026. URL https://proceedings.mlr.press/v318/massad26a.html.

Moonshot AI. Kimi K2.6: Model card, 2026. URL https://huggingface.co/moonshotai/Kimi-K2.6.

Qwen Team. Qwen3.5-9B, 2026. URL https://huggingface.co/Qwen/Qwen3.5-9B. Model card.

Nicholas Rossetti, Massimiliano Tummolo, Alfonso Emilio Gerevini, Luca Putelli, Ivan Serina, Mattia Chiari, and Matteo Olivato. Learning general policies for planning through GPT models. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling*, volume 34, pp. 500–508, 2024. doi: 10.1609/icaps.v34i1.31510.

Enrico Scala and Luigi Bonassi. On using lazy greedy best-first search with subgoaling relaxation in numeric planning problems. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling*, volume 35, pp. 245–249, 2025. doi: 10.1609/icaps.v35i1.36125.

Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research*, 68:691–752, 2020a. doi: 10.1613/JAIR.1.11875.

Enrico Scala, Alessandro Saetti, Ivan Serina, and Alfonso E. Gerevini. Search-guidance mechanisms for numeric planning through subgoaling relaxation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pp. 226–234, 2020b. doi: 10.1609/icaps.v30i1.6665.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. doi: 10.48550/arXiv.2402.03300.

Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDL planning with pretrained large language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. URL https://openreview.net/forum?id=1QMMUB4zfl.

Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 international planning competition. *AI Magazine*, 45(2):280–296, 2024. doi: 10.1002/aaai.12169.

Pulkit Verma, Ngoc La, Anthony Favier, Swaroop Mishra, and Julie A. Shah. Teaching LLMs to plan: Logical chain-of-thought instruction tuning for symbolic planning. *arXiv preprint arXiv:2509.13351*, 2025. doi: 10.48550/arXiv.2509.13351.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. doi: 10.48550/arXiv.2503.14476.

Z.ai. GLM-4.7: Model card, 2025. URL https://huggingface.co/zai-org/GLM-4.7.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A RUNTIME ALGORITHM AND ASYNCHRONOUS CONTROL

Algorithm 1 specifies control flow within one native search phase, with a budget of B state-level requests. The defaults are $K = 3$ generations per request and a shared allowance of $H = 100$ integrated transitions. Each request record r stores the query state $s_q(r)$, a unique run–phase identifier, and a remaining allowance rem_r, initialized to H.

Algorithm 1 TRACS within one native search phase.

Require: Problem $\mathcal{P}$; native phase $\mathcal{A}$; bounds B, H; samples K
Initialize $\mathcal{A}$ on $\mathcal{P}$ with its native search settings
Initialize monitor M, pending requests P, and adapter queue Q
$e \leftarrow$ current run–phase identifier; $n \leftarrow 0$; $b \leftarrow 0$
Attach M to native heuristic-evaluation and frontier-update events
while no native stopping limit has been reached **do**
  for all $(r, y) \in$ POLLREADY(P) **do** ▷ samples from ready requests; no wait
    if $r.\mathrm{phase} \neq e$ or $\mathrm{rem}_r = 0$ **then**
      continue
    end if
    $\pi \leftarrow$ VALIDPREFIX$(s_q(r), y)$ ▷ validate from the recorded state
    $m \leftarrow$ ADMITBOUNDED$(\mathcal{A}, Q, \pi, \mathrm{rem}_r)$
    $\mathrm{rem}_r \leftarrow \mathrm{rem}_r - m$ ▷ shared by all K generations
  end for
  $(w, o) \leftarrow$ SELECTNATIVEORGUIDED$(\mathcal{A}, Q)$
  if $w = \perp$ **then** ▷ both native Open and Q are exhausted
    if $P = \emptyset$ **then**
      break
    end if
    WAITFORREPLYORLIMIT(P); **continue**
  end if
  $x \leftarrow$ NATIVEMATERIALIZEANDEVALUATE(w)
  if x is rejected by native duplicate, bound, or dead-end checks **then**
    continue
  end if
  if $x \models G$ **then**
    return NATIVERECONSTRUCTPLAN(x)
  end if
  $D \leftarrow$ NATIVEEXPAND(x) ▷ close x; use native successor rules
  if DETECTORELIGIBLE(o) **then** ▷ exclude LLM-origin rollout steps
    $n \leftarrow n + 1$; OBSERVEEXPANSION(M, x, D, n)
    UPDATEPLATEAUWINDOWSIFDUE(M, n)
    $q \leftarrow$ SELECTQUERY(M, x, n) ▷ plateau $>$ global $>$ branch
    if $q \neq \perp$, $b < B$, and REQUESTPERMITTED(M, P, q, n) **then**
      Create record r with $r.\mathrm{phase} = e$, $s_q(r) = q$, $\mathrm{rem}_r = H$
      $p_q \leftarrow$ RESIDUALPROMPT$(\mathcal{P}, q)$
      LAUNCHASYNC(P, r, p_q, K) ▷ same prompt, independent samples
      $b \leftarrow b + 1$; mark q requested and start the shared cooldown
    end if
  end if
end while
return the native phase outcome at exhaustion or its stopping limit

A.1 ALGORITHM CONVENTIONS AND TRIGGER BOOKKEEPING

NATIVEMATERIALIZEANDEVALUATE uses an already evaluated state in eager search and materializes a selected edge in lazy search. SELECTNATIVEORGUIDED uses ordinary Open ordering in eager search (Q is empty), and gives the finite queue Q priority in lazy search. The origin tag o distinguishes ordinary search from LLM-origin rollout steps; n counts ordinary base-search expansions, and b counts issued requests.

The detectors update their statistics using information from state insertions, expansions, and heuristic evaluations during native search. The monitor receives heuristic evaluations from both ordinary search and LLM-guided steps, while only ordinary expansions advance n. Eager plateau counters

occupy a fixed-capacity table indexed by rounded h and are updated on state insertion and expansion; lazy plateau counters update only on ordinary expansions. Plateau conditions are checked at the end of each counting window.

SELECTQUERY applies the trigger criteria and priority order in Section 4.1; REQUESTPERMITTED checks cooldown, query-state deduplication, and the pending-request cap. Appendix C.2.2 gives the window sizes, check intervals, bucket-selection settings, and reset rules.

A.2 ASYNCHRONOUS MULTI-SAMPLE REQUESTS

LAUNCHASYNC serializes the query state once for the K generations. They return as one batch after all attempts finish, fail, or time out. At the start of each search iteration, the planner checks whether any LLM responses have arrived. It validates each returned action sequence from its original query state. If no responses are available, native search proceeds without waiting. Responses whose run–phase identifier no longer matches the current phase are discarded.

If both native Open and the adapter queue are empty but requests remain pending, the phase waits for a reply or a stopping limit; exhaustion is declared only when no pending requests remain. Returned plans and subsequent phases follow the outer anytime schedule in Appendix C.2.1.

Beyond the action-validity checks in Section 4.2, VALIDPREFIX stops before the first canonical state revisit within a candidate. Appendix C.1.2 defines this state identity. A state previously seen elsewhere in native search does not by itself terminate replay; native duplicate handling determines its admission. Any nonempty retained prefix may be offered to the adapter, with no additional length threshold. Empty responses, transport failures, and candidates with no valid first action produce no prefix for integration.

ADMITBOUNDED returns the number of transitions charged to the request, $0 \leq m \leq \mathrm{rem}_r$, and the algorithm subtracts m from the shared allowance. Eager integration charges admitted transitions; lazy integration reserves allowance when transitions are scheduled in Q, so queued work already counts toward the request's limit. The allowance is not reset between samples.

B PROMPT AND TRAINING-EXAMPLE FORMAT

The following DEPOTS-NUM training example includes the full domain documentation, residual problem, reasoning trace, and solver-verified action sequence. Training and runtime prompts use this input format; the response delimits its reasoning with `<think>` and `</think>`.

B.1 DOMAIN DOCUMENTATION

The domain description first defines the object types and state-query API. Crates are both movable cargo and surfaces on which other crates can be placed.

Entity hierarchy.

```
Place
  Depot, Distributor
Locatable
  Truck, Hoist
  Surface
    Pallet, Crate
```

Boolean state queries.

```
is_at(obj: Locatable, p: Place)  # object location
is_on(c: Crate, s: Surface)     # crate on a surface
is_in(c: Crate, t: Truck)       # crate inside a truck
is_clear(s: Surface)           # no crate on the surface
is_available(h: Hoist)         # hoist is free
is_lifting(h: Hoist, c: Crate)  # hoist holds the crate
```

Numeric state queries (float).

```
get_weight(c: Crate)            # crate weight
get_current_load(t: Truck)      # total truck load
get_load_limit(t: Truck)        # truck capacity
get_fuel_cost()                 # accumulated fuel consumption
```

The five action definitions below preserve the logical and numeric rules in the domain file. `Pre` lists required predicates; `Add` and `Del` list predicates made true and false. Numeric guards and updates are written explicitly. Unmentioned state components remain unchanged.

Drive: move a truck and its cargo between places.

```
action_Drive(x: Truck, y: Place, z: Place)
Pre:     is_at(x, y)
Add:     is_at(x, z)
Del:     is_at(x, y)
Numeric: get_fuel_cost() += 10
```

Lift: pick up a clear crate from a surface.

```
action_Lift(x: Hoist, y: Crate, z: Surface, p: Place)
Pre:     is_at(x, p), is_at(y, p), is_available(x),
         is_on(y, z), is_clear(y)
Add:     is_lifting(x, y), is_clear(z)
Del:     is_available(x), is_on(y, z), is_at(y, p), is_clear(y)
Numeric: get_fuel_cost() += 1
```

Drop: place a held crate on a clear surface.

```
action_Drop(x: Hoist, y: Crate, z: Surface, p: Place)
Pre:     is_at(x, p), is_at(z, p), is_lifting(x, y), is_clear(z)
Add:     is_available(x), is_on(y, z), is_at(y, p), is_clear(y)
Del:     is_lifting(x, y), is_clear(z)
```

Load: put a held crate into a truck with sufficient capacity.

```
action_Load(x: Hoist, y: Crate, z: Truck, p: Place)
Pre:     is_at(x, p), is_at(z, p), is_lifting(x, y)
Guard:   get_current_load(z) + get_weight(y) <= get_load_limit(z)
Add:     is_available(x), is_in(y, z)
Del:     is_lifting(x, y)
Numeric: get_current_load(z) += get_weight(y)
```

Unload: take a crate from a truck using a free hoist.

```
action_Unload(x: Hoist, y: Crate, z: Truck, p: Place)
Pre:     is_at(x, p), is_at(z, p), is_available(x), is_in(y, z)
Add:     is_lifting(x, y)
Del:     is_available(x), is_in(y, z)
Numeric: get_current_load(z) -= get_weight(y)
```

B.2 Serialized Residual Problem

The problem file gives the following objects, numerical values, true logical predicates, and seven goals. No objects or goals are omitted. The supplied state already has crates in a truck and a hoist holding a crate, illustrating why the proposer must reason from the current residual state.

Objects and current numerical state.

```
Crates: crate0, crate1, crate2, crate3, crate4, crate5, crate6, crate7
Depots: depot0
Distributors: distributor0, distributor1
Hoists: hoist0, hoist1, hoist2
Pallets: pallet0, pallet1, pallet2, pallet3, pallet4
Trucks: truck0

get_current_load(truck0) = 533
get_fuel_cost() = 38
get_load_limit(truck0) = 643
get_weight(crate0) = 69
get_weight(crate1) = 61
get_weight(crate2) = 80
```

```
get_weight(crate3) = 39
get_weight(crate4) = 117
get_weight(crate5) = 80
get_weight(crate6) = 157
get_weight(crate7) = 129
```

Current logical state.

```
is_at(crate2, distributor0)
is_at(crate3, depot0)
is_at(hoist0, depot0)
is_at(hoist1, distributor1)
is_at(hoist2, distributor0)
is_at(pallet0, depot0)
is_at(pallet1, distributor1)
is_at(pallet2, depot0)
is_at(pallet3, depot0)
is_at(pallet4, distributor0)
is_at(truck0, depot0)
is_available(hoist0)
is_available(hoist1)
is_clear(crate2)
is_clear(crate3)
is_clear(pallet0)
is_clear(pallet1)
is_clear(pallet3)
is_in(crate0, truck0)
is_in(crate1, truck0)
is_in(crate4, truck0)
is_in(crate6, truck0)
is_in(crate7, truck0)
is_lifting(hoist2, crate5)
is_on(crate2, pallet4)
is_on(crate3, pallet2)
```

Goals.

```
is_on(crate2, pallet0)
is_on(crate5, crate2)
is_on(crate0, pallet1)
is_on(crate3, pallet2)
is_on(crate4, crate3)
is_on(crate6, crate4)
is_on(crate7, pallet4)
```

B.3 Reasoning Trace and Action Sequence

The response combines a reasoning trace with a Python code block of grounded action calls. The trace below explains the accompanying sequence, including the capacity calculation, preservation of an existing goal, and ordering of actions at different locations.

Reasoning trace.

<think>

1. Identify current progress and the remaining goals. This is not an untouched logistics problem. `truck0` already contains five crates with a total load of 533, and `hoist2` is holding `crate5` at `distributor0`. First compare the current state against every goal to avoid repeating work that is already complete.

1. `is_on(crate2, pallet0)` is not satisfied: `crate2` is still on `pallet4` at `distributor0`.
2. `is_on(crate5, crate2)` is not satisfied: `hoist2` is holding `crate5` at `distributor0`.
3. `is_on(crate0, pallet1)` is not satisfied: `crate0` is inside `truck0`.
4. `is_on(crate3, pallet2)` is already satisfied, and `crate3` is clear.
5. `is_on(crate4, crate3)` is not satisfied: `crate4` is inside `truck0`.
6. `is_on(crate6, crate4)` is not satisfied: `crate6` is inside `truck0`.
7. `is_on(crate7, pallet4)` is not satisfied: `crate7` is inside `truck0`, and `crate2` still occupies `pallet4`.

Only one goal is currently satisfied. Preserve crate3 on pallet2 and extend that stack to pallet2 <- crate3 <- crate4 <- crate6. The other required structures are pallet0 <- crate2 <- crate5 at depot0, pallet4 <- crate7 at distributor0, and pallet1 <- crate0 at distributor1.

2. Analyze locations and transport directions. The goals involve three places. crate2 and crate5 are at distributor0 and must reach depot0. crate4 and crate6 are already in the truck, which starts at depot0; they can be unloaded and stacked after its return. crate7 must be left on pallet4 at distributor0, while crate0 can remain in the truck until the final delivery to distributor1.

A suitable route is depot0 -> distributor0 -> depot0 -> distributor1. At distributor0, exchange cargo by unloading crate7 and collecting crate2 and crate5. After returning to depot0, complete the two local stacks, then deliver crate0 to distributor1.

3. Resolve the capacity constraint before departure. The truck's load is 533 and its capacity is 643, leaving only 110. Both crate5 and crate2 weigh 80, so loading both without first unloading cargo would give $533 + 80 + 80 = 693$, exceeding the limit.

crate1 weighs 61 and belongs to no goal structure. Unload it at depot0 with the available hoist0, reducing the load to $533 - 61 = 472$. The hoist can keep holding crate1 while the truck travels to distributor0. After the truck returns, place crate1 on the currently clear pallet3 to free the hoist.

4. Exchange cargo at distributor0. On arrival, hoist2 is holding crate5. Load that crate first: the truck's load rises from 472 to 552, still within capacity, and the hoist becomes available.

pallet4, the destination for crate7, is occupied by crate2. Lift crate2 from pallet4, making the pallet clear, and load the crate. The new load is $552 + 80 = 632 \leq 643$: only 11 units of capacity remain, but the limit is respected.

Unload crate7 with the now-available hoist, reducing the load to $632 - 129 = 503$. Once the hoist holds crate7, placing it no longer requires the truck to be present. The truck can return to depot0 with crate2, crate5, and the remaining cargo; hoist2 can subsequently place crate7 on the cleared pallet4 to complete the local goal.

5. Preserve the existing goal and build both depot0 stacks. After the truck returns, hoist0 is still holding crate1. Place it on the clear pallet3, freeing the hoist without affecting any goal.

Build the stack above crate3. Its goal relation with pallet2 already holds, so leave crate3 in place. Because it is clear, unload crate4 and place it directly on crate3. The newly placed crate4 is clear; unload crate6 and place it on crate4. This preserves the initially satisfied goal while completing the taller stack.

For the other stack, unload crate2 and place it on the clear pallet0. Then unload crate5 and place it on crate2, which is now a clear supporting surface.

After these unloads, only crate0, weighing 69, remains in the truck. It can now travel to distributor1. Once hoist0 has unloaded crate5, the truck's departure and the hoist's final placement are independent: the truck can move first, and the hoist can finish placing crate5 at depot0.

6. Complete the final goal at distributor1. hoist1 has been available since the initial state, and pallet1 remains clear. When the truck arrives, unload crate0 with hoist1 and place it on pallet1.

Check the final state: crate3 remains on pallet2, with crate4 and crate6 stacked above it; crate2 and crate5 form the stack on pallet0; crate7 is on pallet4; and crate0 is on pallet1. All seven goals hold.

Three Drive actions increase fuel consumption from 38 to 68. Lifting crate2 from pallet4 adds one more unit, giving a final value of 69. The maximum load during the loading stages is 632, below the capacity of 643.

</think>

Original action sequence.

```python
action_Unload(hoist0, crate1, truck0, depot0)
action_Drive(truck0, depot0, distributor0)
action_Load(hoist2, crate5, truck0, distributor0)
action_Lift(hoist2, crate2, pallet4, distributor0)
action_Load(hoist2, crate2, truck0, distributor0)
action_Unload(hoist2, crate7, truck0, distributor0)
action_Drive(truck0, distributor0, depot0)
action_Drop(hoist0, crate1, pallet3, depot0)
action_Drop(hoist2, crate7, pallet4, distributor0)
action_Unload(hoist0, crate4, truck0, depot0)
action_Drop(hoist0, crate4, crate3, depot0)
action_Unload(hoist0, crate6, truck0, depot0)
action_Drop(hoist0, crate6, crate4, depot0)
action_Unload(hoist0, crate2, truck0, depot0)
action_Drop(hoist0, crate2, pallet0, depot0)
action_Unload(hoist0, crate5, truck0, depot0)
action_Drive(truck0, depot0, distributor1)
action_Drop(hoist0, crate5, crate2, depot0)
action_Unload(hoist1, crate0, truck0, distributor1)
action_Drop(hoist1, crate0, pallet1, distributor1)
```

Replaying this 20-action sequence satisfies all seven goals, with final truck load zero, peak load 632, and accumulated fuel cost 69. Runtime validation uses the action block, following Appendix A.2; the reasoning text does not establish validity.

C Experimental Configuration and Supplementary Results

C.1 Training and Reward Details

C.1.1 Data and Optimization

For the residual-task construction in Section 4.4, we set $p = 3$. For a solution of length n, consecutive cuts and trajectory boundaries are separated by at least $\max(10, \lceil 0.15n \rceil)$ actions when the length permits. Shorter solutions receive one interior cut when available.

Both training stages update all Qwen3.5-9B parameters with the settings in Table 4. DAPO's dynamic resampling is capped at ten generation batches.

Table 4: Proposer-training hyperparameters. Batch denotes prompts per optimizer update.

Setting	SFT	DAPO
Training epochs	3	1
Learning rate	2×10^{-6}	1×10^{-6}
Batch / rollouts per prompt	16 / –	8 / 8
Maximum lengths	16,384 total	16,384 prompt + 16,384 response
Sampling (T, p)	–	$(1.0, 1.0)$
PPO epochs	–	1
Policy-ratio clip widths (ℓ, u)	–	$(0.20, 0.28)$
Dual-clipping constant c	–	10
KL penalty coefficient	–	0

The policy-ratio clipping interval is $[1 - \ell, 1 + u]$; c is the dual-clipping constant.

For SFT, we set $\lambda = 2$ in Equation 4. Critical token positions are identified by overlap with numerical object identifiers or arguments in the action block. The aligned weight vectors are computed offline and loaded directly during training.

C.1.2 State Identity and Effective Length

A canonical state ID is the identifier assigned by the native state registry within one preprocessed task. Equality is determined by the packed propositional values and retained regular numeric fluents. Static constants are shared across states, and derived numeric expressions are recomputed. Variables used only for objective accumulation are stored separately from the identity key, together with the planner's path-cost bookkeeping. IDs are compared only within the same task and registry.

In our tasks, native preprocessing classifies `fuel_cost` in DEPOTS-NUM and `total-fuel-used` in ZENOTRAVEL-NUM as such objective-only variables. Their accumulated values do not affect action applicability, feasibility-relevant effects, or goal satisfaction. They therefore remain available for cost evaluation without distinguishing otherwise identical search states. Dynamic quantities that constrain future actions, including `current_load`, remaining `fuel`, and `onboard`, remain in the state representation when relevant to the compiled task. TRACS reuses this native distinction; a variable that affects feasibility cannot be omitted merely because it also appears in the objective. Thus a state-ID revisit denotes a return to the same native search state, potentially with a different accumulated cost. With nonnegative additive costs, such revisits indicate redundancy in the action sequence and may reflect inefficient back-and-forth exploration or attempts to inflate rewards through repeated actions.

Effective length and repeated actions. For the executable actions $(a_1, \ldots, a_L)$ and replayed states $(s_0, \ldots, s_L)$, a repeated canonical state ID at positions $i < j$ identifies a cycle covering action positions $i + 1, \ldots, j$. Let $\mathcal{A}_{\text{loop}}$ be the union of all such positions. Let $\mathcal{A}_{\text{rep}}$ contain every action position whose grounded call, including its arguments, has appeared earlier in the sequence. These later occurrences are excluded even without a state revisit. Training and offline evaluation both use

$$L_{\text{eff}} = L - |\mathcal{A}_{\text{loop}} \cup \mathcal{A}_{\text{rep}}| \,. \tag{8}$$

The union ensures that each position is subtracted only once. During training and offline evaluation, replay continues after state revisits, so subsequent executable actions can still contribute to L_{eff}.

State cycles can contain distinct grounded calls, while a repeated call can be necessary in a different state without forming a cycle. Excluding repeated calls is a conservative progress proxy: we assume necessary exact repetitions are relatively infrequent and accept some under-crediting to prevent repeated calls from inflating length rewards. This discount neither invalidates an applicable action nor prohibits its use by native search. Only state-cycle positions incur P_{loop}; repeated calls alone do not.

C.1.3 Reward Constants

For Equations 5 and 6, we use $L_{\min} = 15$, $L_{\text{cap}} = 40$, $w_{\text{len}} = 4$, $w_{\text{best}} = 2$, $w_{\text{final}} = 1$, $\epsilon_h = 1$, $r_{\text{floor}} = -4$, and $r_{\text{fail}} = -5$. The remaining terms are

$$\begin{aligned} R_{\text{goal}}(C) &= 10 + \min\left(2, \frac{20}{\max(C, 1)}\right), \\ P_{\text{loop}} &= \mathbb{I}[|\mathcal{A}_{\text{loop}}| > 0] + 0.25|\mathcal{A}_{\text{loop}}|, \\ P_{\text{invalid}} &= 0.5\,\mathbb{I}[\text{invalid termination}]. \end{aligned} \tag{9}$$

Non-goal responses meeting $L_{\min}$ score in $[-4, 7]$, above $r_{\text{fail}} = -5$ and below the goal reward in $(10, 12]$.

All heuristic values used by the DAPO reward are computed with NLM-CutPlan's numeric LM-cut heuristic.

C.1.4 Legacy GRPO Control

The Legacy GRPO checkpoint in Table 3 comes from an earlier recipe that already rewarded executable progress, but treated complete-plan generation as the training objective. For a generated trajectory π ending in state s_n, its per-trajectory reward was

$$R_{\text{old}}(\pi) = R^{\text{old}}_{\text{goal}} + R^{\text{old}}_{\text{step}} - P^{\text{old}}_{\text{error}} + R^{\text{old}}_{\text{eff}} \,, \tag{10}$$

where $R^{\text{old}}_{\text{goal}}$ gives a dominant reward when $s_n \models G$ and penalizes a valid but incomplete plan. The dense term was

$$R^{\text{old}}_{\text{step}} = \min\left(R_{\text{cap}}, \sum_{i=1}^{n} \beta\, \mathbb{I}[a_i \text{ is applicable}]\, \Omega(a_i)\right), \tag{11}$$

with $\Omega(a_i) = \mathbb{I}[a_i \notin \{a_1, \ldots, a_{i-1}\}]$. Thus the control rewarded previously unseen grounded action calls when applicable, up to a reward cap. The remaining terms penalized action failures and gave successful plans an inverse-cost bonus proportional to $1/C(\pi)$.

Example: legal actions that inflate reward. In a non-goal DEPOTS-NUM state, lift and replace each of three clear crates on its original support using available hoists. All six grounded calls can be distinct and executable, yet each pair restores the same canonical state while adding cost. The legacy step term alone can credit all six; the current rule marks all six positions as cyclic, yielding $L_{\text{eff}} = 0$ and r_{fail}. In a longer partial response that still meets $L_{\min}$, cyclic positions also incur P_{loop}.

Complete-solution group dropout. To keep optimization focused on complete solutions, the legacy recipe applied group dropout after computing the trajectory-specific rewards. This was intended to discourage convergence to executable but consistently incomplete outputs, or "graceful failures." For a group of K_{train} trajectories, distinct from the runtime sampling count K, let $g_i = \mathbb{I}[s^{(i)}_{L_i} \models G]$. If no response reached the goal, the rule replaced every reward in the group by the minimum score $r_{\text{drop}} = -5$, regardless of partial progress. Consequently,

$$\bar{R} = r_{\text{drop}}, \qquad \sigma_R = 0, \qquad A_i = \frac{R_i - \bar{R}}{\sigma_R + \epsilon} = 0 \quad \forall i. \tag{12}$$

Current group criterion. We now train the model as a search proposer, for which both complete solutions and usable partial continuations are useful outputs. A group receives the uniform minimum score only when every response misses the goal and falls below $L_{\min}$ effective actions. This follows directly from the failure branch of Equation 5, without an additional group-level override. A goal-reaching response receives $R_{\text{goal}}(C)$ regardless of length. The two criteria can be written as

$$\begin{aligned} u_i &= \mathbb{I}\left[g_i = 1 \;\vee\; L^{(i)}_{\text{eff}} \geq L_{\min}\right], \\ D_{\text{old}} &= \mathbb{I}\left[\sum_{i=1}^{K_{\text{train}}} g_i = 0\right], \qquad D_{\text{cur}} = \mathbb{I}\left[\sum_{i=1}^{K_{\text{train}}} u_i = 0\right]. \end{aligned} \tag{13}$$

For the same sampled group, $g_i \leq u_i$ implies $D_{\text{cur}} \leq D_{\text{old}}$. In particular, a group with no complete solution but at least one sufficiently long effective prefix is no longer forced to a uniform failure score.

Example: different progress, identical rewards. Four unsuccessful responses with effective lengths 0, 0, 20, and 30 all receive -5 under the legacy override, giving zero relative advantages. Under the current reward, the last two meet $L_{\min} = 15$ and score above failures; without penalties or heuristic bonuses, their length rewards also distinguish them from each other.

Hard problems can therefore supply learning signals through differences in usable progress before the model can reliably solve them. This can improve sample efficiency and reduce training's bias toward easy, already-solvable problems. Other zero-variance groups also remain possible, including equal rewards among usable responses; DAPO's dynamic sampling filters these groups. The Legacy GRPO control compares the combined reward and optimization recipes, without isolating the optimizer change.

C.2 PLANNER AND RUNTIME CONFIGURATION

C.2.1 NATIVE PLANNERS AND DEPLOYMENT

Table 5 specifies the anytime schedules used by each backbone and its TRACS variant. Thread allocations are reported in Table 6; search time limits are given in Section 5.1.

Table 5: Native anytime configurations. The incumbent bound is passed from one phase to the next in both backbones.

Backbone	Heuristic	Search schedule
Eager: NLM-CutPlan	Numeric LM-cut with second-order simplification, 10 bound iterations, and sub-unit ceilings	Eager best-first search, tie-broken by $(h_{\mathrm{NLM}}, h_{\mathrm{goal}})$; closed states are not reopened; the final phase is repeated
Lazy: Count Downward	Interval-relaxed h_{FF} with unit-cost heuristic evaluation	Lazy greedy search with unit and native costs, followed by lazy weighted A* with $w \in \{5, 3, 2, 1\}$; preferred operators use h_{FF}

Table 6 lists the reported resource profiles. Earlier Depots experiments used varying containers; exact CPU models and vCPU quotas were not retained in their archived run metadata. The CPU entries thus describe reported profiles, not verified allocations for every historical run. Saved scheduling settings are detailed below, and Appendix C.3.3 examines actual search throughput. API model inference is hosted by the providers.

Table 6: **Reported resource profiles.** CPU profiles are not uniform across all archived runs. vCPU counts describe host allocations; native search thread counts are per solver instance. Concurrent jobs share host resources. Host memory capacity was sufficient for the experiments.

Resource	Native baselines	TRACS
CPU model	Intel Xeon Platinum 8352V	Intel Xeon Platinum 8358
Reported CPU frequency	2.10 GHz	2.60 GHz
Allocated vCPUs	32	16
Native search threads	Eager: 18; lazy: 1	Eager: 18; lazy: 1
GPU model / count	No GPU computation	1× A800 (80 GB)
GPU sharing across jobs	Not applicable	Shared inference service
Inference concurrency limit	Not applicable	100 sampled sequences

The default scheduling caps are eight concurrent instances at scales 10–30 and two at scale 40. The archived Depots Count Downward baseline instead records four concurrent instances at scales 10–30; its scale-40 concurrency was not retained. Both eager deployments record `OMP_NUM_THREADS=18`; container-visible CPU counts do not specify the allocated vCPU quota.

Concurrent TRACS solvers submit requests as needed to a single shared vLLM endpoint hosting the proposer under evaluation. The service is configured to run at most 100 sampled sequences concurrently. Excess demand would be queued, but no queueing due to this limit was observed during the experiments. During solver runs, the GPU is dedicated to this inference service, with no other GPU workloads. GPU utilization and GPU time were not separately measured. No GPU out-of-memory errors were observed on the 80 GB device.

Timing and resource scope. The wall-clock timer starts with native anytime search and spans all phases. Translation and model/service startup are excluded; waiting, replay, and reinsertion during search consume the same allowance. These timing rules apply to all wall-clock cutoffs; Appendix C.3 explains the two evaluation protocols and their interpretation.

C.2.2 Intervention and Inference Settings

Table 7 lists the principal detector values at scales 10 and 20. The eager scale-40 profile halves expansion-count intervals and minimum-count thresholds. For lazy search, the plateau window stays fixed, while stall, cooldown, and branch-check intervals are multiplied by 0.5 at scale 30 and 0.25 at scale 40. Other criteria are unchanged.

Table 7: Principal trigger settings before scale-dependent adjustments. Expansion intervals exclude LLM-origin rollout steps; confirmation/reset values count windows, and ancestor depth counts parent links.

Setting	Eager	Lazy
Plateau analysis/window	65,536	65,536
Confirmation windows	3	2
Reset windows	4	2
Plateau qualification	4,096 expanded; 1,024 net growth; ratio 1.05	16,384 expanded; share 0.25
Global-stall interval	500,000	500,000
Branch-check interval	100,000	100,000
Ancestor depth	20	20
Shared request gap	100,000	500,000

Reset windows determine when a bucket's plateau episode ends. Eager search resets the bucket's plateau history and activation state after four consecutive nonqualifying windows, or when a nonqualifying bucket has received no newly opened states for four consecutive windows. The latter case also clears the per-bucket request cooldown; the former retains it. Lazy search deactivates an active bucket after two consecutive nonqualifying windows and clears its request eligibility and since-request expansion count. Both detectors require fresh confirmation before reactivation.

Each native phase permits at most $B = 10$ state-level requests, except that TRACS–Lazy at scale 40 permits $B = 15$. By default, each request produces three independent samples with temperature 0.7, top-p 0.9, and a 16,384-token response limit. At an eager plateau opportunity, TRACS selects among at most three candidate buckets; the lazy detector selects the busiest confirmed qualifying bucket.

C.2.3 End-to-End API Baselines

Kimi K2.6 uses thinking mode, $(T, p) = (1.0, 0.95)$, and a 32,768-token completion limit. DeepSeek-V4.1-Flash uses its low-reasoning setting, provider-default sampling, and a 32,768-token completion limit. GLM-4.7 uses thinking mode, $(T, p) = (1.0, 0.95)$, and the same 32,768-token limit.

Transport, rate-limit, and empty-response failures are rerun rather than interpreted as planning failures. A generation timeout counts as a failed attempt, as does a returned completion that is unparsable, inapplicable, or does not reach the goal. Provider usage schemas do not consistently separate hidden reasoning from answer tokens, so the configured limits and provider-reported completion tokens are not treated as a common measure of FLOPs. The raw usage and response artifacts are retained for reproducibility.

C.3 Evaluation Metrics and Statistical Protocols

Rationale for the evaluation protocols. We use two complementary evaluation cutoffs. Wall-clock cutoffs directly measure the coverage and plan quality achieved within a given time allowance, but controlled comparisons require comparable hardware and resource allocations. Native baselines and TRACS use different CPU configurations, and TRACS additionally uses GPU inference (Appendix C.2.1). The Depots throughput diagnostics in Appendix C.3.3 show that TRACS does not have higher aggregate native-search throughput, although they do not isolate hardware effects.

Paired expansion cutoffs reduce dependence on execution speed by comparing search progress over a common range of native expansions. Within this range, we report coverage and the expansions required to reach matched plan quality. These comparisons are restricted to the same backbone because eager and lazy search perform different work per expansion. Expansion counts exclude inference, replay, and reinsertion overhead; asynchronous response timing can also affect the search trajectory. Thus, wall-clock results characterize performance under the reported deployments, while paired expansion results quantify changes in native search expansion requirements. Together, they

assess practical outcomes and search behavior without establishing equal-compute efficiency or hardware-independent speedup.

Cutoff definitions and use. All solver runs are subject to the scale-specific wall-clock limit T_i in Section 5.1. Wall-clock evaluation uses the best plan available by T_i. A paired expansion cutoff C_i is the smaller total expansion count recorded by two runs of the same backbone. It is applied to their logs after execution, rather than used as a separate stopping rule, and keeps both observations within their recorded expansion ranges. The cutoff is problem- and pair-specific, so it does not define a common expansion budget across configurations. In particular, the same baseline can have different coverage when paired with different TRACS runs.

The table below lists the metric columns from each results table in the main text and appendices, grouped by the evaluation cutoff used to compute them. Request, validation, and admission diagnostics are described separately in Appendix C.5; API and offline proposer evaluations retain their attempt budgets.

Results table	Wall-clock cutoff	Paired expansion cutoff
Table 1 (solvers)	Q; solved count	–
Table 2	–	Matched-quality expansion reduction
Table 3(b)	Coverage; Q	Expansion reduction
Table 9	–	Cost W/T/L
Table 10	Q	Coverage; expansion reduction
Table 11	–	Anytime quality over the paired interval

C.3.1 Terminal Quality and Coverage

Reference pool. Let $\mathcal{M}$ contain the four solver configurations in Table 1: NLM-CutPlan, TRACS–Eager, Count Downward, and TRACS–Lazy. Let $c_i^m(t)$ be the lowest native plan cost found by solver m by time t, or $+\infty$ if it has found no valid plan. An incumbent from a run that terminates normally before T_i remains available; plans found after the deadline are excluded. Set

$$c_i^{\mathrm{ref}} = \min_{m \in \mathcal{M}} c_i^m(T_i). \tag{14}$$

This is the best observed cost at the shared deadline, not a known optimum. All four solvers use the same reference pool and test instances. API attempts, other checkpoints, and plans found beyond T_i do not enter this pool.

Quality and coverage. Using the cost-ratio form of IPC satisficing evaluation (Taitler et al., 2024; IPC 2026 Numeric Track Organizers, 2026), define

$$\begin{aligned} q_i^m &= \begin{cases} 0, & c_i^m(T_i) = +\infty, \\ 1, & c_i^m(T_i) = 0, \\ c_i^{\mathrm{ref}} / c_i^m(T_i), & \text{otherwise}, \end{cases} \\ Q_m &= \frac{1}{N} \sum_{i=1}^{N} q_i^m, \qquad S_m = \sum_{i=1}^{N} \mathbb{I}[c_i^m(T_i) < +\infty]\,. \end{aligned} \tag{15}$$

Here N is the number of test problems in the reported group, S_m is the solved count, and coverage is S_m/N. Costs are nonnegative; a valid zero-cost plan receives score one. If no solver succeeds, every score for that problem is zero. All problems receive equal weight, including those solved by only one method; no jointly solved or matched-quality subset is selected.

The score lies in $[0, 1]$. A higher Q need not imply higher coverage or lower cost on every jointly solved problem. It does not reward earlier discovery of the same plan.

Reference pools in supplementary evaluations. Cutoff choice is separate from reference-pool choice. Ablation and repeatability Q use their own pools (Appendices C.4 and C.6); the anytime score in Appendix C.7 uses a two-checkpoint pool.

One interrupted scale-40 Depots TRACS–Eager run is conservatively scored as unsolved ($Q = 0$) in the full test set, despite recorded incumbents, and excluded from paired effort and throughput comparisons. Its recorded query activity is retained.

C.3.2 Matched-Quality Expansion Reduction

Expansion counts include ordinary and LLM-guided native expansions, excluding inference, replay, and heuristic evaluation during reinsertion. They differ from the trigger counters in Section 4.1, which exclude LLM-origin rollout steps.

For each baseline–TRACS pair, let C_i denote its paired expansion cutoff. The target is the baseline's best plan cost by C_i. A problem is matched if both methods reach this cost or better by C_i. Let $E_{\mathrm{base},i}$ and $E_{\mathrm{TRACS},i}$ be their first-hit expansion counts for this target. Over the n matched problems, the percentage reduction is

$$R = 100 \left[1 - \exp\left(\frac{1}{n} \sum_{i=1}^{n} \log \frac{E_{\mathrm{TRACS},i}}{E_{\mathrm{base},i}} \right) \right]. \tag{16}$$

Problems solved by only one method, or where TRACS misses the target cost by the cutoff, do not receive an expansion ratio. They still contribute to coverage at C_i; Table 1 independently scores all problems at T_i. For each confidence interval in Table 2, we draw 2,000 nonparametric bootstrap samples of matched problems, separately for each domain, backbone, and scale, and take the 2.5th and 97.5th percentiles.

C.3.3 Observed Search Throughput

We examine both Depots backbones to relate deployment speed to solver outcomes. For a baseline–TRACS pair, define $\rho_i = (E_i^{\mathrm{base}}/t_i^{\mathrm{base}})/(E_i^{\mathrm{TRACS}}/t_i^{\mathrm{TRACS}})$, where E_i counts native search expansions and t_i is search wall time. Table 8 reports geometric means; $\rho > 1$ indicates higher baseline throughput.

Panel (a) compares the longest initial sequence of completed phases before the first TRACS request whose phase-level bounds, outcomes, plan costs and lengths, and expanded/evaluated/generated/reopened counts match between runs. Each instance contributes one prefix. Minimum durations of 1 and 10 seconds in both runs reduce sensitivity to short-run timing effects. These statistics match recorded work, without asserting identical state-by-state traces.

Panel (b) uses complete pairs at scales 30–40, where both runs reach the shared time limit. All quantities within each row use the same instances; the per-instance reference costs for Q remain those of Appendix C.3.1. The interrupted eager pair is excluded.

Table 8: **Depots search throughput and outcomes.** Arrows show baseline-to-TRACS outcomes. Higher ρ favors baseline expansion throughput; "B higher" counts pairs with $\rho_i > 1$. Panel (b)'s eager scale-40 row uses 12 complete pairs, whereas Table 1 retains all 13 test instances.

(a) Matching search phases before the first request

Backbone	Min. time (s)	n	B higher	ρ
Lazy	1	66	66	1.32
Lazy	10	24	24	1.36
Eager	1	99	96	1.29
Eager	10	54	52	1.34

(b) Full runs on difficult instances

Backbone/scale	n	B higher	ρ	Q	Solved
Lazy/30	50	48	1.39	$0.71 \to 0.93$	$39 \to 50$
Lazy/40	13	13	1.32	$0.25 \to 1.00$	$4 \to 13$
Eager/30	50	49	1.30	$0.41 \to 0.42$	$48 \to 49$
Eager/40	12	12	2.08	$0.44 \to 0.41$	$12 \to 11$

Baseline throughput is higher in aggregate already before query submission. The large lazy gains and modest eager scale-30 gain therefore accompany lower native-search throughput; eager scale 40 still regresses. Full-run rates also reflect different search trajectories and work per expansion. These diagnostics support the interpretation that TRACS's gains are not driven by a hardware advantage in native search.

C.4 Ablation Evaluation Protocols

All local-model evaluations use temperature 0.7, top-p 0.9, and a 16,384-token response limit. These settings apply to runtime inference and the offline evaluations in Table 3(a) and (c).

Training checkpoints. Table 3(a) uses 319 residual problems derived from the DEPOTS-NUM test set solely for inference-time evaluation: 134 Easy problems at scales 10–20 and 185 Hard problems at scales 30–40. Checkpoint step counts refer to optimizer updates. SFT ends at 1,100 updates; DAPO-200steps is intermediate, and DAPO (ours) is the final proposer.

Effective length follows Equation 8. Goal@3 is the fraction of problems solved in at least one of the three attempts. Length thresholds differ across training, offline evaluation, and runtime admission:

- **Training reward:** a non-goal response requires $L_{\text{eff}} \geq L_{\min} = 15$ for partial credit. Goal-reaching responses receive $R_{\text{goal}}(C)$ regardless of length.
- **Offline Usable@3:** the fraction of problems with at least one response that reaches the goal or has $L_{\text{eff}} \geq L_{\min}^{\text{eval}} = 10$.
- **Runtime admission:** one retained action suffices for a prefix to be considered by the adapter; validation and native filtering follow Appendix A.2.

To report mean L_{eff}, we select one response per problem. Goal-reaching responses take priority, with ties resolved by lower plan cost, then fewer executed actions. If no response reaches the goal, we prefer a usable response, then larger L_{eff}, then greater best and final heuristic progress, in that order. Remaining ties in either case favor fewer cyclic actions, then the earlier attempt. The reported value averages the selected responses' effective lengths over all problems. TO and Loop are, respectively, the fractions of all attempt slots that time out or exhibit a canonical state revisit. Timeouts count as failed, unusable, and zero-length; they remain in both denominators and have no observed loop.

Solver-level controls. All four TRACS configurations in Table 3(b) use the same CPU/GPU deployment, concurrency, and wall-clock limits (Appendix C.2.1); Count Downward uses the native-baseline deployment. Window-random selects a position within each eligible trigger window instead of waiting for stagnation. Both the triggering and sample-count controls use the final DAPO proposer.

The SFT-1100steps run uses seed 101, while the DAPO row uses the original run. Their expansion reductions use configuration-specific cutoffs and 37 versus 35 baseline-matched problems, respectively, so this difference does not directly estimate the checkpoint effect. Appendix C.7 instead pairs both checkpoints at seed 101.

For the Usable/call denominator, Window-random and one-sample adaptive triggering issue 849 and 851 requests; three-sample adaptive triggering issues 818 with SFT and 832 with DAPO. Coverage and terminal Q follow Equation 15 at the shared wall-clock cutoffs over all 63 problems. The Count Downward row reports its terminal coverage, independent of the pairings used for expansion reduction. The reference cost for Q is the lowest cost found by any of the five configurations before the wall-clock cutoff. This pool differs from Table 1, so absolute Q values should be compared within each table.

Proposer scale extrapolation. Table 3(c) uses three attempts per problem and the same effective-length calculation, response-selection rule, and timeout handling as the checkpoint evaluation. H-best and H-end use the response selected for mean L_{eff}, with no separate selection for either metric. For these two rates, replay retains only the executable prefix before the first canonical state revisit, consistent with runtime admission. Let its states be $(s_{i,0}, \ldots, s_{i,m_i})$ for problem i, and let h be the numeric LM-cut heuristic used by the NLM-CutPlan scorer. Over all $N = 40$ problems at each scale, including those whose selected response does not meet the offline usability threshold, we

report

$$\text{H-best} = \frac{100}{N}\sum_{i=1}^{N}\mathbb{I}\left[\min_{0\le j\le m_i} h(s_{i,j}) < h(s_{i,0})\right],$$

$$\text{H-end} = \frac{100}{N}\sum_{i=1}^{N}\mathbb{I}[h(s_{i,m_i}) < h(s_{i,0})].$$

An empty or unavailable executable prefix counts as no improvement. Endpoint heuristic values worsen on 10/40 problems at scale 45 and 15/40 at scale 50; the remaining non-improving endpoints tie their initial values. These rates measure improvement frequency, not magnitude. Solver-level gains at these scales are not evaluated.

C.5 Solution Quality and Intervention Diagnostics

In Table 9, Cost W/T/L counts lower, equal, and higher TRACS plan costs at the paired expansion cutoff C_i relative to the baseline on jointly solved pairs. *Req./inst.* is the mean number of state-level requests per problem, summed across its native search phases. *Usable* is the fraction of all online generations that do not time out and yield a nonempty, executable, noncyclic prefix. *Inj./req.* is the mean number of states admitted per request after validation and native filtering; it measures admission, not the number of states that contribute to a discovered solution.

Table 9: **Solution-quality and intervention diagnostics by problem scale.** Cost W/T/L uses paired expansion cutoffs. D and Z denote Depots-Num and ZenoTravel-Num. Means are rounded; a displayed zero can represent a small nonzero average.

Eager: NLM-CutPlan

Set	Cost W/T/L	Req./inst.	Usable	Inj./req.
D-10	0/48/0	0.0	83.3%	0.00
D-20	2/48/0	14.1	79.5%	0.32
D-30	3/45/0	8.3	87.8%	1.09
D-40	1/9/1	6.7	92.0%	2.70
Z-10	0/50/0	0.1	78.6%	0.00
Z-20	0/50/0	14.3	91.1%	0.24
Z-30	2/38/0	8.8	89.6%	1.34
Z-40	2/9/1	7.5	90.9%	3.01

Lazy: Count Downward

Set	Cost W/T/L	Req./inst.	Usable	Inj./req.
D-10	0/48/0	0.1	88.9%	0.00
D-20	1/49/0	11.2	81.2%	2.16
D-30	8/23/6	11.7	78.7%	11.52
D-40	3/1/0	18.9	84.4%	13.23
Z-10	0/50/0	0.0	82.6%	0.00
Z-20	0/50/0	12.0	79.6%	2.11
Z-30	6/26/3	11.4	89.0%	10.98
Z-40	2/2/1	20.0	87.1%	12.40

Usable generations do not necessarily survive native filtering, as the low eager admission rates illustrate. The cost-loss entries also show that quality gains are not uniform across instances.

C.6 Hard-Instance Repeatability Check

We reran TRACS–Lazy on the same Depots-Num test problems at scales 30 and 40, using the same proposer checkpoint and configured runtime settings but explicit experiment seeds 101 and 202. The original run had no explicit seed. Each run is paired with the same completed Count Downward baseline.

Terminal Q uses Equation 15. Its reference cost is the lowest cost found before the wall-clock cutoff by the baseline or any of the three TRACS runs. Baseline Q is therefore identical across comparisons. All problems receive equal weight within each scale and in the 63-problem aggregate.

Table 10: **Targeted DEPOTS-NUM Lazy reruns.** Coverage and Q show baseline $\rightarrow$ TRACS. Coverage and reduction use paired expansion cutoffs; Q uses wall-clock cutoffs and a different reference pool from Table 1. Reduction is the geometric mean expansion reduction (%); n counts matched-quality problems.

Scale	TRACS run	Coverage	Q	Reduction (%)	n
30	Original	37/50 $\rightarrow$ 50/50	0.70 $\rightarrow$ 0.89	27.05	31
	Seed 101	37/50 $\rightarrow$ 50/50	0.70 $\rightarrow$ 0.96	30.52	32
	Seed 202	35/50 $\rightarrow$ 50/50	0.70 $\rightarrow$ 0.90	11.97	30
40	Original	4/13 $\rightarrow$ 13/13	0.25 $\rightarrow$ 0.91	25.13	4
	Seed 101	4/13 $\rightarrow$ 13/13	0.25 $\rightarrow$ 0.84	7.15	3
	Seed 202	4/13 $\rightarrow$ 13/13	0.25 $\rightarrow$ 0.79	20.30	3

All three TRACS runs retain full coverage and exceed the baseline's Q at both scales. Across all 63 problems, their aggregate Q ranges from 0.88 to 0.94, compared with 0.61 for the baseline. Matched-quality expansion reduction also remains positive in every run, although its magnitude varies. These are reruns on a fixed test set, not an independent dataset replication; the scale-40 expansion reduction rests on only three or four matched problems.

C.7 SFT–DAPO ANYTIME COMPARISON

We compare SFT-1100steps with the final DAPO proposer on all 63 DEPOTS-NUM problems at scales 30–40. The SFT run uses seed 101, and we pair it with the DAPO seed-101 run from Appendix C.6, keeping the search, triggering, sampling, and request-budget settings fixed. Both checkpoints solve all 63 problems within their paired expansion cutoffs.

Metric. For this checkpoint pair, let C_i denote the paired expansion cutoff (Appendix C.3), and let $c_i^m(e)$ be the best plan cost found by checkpoint $m \in \{\mathrm{SFT}, \mathrm{DAPO}\}$ after e expansions. Set $c_i^{\mathrm{ref}} = \min_m c_i^m(C_i)$, a shared reference cost rather than a known optimum. Following the cost-ratio quality score used in IPC satisficing evaluation (Taitler et al., 2024), define

$$q_i^m(e) = \begin{cases} 0, & \text{if no plan has been found by } e, \\ c_i^{\mathrm{ref}}/c_i^m(e), & \text{otherwise,} \end{cases}$$
$$A_i^m = \frac{1}{C_i} \int_0^{C_i} q_i^m(e)\, de. \tag{17}$$

We compute this additional anytime measure exactly from the stepwise incumbent logs and average A_i^m equally over all problems, without selecting a matched-quality subset or an intermediate budget. It lies in $[0, 1]$ and represents expected normalized quality at a uniformly chosen expansion count in $[0, C_i]$, rewarding earlier solving and cost improvement.

Table 11: **Paired seed-101 anytime quality.** Scores average quality over paired expansion intervals; higher is better. Δ is the mean paired difference, DAPO minus SFT. The aggregate gives each of the 63 problems equal weight.

Scale	Problems	SFT-1100steps	DAPO	Δ
30	50	0.7910	0.8228	+0.0318
40	13	0.7601	0.7889	+0.0288
All	63	0.7846	0.8158	+0.0312

Results and scope. DAPO improves mean anytime quality by 4.0% relative to SFT, computed from unrounded scores. It has a higher score on 28 problems, ties on 16, and has a lower score on 19.

This is an exploratory analysis of one paired seed. A paired percentile bootstrap with 10,000 instance resamples (resampling seed 101) gives a nominal 95% interval of $[0.0061, 0.0573]$ for the mean

score difference. This interval describes variation across the tested problems, does not account for the exploratory choice of metric, and does not estimate variation across training or search seeds. The result supports additional anytime utility in this run, rather than establishing a consistent advantage across seeds.

D Conditional Completeness and End-to-End Plan Generation

We make explicit the state-space and search assumptions behind Property 2 in Section 4.5, then compare the resulting preservation guarantee with repeated end-to-end plan generation.

D.1 Scope of the Completeness Guarantee

Finite search states. Property 2 requires finitely many native search states, not merely finitely many objects. The state identity in Appendix C.1.2 excludes objective-only accumulators without changing feasibility. In the evaluated Depots-Num tasks, object configurations are finite and vehicle loads are sums of fixed object weights. In our ZenoTravel-Num instances, locations and onboard counts are bounded, and fuel takes bounded integer values under the consumption and refueling rules. These properties give finite native search-state sets for our instances; they need not hold in general numeric planning.

Search phases covered. Property 2 also requires native duplicate handling and pruning to preserve solution reachability under reordered expansion. With a finite cost bound, a cheaper path to an expanded state can bring previously pruned continuations within the bound. These must still be explored, for example by reopening and re-expanding the state; updating its parent pointer alone is insufficient. Our eager schedule does not reopen Closed states (Table 5), so Property 2 does not establish completeness of its later cost-bounded improvement phases.

D.2 Comparison with Repeated Plan Generation

Let $\Pi^*(\mathcal{P})$ be the valid plans for a solvable problem and let $S_i = \{\pi_i \in \Pi^*(\mathcal{P})\}$ denote success on retry i. The cumulative success probability is

$$q_N(\mathcal{P}) = \Pr\left[\bigcup_{i=1}^{N} S_i\right] \uparrow q_\infty(\mathcal{P}) \leq 1. \tag{18}$$

Table 1 reports Pass@1 and Pass@3 as solved-problem counts. Dividing each by the corresponding test-set size gives an empirical estimate of the average $q_1(\mathcal{P})$ or $q_3(\mathcal{P})$ across those problems, rather than a completeness guarantee for every solvable problem. Without assumptions on the retry distribution, even $\Pr[S_i] > 0$ for every i does not imply $q_\infty = 1$. For example, retries may share a failure mode: if $S_1 = S_2 = \cdots = S$ and $\Pr[S] = p \in (0, 1)$, then $q_N = p$ for every N. For i.i.d. retries with a fixed per-instance success probability $p = p(\mathcal{P}) > 0$, the familiar $q_N = 1 - (1 - p)^N \to 1$ does hold. This is an asymptotic probabilistic guarantee, whereas Equation 7 holds for every response sequence under the assumptions of Property 2.

Validation certifies a returned plan but does not ensure eventual generation of one. Property 2 preserves native completeness even when every LLM proposal is unhelpful.