UNIFIED PLAN VERIFICATION WITH STATIC RUBRICS AND DYNAMIC POLICIES FOR RELIABLE LLM PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents can decompose tasks, call tools, and execute multi-step plans, yet they frequently fail for two reasons: (i) pre-execution plans look plausible but are incomplete, inconsistent, or ill-posed; and (ii) during execution, tool outputs reveal conflicts or policy violations that the agent neither detects nor repairs. Existing "LLM-as-judge" scoring is unstable and opaque, while reactive agents lack grounded, learnable control. We introduce VERA, a VERification-Aware planning infrastructure that inserts explicit checks both before and during execution. First, Static Verification via Rubrics (SVR) instantiates an instance-specific, binary checklist from a general taxonomy (completeness, correctness, executability), yielding auditable, stable decisions and actionable feedback for plan revision. Second, a Dynamic Verification Policy (DVP) enforces run-time control: a prompt-optimized rulebook (learned via MCTS-style discrete search, no weight updates) consumes the step context and tool outputs to emit symbolic actions—e.g., browse more candidates, switch tool, skip, backtrack, or accept. VERA is representation-agnostic and applies to structured plans with schemas/tools, unstructured conversational plans, and natural-language plans without tools. Across three regimes, VERA consistently improves task success and constraint satisfaction over strong prompting and agent baselines, reduces temporal/budget and policy violations, and provides rubric-level diagnostics that localize errors. Ablations show SVR (pre-execution screening) and DVP (execution-time control) are complementary; learned rulebooks outperform human-written heuristics with modest extra compute. We release prompts, rulebooks, and evaluation code to facilitate verification-aware agent research.

1 INTRODUCTION

Large Language Model (LLM) agents increasingly plan and execute multi-step tasks across interactive, tool-using environments (Xie et al., 2024; Prasad et al., 2023; Wu et al., 2024). Plan verification, where the agent checks whether the proposed plan is complete, correct, and executable, has become a central component (Parmar et al., 2025; Li et al., 2024; Pathak et al., 2025; Gou et al., 2025). Automating plan verification at scale can improve robustness by catching errors before they cascade and by steering execution when reality diverges from user intent. Nonetheless, prior work reveals three recurring weaknesses. **(1) Lack of grounded and stable judgment:** Prior work often assessed plans with opaque or unstable scalar scores rather than explicit, auditable criteria; for example, Parmar et al. (2025) does not use grounded, instance-specific checks and is sensitive to re-prompting, and Li et al. (2024) relies on a reward model trained from additional labeled data over three coarse criteria, tying verification to task-specific training data. Recent rubric-based efforts (Gou et al., 2025) underscore both the need and feasibility of more structured evaluation, but remain focused on answer-level assessment rather than plan-level verification signals. **(2) Lack of learnable, explicit verification that drives the agent policy:** While many frameworks interleave reasoning with acting (Yao et al., 2023b; Shinn et al., 2023; Prasad et al., 2023; Yao et al., 2023a), adaptation of the agent's actions is typically driven by free-form prompts or hand-tuned heuristics, rather than learnable verification criteria that are explicitly grounded on the current step, the observed tool outputs, and other planning contexts. This formulation thus limits the auditability and effectiveness of

Table 1: Comparison between VERA and existing agent approaches. VERA supports both pre-execution verification and in-execution verification and is agnostic to plan representation.

Approach	Pre-execution Verification			In-execution Verification		Supported Plan Representation
	Supported?	Rubric-based Screening	Replanning	Supported?	Learnable Verification	(*F*ree-text or *S*tructured plan)
Vanilla Prompting	×	×	×	×	×	F
ReAct (Yao et al., 2023b)	×	×	×	✓	×	F
Reflexion (Shinn et al., 2023)	×	×	×	✓	×	F
ADaPT (Prasad et al., 2023)	×	×	✓	✓	×	S
PlanGen (Parmar et al., 2025)	✓	✓	✓	×	×	S/F
AdaPlanner (Sun et al., 2023)	✓	×	✓	✓	×	S
DPPM (Lu et al., 2025)	✓	×	✓	✓	✓	S
LLM-Modulo (Gundawar et al., 2024)	✓	×	✓	✓	✓	S
Ours	✓	✓	✓	✓	✓	S/F

the frameworks. **(3) Ad-hoc pre-and during-execution plan verifications:** Existing systems often check plans ad hoc (before or between actions) but lack a unified mechanism that (i) instantiates task-specific, rubric-level checks before execution, and (ii) maintains those checks during execution to reconcile user-specified constraints with observed tool outputs and newly surfaced facts. Recent benchmarks (Xie et al., 2024; Yao et al., 2024; Zheng et al., 2024) highlight the need for such end-to-end verifications under multi-constraint Xie et al. (2024) and policy-bound Yao et al. (2024)settings. Despite steady progress, these gaps have led to brittle plans, unstable judgments, and failures that are hard to detect and repair in practice (Wei et al., 2025).

To address these weaknesses, we introduce VERA 1, a verification-aware framework that pairs pre-execution verification with during-execution verification-driven policy control for both structured schemas (Tantakoun et al., 2025) and free-text plans (Zheng et al., 2024). To enable grounded and stable judgment, VERA replaces opaque scoring, a single scalar or pass/fail verdict without explanation, with **Static Verification via Rubrics (SVR)**. Specifically, given a fixed, generic taxonomy specifying the desired plan properties (i.e., completeness, correctness, and executability), SVR adopts a rubric generator to instantiate the taxonomy into instance-specific, binary checks. Each check is a concrete yes or no question tied to the current task (e.g., "*Does the plan include lodging for every night?*" for a multi-day trip). Optionally, user- or domain-specific hard rules that the generic taxonomy may not cover (e.g., "*never exceed $500 total airfare*" can be provided as additional constraints. In addition, to prevent cascading failures during execution, VERA includes a pre-execution replanning loop driven by SVR feedback. If the initial plan fails the checklist, the agent iteratively revises the plan using localized feedback until a valid candidate is admitted, avoiding costly execution-time corrections. The SVR output is a checklist with pass or fail decisions and brief rationales that localize errors before any tool calls. VERA further introduces **Dynamic Verification Policy (DVP)**, which drives the agent policy during plan execution with an instance-specific "rulebook" that verifies the plan explicitly based on the grounded planning contexts (e.g., checking whether a flight search returns no results, in which case the agent is suggested to try alternative transportation). The rulebook is learned by editing and evaluating textual rules on training scenarios through prompt optimization (Wang et al., 2023; Srivastava & Yao, 2025), which reduces reliance on hand-written heuristics while improving upon free-form prompts. Finally, VERA unifies the dual phases of plan verification, with SVR gating plans before execution and DVP maintaining alignment during execution as new evidence arrives, providing continuous verification signals that improve reliability. A comparison between VERA and state-of-the-art agent approaches can be found in Table 1.

We evaluate VERA on three datasets: TravelPlanner (Xie et al., 2024), τ-bench (Yao et al., 2024), and NaturalPlans (Zheng et al., 2024), where VERA is shown to outperform state-of-the-art baselines by 26.64%, 13.13% and 6.24% end-task success rate, respectively. Qualitative traces show that SVR prevents ill-posed plans effectively, while DVP mitigates runtime conflicts by leveraging the learned rulebook. In addition, we observed improvement in the stability of pre-execution plan verification, with reduced variance across runs, as well as a bounded overhead, with cost–performance curves that preserve most gains under a modest verification frequency. Collectively, these results support the claim that explicit, checklist-based pre-execution screening combined with learned, execution-time control yields stable and effective LLM planning.

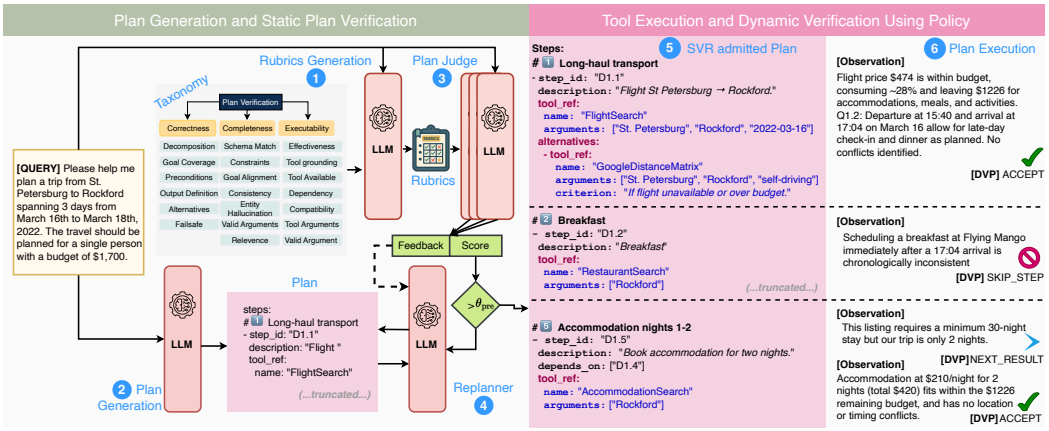

Figure 1: Overview of VERA. Given a user query, VERA first instantiates an SVR rubric (Step 1) and generates an initial plan (Step 2). A rubric-guided judge then verifies the plan (Step 3) and provides targeted feedback for revision (Step 4). Once a plan passes all rubric checks (Step 5), it is admitted for execution. During execution (Step 6), the Dynamic Verification Policy (DVP) monitors tool outputs and adaptively accepts, skips, or explores alternative actions to ensure constraint satisfaction and plan robustness.

2 Related Work

Prior work on LLM-based planning spans reasoning traces (Wei et al., 2022; Yao et al., 2023a), interleaved agent execution (Yao et al., 2023b; Shinn et al., 2023), and planning-first pipelines (Parmar et al., 2025; Sun et al., 2023; Lu et al., 2025; Gundawar et al., 2024). While these systems improve robustness via tool use, decomposition, or modular verification, they typically rely on unstructured feedback, coarse reward models, or domain-specific constraints that are either not auditable or not learnable. AoP (Li et al., 2024) formalizes multi-agent workflows but still lacks persistent, instance-specific checks during execution. In contrast, VERA introduces a unified, rubric-based framework where pre-execution screening (SVR) and execution-time policy control (DVP) are both grounded in learnable, instance-specific verification rules. This yields stable and actionable verification signals across diverse planning settings. A detailed comparison and extended discussion with baselines is provided in Appendix 5.3.

3 VERA: LLM Planning with Unified Plan Verifications

3.1 Preliminaries

We consider an agent that solves a user request Q with optional hard constraints $\mathcal{C}$ by interacting with an environment $\mathcal{E}$ that exposes a catalog of typed tools $\mathcal{T}$. The agent achieves this goal through *planning* (Aghzal et al., 2025), where the objective is to find a sequence of actions $\mathcal{P} = \langle a_1, a_2, \cdots, a_T \rangle$ that transitions the agent from the initial state s_0 to a terminal state s_T that indicates the success of solving the user request. In the interactive, tool-using task environments we focus on, each action a_t is either a tool invocation, i.e., $a_t = (\texttt{op}, \texttt{args})$, with $\texttt{op} \in \mathcal{T}$ being a tool and $\texttt{args}$ being the typed arguments, or a terminal $\texttt{finish}(y)$, which returns a final artifact y. We define $s_0 = (Q, \mathcal{C})$ as the initial state. Once the agent executes an action a_t, the environment emits an observation o_t based on the tool-calling, yielding a new state $s_{t+1} = (Q, \mathcal{C}, o_1, \cdots, o_t)$, and the process repeats until the agent decides to terminate and perceives the final artifact y. The success of the plan can be formally defined by a task utility $U(y; Q, \mathcal{C})$.

Existing LLM-based planning frameworks often directly prompt their LLM backend to generate $\mathcal{P}$ (Wei et al., 2025). However, naively executing $\mathcal{P}$ could miss required elements, violate constraints, or drift at runtime. In VERA, we address these issues by proposing a rubric-based *pre-execution verification* of $\mathcal{P}$, followed by *during-execution control* through a learned verification policy.

3.2 PRE-EXECUTION VERIFICATION: STATIC VERIFICATION VIA RUBRICS (SVR)

VERA first introduces Static Verification via Rubrics (SVR) for pre-execution plan verification.

A Generic Taxonomy for Plan Verification. SVR screens a candidate plan $\mathcal{P}$ before any tool call using a fixed, human-authored taxonomy (Fig. 1), denoted as $\mathcal{I}$. The taxonomy, inspired by Wei et al. (2025), encompasses three generic properties, i.e., Completeness, Correctness, and Executability. **Completeness** asks whether the required elements and structure are present, including invokable steps, goal coverage, explicit preconditions, output definition, and grounded alternatives for tools. **Correctness** enforces logical or factual soundness, including schema matches where applicable, constraint satisfaction, step-to-goal alignment, internal consistency, hallucination avoidance, and relevance. **Executability** tests whether the plan can actually run in the target environment, which involves tool grounding/availability, dependency satisfiability, environment compatibility, argument counts/types, and semantic validity.

Automatic Rubric Generation for Grounded Plan Verification. To enable grounded plan verification, SVR introduces a rubric generator, which maps the generic taxonomy to an *instance-specific* rubric, i.e., $\Gamma(Q, \mathcal{C}, \mathcal{I}) \to \mathcal{R} = \{r_1, \ldots, r_m\}$. Specifically, the rubric consists of binary checks phrased as concrete yes or no questions tied to the task and constraints, e.g., under Completeness: *"Does the plan include lodging for every night within the requested dates and budget?"*; under Correctness: *"Does each booking step respect the $500 budget limit?"*; and under Executability: *"Are all required inputs, such as travel dates and budget constraints, obtained before attempting to book?"* The instantiated rubric $\mathcal{R}$ is fixed for the given $(Q, \mathcal{C})$ and reused verbatim across other execution steps. It is important to note that the rubric is generated *automatically* based on the task, which thus renders SVR scalable.

Judging, Aggregation, and Plan Admission. Given $(Q, \mathcal{C}, \mathcal{P})$, we present the generated rubric items $\mathcal{R}$ to a set of LLM judges, denoted as $\{J_k\}_{k=1}^{K}, K \geq 1$, for scoring.[1] Each judge returns a binary decision $g_r^{(k)}(\mathcal{P}) \in \{0, 1\}$ with a short rationale grounded in the planning context for each rubric item r; all rubric items are weighted equally. When $K > 1$, we aggregate per-item verification results by a majority vote over judges, with ties mapped to 0, to obtain $\bar{g}_r \in \{0, 1\}$. The SVR score is the simple mean $\Phi_{\text{pre}}(\mathcal{P}) = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \bar{g}_r$. A plan is admitted if $\Phi_{\text{pre}}(\mathcal{P}) \geq \theta_{\text{pre}}$, where θ_{pre} is a threshold pre-defined per domain. Otherwise, we aggregate feedback from judges on failed items and their rationales and invoke a replanner to edit the plan. SVR re-evaluates the edited plan against the same rubric, repeating until it yields the admitted plan $\mathcal{P}^{\text{static}}$ or it has replanned for a pre-defined maximal number of rounds (in which case, we return the final revised plan from the last round as $\mathcal{P}^{\text{static}}$. By decomposing the plan verification into a series of factoid binary checks, the LLM judges empirically display much more stable scoring compared to opaque scalar scoring in prior work (Parmar et al., 2025), which also makes the verification process auditable.

3.3 DURING-EXECUTION CONTROL: DYNAMIC VERIFICATION POLICY (DVP)

Complementing the pre-execution plan verification, VERA further includes a Dynamic Verification Policy (DVP) that guides the agent planning through in-execution plan verification.

A Verification Policy during Execution. Given an SVR-admitted plan $\mathcal{P}^{\text{static}} = \langle a_1, \ldots, a_T \rangle$, DVP dynamically verifies each action during the agent's execution. At each step t, it checks whether the observation resulting from executing a_t remains consistent with the task constraints and selects an explicit control action based on its verification. Formally, DVP achieves this goal via a *verification policy* of $a_t^{\text{verify}} \sim \pi_{\text{verify}}\big(\cdot \mid s_t, \mathcal{P}^{\text{static}}, \mathcal{E}\big)$, where a_t^{verify} is a policy action chosen from: `accept`, which accepts the current observation and moves on to executing a_{t+1}, `next_result` requests further execution results for a_t (e.g., displaying the next few returned flight choices), `alt_tool` re-runs a_t using a different available tool in $\mathcal{E}$, `skip_step` skips a_t and proceed to a_{t+1}, and `backtrack` reverts to the most recent committed step that can resolve the conflict and replans

[1] In practice, we found that involving multiple judges could diversify the verification feedback; however, SVR is still applicable when $K = 1$.

from there. In doing so, the verification policy implicitly controls the agent's task planning, resulting in an effective plan $\mathcal{P}^{\text{dyna}} = \langle a_1, \cdots, a_{T'} \rangle$ that could diverge from $\mathcal{P}^{\text{static}}$.

Learning Verification Rulebook with Prompt Optimization. The key challenge of DVP lies in implementing the verification policy π_{verify}. Prior work has dominantly relied on hand-crafted heuristics or unstructured prompts to implement the verification, which makes the verification criteria implicit and the judgment sensitive to long trajectories and distractors (Yao et al., 2023b; Shinn et al., 2023; Prasad et al., 2023). In VERA, we propose the idea of *learning a rulebook for dynamic plan verification*. A rulebook is a prompt that specifies domain-specific guidelines for checking the observation o_t (i.e., the output of executing a_t) against the task constraint. We learn the rulebook through discrete prompt optimization (Srivastava & Yao, 2025; Wang et al., 2023). Specifically, we treat the optimization as a discrete search over candidate rulebooks using a Monte Carlo Tree Search (MCTS)-style procedure to balance exploration (trying new rule variants) and exploitation (refining effective ones) Wang et al. (2023). Formally, we define an Markov Decision Process (MDP) over policy versions: the optimization state at iteration t[2] is the current policy $\pi^{(t)}_{\text{verify}}$; an optimization action is a natural-language edit to its rulebook; and the transition is $\pi^{(t+1)}_{\text{verify}} = \textsc{Edit}\big(\pi^{(t)}_{\text{verify}}, \text{edit}\big)$. At evaluation time, $\pi^{(t)}_{\text{verify}}$ maps $(s_t, \mathcal{P}^{\text{static}}, \mathcal{E})$ to an action in $\{$`accept`, `next_result`, `alt_tool`, `skip_step`, `backtrack`$\}$. For example (Fig. 1), if $\pi^{(t)}_{\text{verify}}$ lacks any schedule-feasibility check, the agent may book *breakfast* immediately after a late-arrival flight. Observing this error yields an edit that inserts: "*Is arrival time $\leq$ the next step's start time? Compare arrival time with the next step's start to ensure downstream schedule feasibility.*" Applying this edit produces $\pi^{(t+1)}_{\text{verify}}$, which then rejects or defers such temporally inconsistent steps in future executions.

At each MCTS node we keep the current policy version π^{t}_{verify} and a mini-batch of training sample $\{(Q_i, \mathcal{C}_i, \mathcal{E}_i, \mathcal{P}^{\text{static}}_i, y_i)\}$. The optimization loop proceeds in four steps: **(1) Plan Execution:** for each i, run $\mathcal{P}^{\text{static}}_i$ under $\pi^{(t)}_{\text{verify}}$ in $\mathcal{E}_i$, producing a final artifact $\hat{y}^{(t)}_i$ and a step-level log of observations and chosen control actions a^{verify}_t. **(2) Error extraction:** compare $\hat{y}^{(t)}_i$ with y_i and parse the logs to identify actionable control failures (e.g., accepting a constraint violation, unnecessary `skip_step`). **(3) Feedback:** use a meta-prompt m_{fb} to summarize observed errors into actionable edits u_t, using an LLM optimizer, to the rulebook of $\pi^{(t)}_{\text{verify}}$. **(4) Update:** apply another meta-prompt, on an optimizer LLM, m_{opt} to generate the revised policy $\pi^{(t+1)}_{\text{verify}} = \textsc{Edit}\big(\pi^{(t)}_{\text{verify}}, u_t\big)$. We score each policy π_S on a development set $\mathcal{V}$ using task-specific metrics: $R\big(\pi^{(t)}_{\text{verify}}\big) = \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} \text{Metric}\big(\hat{y}^{(t)}_i, y_i\big)$ where Metric_i is pass rate for TRAVELPLANNER, exact match for NATURALPLANS, and pass@1 for τ-BENCH. The search continues for a fixed tree depth and returns $\pi^{\star}_{\text{verify}} \in \arg\max_{t\in\{0,\ldots,T\}} R\big(\pi^{(t)}_{\text{verify}}\big)$ which we freeze for test-time control. This adapts prompt optimization to *policy learning*: Instead of tuning task instructions, we discover an interpretable verification rulebook that selects explicit control actions during execution, without modifying $\mathcal{P}^{\text{static}}$ or training new models. We provide additional details and a complete algorithm with examples in Appendix B.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate our planning framework on three diverse benchmarks, each stressing different aspects of planning spanned across multiple domains: **(i)TravelPlanner** (Xie et al., 2024): A realistic trip-planning benchmark grounded in tool use (e.g., flight, hotel, map APIs) and multi-day itinerary and commonsense constraints. Tasks require constructing travel plans over multiple days, satisfying explicit user needs (budget, preferences) and implicit commonsense constraints (feasible routes, accommodations each night). The emphasis is on structured constraint satisfaction, i.e., the agent must gather information via tools and assemble a coherent plan that meets all requirements

[2]In this context, we double-use the index t to denote the iteration during prompt optimization.

(e.g., no overlapping flights, within budget) before execution. This benchmark tests our framework's ability to verify plans against hard constraints and leverage external tools for information. **(ii) τ-bench** (Yao et al., 2024): Interactive tasks in real-world domains where the agent converses with the user, invokes domain-specific APIs, and must follow policy or rules. The benchmark evaluates goal-state consistency and reliability under domain constraints. **(iii) NaturalPlan:** (Zheng et al., 2024)Free-form planning of meetings, calendar, travel, expressed in natural language but with symbolic structure. It emphasizes the execution of the exact plan and the adaptability under changing conditions.

Evaluation Metrics. We adopt benchmark-specific metrics that capture both correctness and robustness. TravelPlanner is evaluated on delivery rate (whether a complete plan is produced), hard-constraint and commonsense-constraint pass rates (satisfaction of explicit user requirements vs. implicit feasibility), and final success rate (all constraints simultaneously satisfied). τ-bench uses database-state accuracy, comparing the final tool-updated state against the annotated goal, together with pass@1 metrics that assess consistency across repeated trials of the same task. NaturalPlan relies on exact match, requiring the agent's generated symbolic plan to align perfectly with the gold reference, thereby stressing precision and adaptability in free-form planning.

Baselines. We compare VERA against ten representative baselines spanning prompting-only, reasoning-augmented, and planning-centric paradigms. **Vanilla Prompting** serves as a single-shot generation baseline with no planning or feedback. **CoT** (Wei et al., 2022) and **ToT** (Yao et al., 2023a) enhance reasoning bandwidth via intermediate traces or tree search but offer no structured verification. **ReAct** (Yao et al., 2023b) and **Reflexion** (Shinn et al., 2023) interleave tool use with thoughts or self-critiques, enabling partial in-execution correction but lacking any pre-execution screening. **ADaPT** (Prasad et al., 2023) adaptively refines plans during execution when stuck, but performs no up-front validation. **PlanGen** (Parmar et al., 2025) and **DPPM** (Lu et al., 2025) generate complete plans and apply pre-execution filters, though their checks are often heuristic or schema-specific. **AdaPlanner** (Sun et al., 2023) revises plans reactively under tool feedback but lacks plan gating before execution. **LLM-Modulo** (Gundawar et al., 2024) incorporates symbolic verifiers into both planning phases, but depends on handcrafted rules and lacks learnable verification. In contrast, VERA unifies pre-execution (SVR) and in-execution (DVP) under a single, rubric-based interface that is learnable, instance-grounded, and generalizable across domains.

Hyperparameters and Experimental Settings In our pipeline, we leverage GPT-4o in our experiments for planning, evaluation, and rubric generation. Throughout our experiments, we use $k = 1$ demonstrations for initial plan generation to satisfy the plan schema (if any). For the remaining modules (judge, rubrics, etc.), we do not use any demonstrations.[3] The sampling temperature by default is 0.7 for all LLMs. Further experimental details can be found in Appendix D, which includes reproducible prompts.

4.2 Main Results

Unified SVR and DVP improve overall results on all datasets. On TravelPlanner, VERA delivers executable plans in 100% of cases, matching the best pre-execution baselines and improving over during-execution methods, indicating that plans admitted by SVR are consistently executable. On the end metric *Final*, VERA reaches 44.44, exceeding the strongest test-time pre-execution baseline DPPM by $+26.64$ points $(17.80 \mapsto 44.44)$, and surpassing the trained LLM-Modulo* by $+23.84$ $(20.60 \mapsto 44.44)$. It also improves during execution approaches, e.g., ADaPT, by large margins $(6.60 \mapsto 44.44)$. In addition, VERA attains the highest commonsense and hard-constraint pass rates in Table 2. Qualitatively, we find DVP closes common failures such as exploring alternate tools when there are no valid candidates or violating temporal/budget constraints; some dataset-specific priors (e.g., always prefer flights for certain routes) are not encoded, leaving headroom despite the strongest reported Final.

On τ-Bench, VERA attains 73.04 on Retail and 60.00 on Airline, improving over the best during-execution baselines Reflexion (54.78, 52.00) by $+18.26$ $(54.78 \mapsto 73.04)$ and $+8.00$ $(52.00 \mapsto 60.00)$, respectively; gains over ReAct (53.70, 46.00) are $+19.34$ and $+14.00$. On NaturalPlans,

[3] Our framework is flexible enough to support additional in-context demonstrations.

Table 2: **Main results across three planning regimes.** We implement all the baselines ourselves using the settings and prompts specified in the paper. We were not able to replicate the numbers for LLM-Modulo; hence, we report the numbers from their original paper for the available dataset, leaving others with a '-'. All values are percentages (%). Best per column in **bold** with a light tint. Small green deltas on OURS indicate absolute gains over *Vanilla Prompting*.

Method	TravelPlanner (Constraint-level Rates ↑)				TauBench (Pass@1 ↑)		NaturalPlans (Exact Match ↑)		
	Delivery	Common-sense	Hard Constr.	Final	Retail	Airline	Trip	Meeting	Calendar
Vanilla Prompting	*89.40*	*02.80*	*10.60*	*0.60*	*23.48*	*24.00*	*03.70*	*45.20*	*43.70*
CoT	*100*	*22.22*	*26.67*	*04.44*	*24.34*	*30.00*	*26.60*	*24.70*	*33.60*
ToT	*100*	*24.50*	*27.60*	*04.44*	*28.70*	*34.00*	*42.63*	*41.40*	*58.90*
ReAct	*82.20*	*03.90*	*06.70*	*0.60*	*53.70*	*46.00*	*34.70*	*38.90*	*47.90*
Reflexion	*93.90*	*08.80*	*04.44*	*0.00*	*54.78*	*52.00*	*35.75*	*34.80*	*51.70*
ADaPT	*95.30*	*12.50*	*11.70*	*06.60*	*42.61*	*38.00*	*36.80*	*36.50*	*48.90*
AdaPlanner	*97.80*	*14.50*	*13.70*	*15.60*	*46.09*	*42.00*	*43.90*	*45.66*	*55.70*
DPPM	*100*	*21.10*	*22.20*	*17.80*	*47.83*	*48.00*	*45.80*	*45.40*	*60.70*
PlanGen	*100*	*16.10*	*16.70*	*12.80*	*41.75*	*46.00*	*44.75*	*45.63*	*62.66*
LLM-Modulo*	*100*	*40.60*	*39.40*	*20.60*	-	-	-	-	-
VERA	***100*** (+11)	***44.44*** (+41.6)	***48.89*** (+38.20)	***44.44*** (+43.80)	***73.04*** (+50.40)	***60.00*** (+36)	***48.17*** (+44.30)	***49.80*** (+4.608)	***68.90*** (+25.20)

Legend: Structured Plans, Unstructured Plans, or Unstructured and tool-free Plans

VERA sets the best scores on Meeting Scheduling (49.80) and Calendar Planning (68.90), improving over the strongest baselines by $+4.17$ $(45.63 \mapsto 49.80)$ and $+6.24$ $(62.66 \mapsto 68.90)$. On Trip Planning, VERA reaches 48.17, topping PlanGen (44.75) by $+3.42$ $(44.75 \mapsto 48.17)$ and ToT (42.63) by $+5.54$.

Qualitatively, we observe that DVP plays a key role in mitigating execution-time failures by enforcing rubric-derived rules specific to each domain (Appendix C, Table C). In NATURALPLANS, common agent errors include malformed headers or incorrect day transitions, e.g., missing the final "Day D" line or breaking contiguous day arithmetic (C2). DVP learns to identify such failures (e.g., invalid recurrence or unbalanced visit–flight blocks) and initiates localized repairs, improving syntactic well-formedness and final scores without degrading content coverage. On τ-Bench, DVP enforces task-type isolation and booking protocol consistency. For instance, it detects policy violations such as switching tasks mid-interaction (e.g., booking a new flight while canceling another), or skipping mandatory booking steps like confirming baggage count or final payment consent. In such cases, DVP prompts clarification or correction before proceeding, avoiding silent errors or premature tool calls. While SVR ensures only structurally and semantically sound plans are admitted upfront, DVP provides continuous guardrails that reconcile the evolving execution context with task-specific requirements. Notably, some domain priors (e.g., favoring flights over trains in Europe) remain unencoded, suggesting further headroom even with the strongest reported *Final* performance.

Insight 1: SVR admits only plans that pass a fixed checklist of completeness, correctness, and executability, achieving 100% delivery on *TravelPlanner*. DVP then keeps execution aligned with Q and hard constraints from $\mathcal{C}$ and $\mathcal{E}$ by selecting explicit control actions at runtime.

5 ADDITIONAL ANALYSES

5.1 ANALYSIS OF SVR

SVR complements VERA by providing accurate and grounded judgment A key goal of SVR is to replace noisy scalar rewards with grounded, auditable criteria for plan validation. We compare four planning approaches by asking a simple question: "*as the reward increases, how much does the chance of success rise?*" For each method[4], we model $\Pr(\text{success} \mid \text{reward})$ with a logistic link on the raw points and visualize it alongside binned estimates at 10 points with binomial CIs. We fit a logistic link to summarize the monotone reward–success relationship and use AUC (with 95% CIs) as a comparable separation metric.

[4]For PlanGen, we use the same prompt provided in the paper, while for AdaPlanner and DPPM, we use another prompt to give a reward based on evaluation feedback

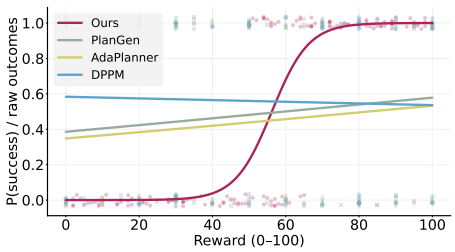

Figure 2: We plot $\Pr(\text{success} \mid \text{reward})$ using a logistic fit (solid line) and binned empirical estimates (points with 95% binomial CIs). VERA with SVR provides more stable and faithful verification signals.

Figure 2 shows a sharp transition at the ∼56 reward threshold with AUC ∼0.97, indicating a strong discriminative power and a steep reward-to-success coupling. PlanGen rises only mildly (threshold ∼60, AUC ∼0.57), while AdaPlanner and DPPM are similarly shallow. This confirms that SVR provides more faithful verification, enabling stable plan admission decisions. Qualitatively, as shown in Table C, SVR succeeds because its fine-grained rubrics encode essential planning principles that others leave implicit. For instance, the *Decomposition* rubric ensures each step is atomic and tool-invokable, by generating binary rubrics like "`PASS if the plan breaks the trip into atomic tool-invokable steps (...) FAIL if steps are coarse (e.g., 'book flight and hotel' as one) (...)`" Under *Hallucination Avoidance*, with a rubric "`PASS if all concrete entities (...) are sourced from prior operator outputs; (...); FAIL if fabricated identifiers appear without provenance.`", SVR checks that all concrete entities (e.g., flight numbers, hotels) are grounded in prior tool outputs or clearly marked as placeholders, preventing fabricated results from leaking into execution. These precise, interpretable checks allow SVR to reject ill-formed plans early and consistently admit those that are complete, correct, and executable.

Insight 2: SVR's fine-grained rubrics yield grounded, reliable verification: higher rewards imply higher success, unlike prior noisy reward heuristics.

SVR provides stable and interpretable judgment across raters. To evaluate judgment stability, we compare SVR (VERA) and PlanGen, the two approaches that explicitly assign plan-level verification scores. PlanGen uses free-form constraint extraction followed by scalar scoring, while VERA uses binary rubric-based checks aggregated into a structured reward. We prompt three judges on the same plan under each method and measure their agreement after binning scores into 20 equal-width bins over $[-100, 100]$. We group each trial into one of three categories: (i) full agreement (3/3 judges in the same bin), (ii) partial agreement (2/3 judges in the same bin), and (iii) full disagreement (each in a different bin).

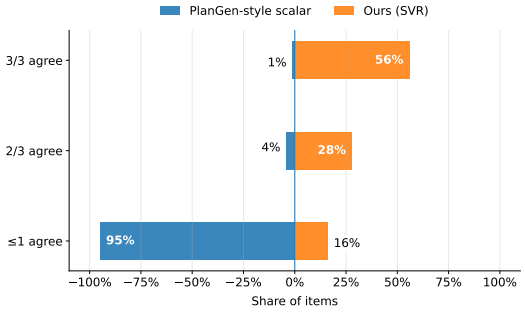

Figure 3: Inter-judge agreement over verification scores. SVR yields stable judgment (56% 3-way agreement), while PlanGen shows highly divergent scores (95% complete disagreement).

As shown in Figure 4, PlanGen exhibits highly unstable verification: 95% of its plans fall into the disagreement case (1/1 agreement), with only 1% full consensus. In contrast, VERA flips this distribution, with 56% of plans achieving full agreement, and only 16% showing total disagreement. Beyond binning artifacts, we also report scale-invariant metrics. *SVR yields higher inter-judge consistency* across the board: mean pairwise Pearson correlation 0.449 vs 0.379, intra-class correlation ICC(1,k) 0.728 vs 0.613, and lower average deviation 25.8 vs 30.3. These results confirm that rubricized, binary-feedback-based verification not only improves plan quality, but also yields more stable and auditable decisions.

Insight 3: SVR enables stable verification: 3-way judge agreement rises to 56% under rubric-based checks, compared to just 1% in free-form scoring (PlanGen).

5.2 Analysis of DVP

DVP improves runtime decisions via learned, task-specific rulebooks. While SVR can admit only well-formed plans, effective execution still requires handling runtime deviations, such as, unavailable tool results (no flight options on a given date) or implicit constraint violations (scheduling conflicts). Figure 4 shows that disabling DVP (–DVP) causes final success to drop sharply from 44.44% to 20.0%, and hard constraint adherence from 47.10% to 12.38%, despite using SVR for plan admission. Substituting DVP with ReAct-style prompting (+ReAct) improves slightly (22.22% final success), but still underperforms across commonsense and constraint metrics. Notably, from Table 2, a standalone ReAct agent, without SVR or DVP, achieves just 3.9% commonsense and 6.7% hard-constraint satisfaction. Adding SVR raises these by over +15 points, while further replacing ReAct with DVP doubles the success rate and restores grounded constraint enforcement. This confirms that DVP provides more precise and effective runtime control than unstructured prompting, especially when built on stable SVR-admitted plans.

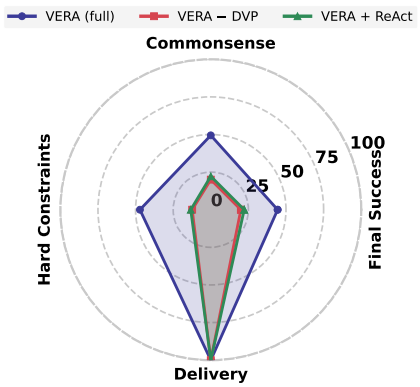

Figure 4: Disabling DVP or replacing it with ReAct-style prompting significantly degrades performance

Qualitatively, we find that **DVP** effectively mitigates commonsense violations and enforces hard constraints through prompt-optimized control policies (Table C). In TravelPlanner, it resolves temporal conflicts (e.g., late arrivals) by deferring downstream steps, avoiding brittle failures when rigid scheduling collides with real-world constraints. It also enforces dietary and budget constraints by skipping over-ineligible meals or hotels, and activates alternate strategies when no feasible lodging option exists. Notably, DVP exhibits **forward-looking, cost-aware behavior** via rules that anticipate future mandatory costs and proactively prune expensive early options to preserve feasibility, e.g., skipping soft goals or retrying with cheaper candidates. These behaviors are governed by textual rulebooks optimized from demonstrations, without hand-coded heuristics or model fine-tuning. SVR contributes by gating ill-formed or constraint-violating plans upfront, but DVP remains central in handling in-execution-time contingencies across domains.

Insight 4: DVP learns structured and interpretable control policies that resolve common sense conflicts, enforce hard constraints, and optimize for future costs, achieving robust execution.

5.3 Generalization to GameOf24

Table 3 shows that VERA attains **71.00%** accuracy on *Game of 24*, outperforming *ToT* (52.00%), *CoT* (7.00%), and *Vanilla* (6.00%). We attribute the gains to the SVR-admitted plan's explicit search skeleton and guards: (i) exhaustive but non-redundant expansion by "*pick any unordered pair* (x, y)" and enumerate $\{x{+}y,\ x{-}y,\ y{-}x,\ x{\times}y,\ x/y\ (y{\neq}0),\ y/x\ (x{\neq}0)\}$ while "*treating* $+, \times$ *as one order only*" to avoid symmetric duplicates; (ii) recursive reduction "*replace* $\{x,y\}$ *with* r *and repeat until list length* $= 1$", with a crisp termination test "*if* $|r-24| < 10^{-6}$ *success; otherwise backtrack*"; and (iii) cheap but crucial pruning: "*skip identical states (same multiset up to order)*" and "*skip divisions by zero*." These checks make completeness and safety auditable *before* execution (SVR), while DVP's runtime control turns them into concrete actions, e.g., issuing `backtrack` when the singleton result $\neq 24$, or `next_result` when a candidate reproduces a visited multiset, without editing plan text. Together, the plan-level invariants and execution-time decisions yield a stronger, more stable search than free-form prompting or unconstrained tree sampling.

Method	Accuracy (%)
Vanilla	6.00
CoT	7.00
ToT	52.00
VERA	**71.00**

Table 3: **Game of 24** accuracy comparison.

Insight 5: Structured verification plus execution-time control transfers across domains, even without tools or retraining.

REFERENCES

Mohamed Aghzal, Erion Plaku, Gregory J Stein, and Ziyu Yao. A survey on large language models for automated planning. *arXiv preprint arXiv:2502.12435*, 2025.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, et al. Mind2web 2: Evaluating agentic search with agent-as-a-judge. *arXiv preprint arXiv:2506.21506*, 2025.

Atharva Gundawar, Mudit Verma, Lin Guan, Karthik Valmeekam, Siddhant Bhambri, and Subbarao Kambhampati. Robust planning with llm-modulo framework: Case study in travel planning. *arXiv preprint arXiv:2405.20625*, 2024.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. Agent-oriented planning in multi-agent systems. *arXiv preprint arXiv:2410.02189*, 2024.

Zhengdong Lu, Weikai Lu, Yiling Tao, Yun Dai, ZiXuan Chen, Huiping Zhuang, Cen Chen, Hao Peng, and Ziqian Zeng. Decompose, plan in parallel, and merge: A novel paradigm for large language models based planning with multiple constraints. *arXiv preprint arXiv:2506.02683*, 2025.

Mihir Parmar, Xin Liu, Palash Goyal, Yanfei Chen, Long Le, Swaroop Mishra, Hossein Mobahi, Jindong Gu, Zifeng Wang, Hootan Nakhost, et al. Plangen: A multi-agent framework for generating planning and reasoning trajectories for complex problem solving. *arXiv preprint arXiv:2502.16111*, 2025.

Aditya Pathak, Rachit Gandhi, Vaibhav Uttam, Arnav Ramamoorthy, Pratyush Ghosh, Aaryan Raj Jindal, Shreyash Verma, Aditya Mittal, Aashna Ased, Chirag Khatri, et al. Rubric is all you need: Enhancing llm-based code evaluation with question-specific rubrics. *arXiv preprint arXiv:2503.23989*, 2025.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Saurabh Srivastava and Ziyu Yao. Revisiting prompt optimization with large reasoning models-a case study on event extraction. *arXiv preprint arXiv:2504.07357*, 2025.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245, 2023.

Marcus Tantakoun, Christian Muise, and Xiaodan Zhu. Llms as planning formalizers: A survey for leveraging large language models to construct automated planning models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 25167–25188, 2025.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*, 2023.

Hui Wei, Zihao Zhang, Shenghua He, Tian Xia, Shijia Pan, and Fei Liu. Plangenllms: A modern survey of llm planning capabilities. *arXiv preprint arXiv:2502.11221*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Duo Wu, Jinghe Wang, Yuan Meng, Yanning Zhang, Le Sun, and Zhi Wang. Catp-llm: Empowering large language models for cost-aware tool planning. *arXiv preprint arXiv:2411.16313*, 2024.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. *URL https://arxiv. org/abs/2305.10601*, 3, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. T-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Huaixiu Steven Zheng, Swaroop Mishra, Hugh Zhang, Xinyun Chen, Minmin Chen, Azade Nova, Le Hou, Heng-Tze Cheng, Quoc V Le, Ed H Chi, et al. Natural plan: Benchmarking llms on natural language planning. *arXiv preprint arXiv:2406.04520*, 2024.

A Extended Related Work Discussion

Reasoning traces without verification. Early progress on test-time reasoning in LLMs focused on producing richer intermediate traces to improve final answers. *Chain-of-Thought* (CoT) elicits step-by-step rationales, often boosting accuracy on arithmetic, commonsense, and symbolic tasks (Wei et al., 2022). *Tree-of-Thought* (ToT) explores branching solution paths and uses search or voting to select a final trajectory (Yao et al., 2023a). While these methods increase reasoning bandwidth, they typically lack *instance-grounded verification*: rationales are not checked against task-specific constraints or external tool feedback before execution. As a result, scalar self-ratings or majority votes can be unstable across prompts or seeds and cannot localize actionable faults such as missing preconditions, violated hard rules, or unexecutable steps.

Interleaving reasoning and acting. Agent frameworks interleave token-level reasoning with tool calls to ground decisions in observations. *ReAct* mixes thoughts and actions to guide retrieval and tool use (Yao et al., 2023b); *Reflexion* adds self-critique to iteratively repair failures (Shinn et al., 2023); and plan-while-acting variants (e.g., Prasad et al., 2023) adapt via free-form prompts or heuristic controllers. These systems improve robustness in open-world settings but generally drive policy updates with unstructured text feedback ("try again," "revise step") rather than *learned, explicit verification criteria* tied to the current step and observed tool outputs. Consequently, their judgments are hard to audit and difficult to reuse across tasks or runs.

Planning-first and adaptive planners. Several works make the plan itself a first-class object. *AdaPlanner* adaptively revises plans during execution under feedback and failures (Sun et al., 2023). *PlanGen* (multi-agent planning with verification and constraint extraction) aims to generate and screen full plans before execution (Parmar et al., 2025). *DPPM* (Decompose, Plan in Parallel, and Merge) targets multi-constraint problems by parallelizing sub-plans and merging them under a coordinator (Lu et al., 2025). *LLM-modulo* composes LLMs with modular skills/verifiers to scaffold complex tasks (Gundawar et al., 2024). These approaches move beyond token-level heuristics, but plan checks are often ad hoc (schema-specific rules, coarse reward models, or opaque scalar scores) and typically do not *unify* pre-execution screening with execution-time policy control under a shared, learnable verification interface.

Multi-agent orchestration and Agent-Oriented Planning (AoP). Multi-agent pipelines specialize roles (planner, critic, tool specialist, executor) for division of labor. Agent-Oriented Planning (AoP) formalizes this with a learned reward model over coarse criteria and additional labeled supervision (Li et al., 2024). While this improves coordination, verification signals are bound to the reward model and dataset, limiting portability and interpretability. Moreover, AoP and related orchestrations typically lack an *instance-specific* checklist that persists from pre-execution to runtime to resolve conflicts when the world diverges from the initial plan.

Algorithm 1 Offline learning of Runtime Control Policy (RCP) via prompt optimization

Inputs:
Training set $\mathcal{D}_{\text{train}} = \{(Q_i, \mathcal{C}_i, \mathcal{E}_i, \mathcal{P}_i^\star, y_i)\}$,
policy LLM (fixed) $\mathcal{M}_{\text{pol}}$, optimizer LLM $\mathcal{M}_{\text{opt}}$, reward $\mathcal{R}$, penalty Cost,
batch size k, depth limit L, iterations τ, exploration weight c (*all decoding temperature* 0)
Initialize:
Initial policy prompt S_0; mappings $A : \mathcal{S} \mapsto \mathcal{U}$ (available edits), $\text{ch} : \mathcal{S} \times \mathcal{U} \mapsto \mathcal{S}$,
rewards $r : \mathcal{S} \times \mathcal{U} \mapsto \mathbb{R}$, $Q : \mathcal{S} \times \mathcal{U} \mapsto \mathbb{R}$, visits $\mathcal{N} : \mathcal{S} \mapsto \mathbb{N}$; best policy $S^\star \leftarrow S_0$, best score $R^\star \leftarrow -\infty$
for $n \leftarrow 1, \ldots, \tau$ **do**
  Sample mini-batch $B \subset \mathcal{D}_{\text{train}}$, $|B| = k$; set $S_t \leftarrow S_0$
  for $t \leftarrow 0, \ldots, L-1$ **do**
    if $A(S_t) \neq \emptyset$ **then** ▷ *Selection (UCB)*
      $u_t \leftarrow \arg\max_{u \in A(S_t)} \left[Q(S_t, u) + c\sqrt{\frac{\ln(\mathcal{N}(S_t)+1)}{\mathcal{N}(\text{ch}(S_t,u))+1}} \right]$
      $S_{t+1} \leftarrow \text{ch}(S_t, u_t)$; $\mathcal{N}(S_t) \mathrel{+}= 1$
    else ▷ *Expansion + Simulation*
      Rollout: For each $(Q_i, \mathcal{C}_i, \mathcal{E}_i, \mathcal{P}_i^\star, y_i) \in B$, execute $\mathcal{P}_i^\star$ under π_{S_t} with $\mathcal{M}_{\text{pol}}$ to get trace τ_i and output $\hat{y}_i$
      Error mining: $E_B \leftarrow \textsc{MineErrors}(\{\tau_i, \hat{y}_i, y_i\}_{i \in B})$ *(e.g., missed next_result, wrong alt_tool, unnecessary backtrack, constraint violations)*
      Edit proposal: $u_t \sim \mathcal{M}_{\text{opt}}(u \mid S_t, E_B, m_{\text{fb}})$
      Policy update: $S_{t+1} \sim \mathcal{M}_{\text{opt}}(S \mid S_t, u_t, m_{\text{opt}})$
      $A(S_t) \leftarrow \{u_t\}$; $\text{ch}(S_t, u_t) \leftarrow S_{t+1}$; $r(S_t, u_t) \leftarrow \frac{1}{|B|} \sum_{i \in B} \left[\mathcal{R}(\hat{y}_i, y_i) \right]$; $\mathcal{N}(S_t) \mathrel{+}= 1$
    end if
    if $\textsc{EarlyStop}(S_{t+1})$ **then break**
    end if
  end for
  Backprop: propagate rollout rewards to update $Q(\cdot, \cdot)$ along the visited path
  Track best: if $r(S_t, u_t) > R^\star$ on a dev split, set $R^\star \leftarrow r(S_t, u_t)$, $S^\star \leftarrow S_{t+1}$
end for
return $S^\star$ *(frozen at test time; induces $\pi_{S^\star}(a_t \mid Q, \mathcal{P}^\star, \mathcal{E}, c_t, s_t, o_t)$ over $\mathcal{A}$)*

Decomposition, search, and parallelization. Orthogonal lines study search over plans (beam, MCTS), subgoal discovery, or parallel synthesis of plan segments before reconciliation. ToT-style search (Yao et al., 2023a), adaptive decomposition (e.g., *ADaPT*-like strategies that split or refine when stuck; Prasad et al., 2023), and parallel planning/merging (Lu et al., 2025) improve coverage but typically assume either (i) a downstream scorer that is not grounded in the instance (susceptible to prompt variance) or (ii) hand-crafted constraints that do not generalize across domains.

Rubric-based evaluation and LLM-as-Judge. Recent work augments LLM evaluation with rubrics or structured criteria to reduce the brittleness of scalar judgments (Gou et al., 2025; Wei et al., 2025). However, most rubricization targets *answer-level* quality (faithfulness, helpfulness), not *plan-level* executability signals. When plan checks are attempted, they are often coarse (pass/-fail) or detached from tool observations, which limits their ability to steer execution.

Our contribution in context. VERA differs from the above along three axes aligned with the issues surfaced in §1. First, instead of opaque scalars or pass/fail checks, **Static Verification via Rubrics (SVR)** instantiates a fixed taxonomy (completeness, correctness, executability) into *instance-specific, binary* questions (e.g., "is lodging allocated for every night?") that are auditable and editable, yielding stable, localized pre-execution diagnostics (contrast with CoT/ToT/ReAct/Reflexion and heuristic screens in Parmar et al., 2025; Li et al., 2024). Second, **Dynamic Verification Policy (DVP)** learns an execution-time rulebook via prompt optimization to map grounded contexts (current step, tool outputs, constraints) to policy control (retry, replan, relax, backtrack), replacing free-form heuristics with explicit, learnable verification that *drives* the agent policy (contrast with Yao et al., 2023b; Shinn et al., 2023; Prasad et al., 2023; Sun et al., 2023). Third, VERA *unifies* pre- and during-execution verification so that the same rubric family gates initial plans and persists as

Algorithm 2 Unified Verification-Aware Planning via SVR and DVP

Inputs:
User query Q, task environment $\mathcal{E}$, operator library $\mathcal{T}$, rubric generator $\mathcal{M}_{rubric}$, planner $\mathcal{M}_{plan}$, static verifier $\mathcal{V}_{\text{SVR}}$, dynamic verifier policy π_{verify}

Phase 1: Static Verification via Rubrics (SVR)
(1) Generate planning rubrics $\mathcal{R} \leftarrow \mathcal{M}_{rubric}(Q)$
(2) Generate candidate plan $\mathcal{P}^{\text{static}} \leftarrow \mathcal{M}_{plan}(Q)$
(3) Evaluate rubric checklist $C \leftarrow \mathcal{V}_{\text{SVR}}(\mathcal{P}^{\text{static}}, \mathcal{R})$
if C contains any failure **then**
  Provide feedback and replan: $\mathcal{P}^{\text{static}} \leftarrow \mathcal{M}_{plan}(Q, \text{feedback})$
  Repeat Steps (3) until plan passes all rubrics or max attempts
end if

Phase 2: Dynamic Verification Policy (DVP)
Initialize execution trajectory $\mathcal{P}^{\text{dyna}} \leftarrow [\,]$
for $t \leftarrow 1$ to T **do**
  Execute a_t from $\mathcal{P}^{\text{static}}$, observe o_t
  Choose verifier action $a_t^{\text{verify}} \sim \pi_{\text{verify}}(o_t, \mathcal{P}^{\text{static}}, \mathcal{E})$
  Case: a_t^{verify}:
  if $a_t^{\text{verify}} =$ `accept` **then**
    Append a_t to $\mathcal{P}^{\text{dyna}}$ and proceed to a_{t+1}
  else if $a_t^{\text{verify}} =$ `next_result` **then**
    Request additional outputs for a_t from $\mathcal{E}$ and re-evaluate
  else if $a_t^{\text{verify}} =$ `alt_tool` **then**
    Re-execute a_t using an alternative tool in $\mathcal{T}$
  else if $a_t^{\text{verify}} =$ `skip_step` **then**
    Discard a_t, move to a_{t+1}
  else if $a_t^{\text{verify}} =$ `backtrack` **then**
    Revert to previous safe step a_j ($j < t$) and replan forward
  end if
end for
return Final executed plan $\mathcal{P}^{\text{dyna}}$

actionable runtime checks (contrast with ad-hoc pre-only screens in Parmar et al., 2025 or controller-only adaptations in Yao et al., 2023b; Shinn et al., 2023). Empirically, on TRAVELPLANNER, NATURALPLANS, and τ-BENCH, this design yields higher end-task success and lower variance while keeping verification overhead bounded (see §4.1).

B ADDITIONAL IMPLEMENTATION DETAILS

DVP Learning To implement the verification policy π_{verify}, we optimize a domain-specific rulebook using discrete prompt optimization via Monte Carlo Tree Search (MCTS). At each iteration (Algorithm 1), the task model simulates execution under the current rulebook, an LLM-based optimizer analyzes failures and proposes edits, and a reward function evaluates constraint satisfaction. The algorithm refines the prompt through guided feedback loops, gradually improving its ability to produce grounded and correct runtime decisions without relying on hand-crafted heuristics or model fine-tuning.

VERA To integrate both pre-execution validation and during-execution control, we formulate a unified algorithm (Algorithm 2) that first applies Static Verification via Rubrics (SVR) to filter ill-formed plans and then executes the verified plan under a Dynamic Verification Policy (DVP). Given a user query, SVR generates a task-specific rubric and filters candidate plans by checking for completeness, correctness, and executability. Only plans that pass all checklist items are admitted. During execution, DVP monitors each step by comparing tool outputs to expected constraints and selects explicit control actions—such as accepting, skipping, retrying with alternative tools, or backtracking—based on a learned, prompt-optimized rulebook. This two-phase framework ensures that plans are not only structurally sound at admission but also remain robust and reactive under runtime uncertainty.

C QUALITATIVE ANALYSIS OF DVP

NaturalPlans: Temporal Structure and Plan Repair. In the NATURALPLANS setting, VERA relies on DVP to enforce strict temporal formatting and maintain structural correctness during execution. For instance, it detects violations of the `F0. STRICT HEADER` rule—e.g., plans that begin with incorrect headers or mismatch the city count—and emits a `backtrack` signal to trigger header repair. Additionally, DVP identifies arithmetic inconsistencies in visit blocks (via `C2. CONTIGUOUS DAY RANGES`) where day spans, flight transitions, or final day coverage are misaligned. Rather than discarding the entire plan, DVP corrects only the affected segments, ensuring that well-formed prefixes are preserved. These fine-grained runtime interventions produce localized, structure-preserving edits that static SVR may overlook, especially in longer, partially valid itineraries.

TauBench: Dialogue Policy and API Verification. In τ-BENCH, DVP plays a crucial role in dialogue management and tool-grounded execution. It enforces task isolation as prescribed by `1.1 Single Task Focus`, prompting the user to confirm intent when new requests interrupt ongoing ones—e.g., asking to rebook while canceling. Such interventions are realized via `alt_operator` or `skip_step` control actions, which guide the conversation back to a valid interaction state. Furthermore, in booking flows (`3.1–3.5`), DVP validates that the agent (i) collects all required information (e.g., user ID, trip type, baggage count), (ii) confirms booking with the user before issuing the API call, and (iii) checks the returned fields for mismatches. When discrepancies arise, DVP issues `backtrack` or `next_result` signals to retry or escalate, enforcing precise API-grounded behavior beyond what ReAct-style prompting can achieve.

TravelPlanner: Budget-Aware Control and Operator Switching. In TRAVELPLANNER, DVP ensures budget feasibility and step validity through forward-looking control. When transport steps violate `Q1.0 TRANSPORT-COST GUARD`—e.g., when a flight would consume remaining budget required for hotels—DVP emits an `alt_operator` signal to prefer cheaper options such as driving or taxis. For meals, DVP uses `Q4.2` to skip steps that fall outside valid time windows or violate soft precedence rules. Hotel decisions are guided by rules such as `Q3.3 OCCUPANCY FLEX`, where DVP triggers paging or backtracking when listings fail minimum-night or pricing constraints. Additionally, the `Q7.1 Budget Aggregator` prevents downstream budget violations by anticipating cumulative costs and selecting `next_result` to explore cheaper alternatives. These dynamic adjustments—rooted in learned, prompt-optimized rulebooks—enable robust, constraint-respecting execution without hand-coded heuristics.

Examples of Dynamic Verification Policies for Different Datasets

TRAVELPLANNER

```
<Flights / Long-distance Transport>:
Q1.0  TRANSPORT-COST GUARD
      Let  R     = remaining_budget   (before booking this leg)
           Hmin  = min(cost of all still-unbooked **mandatory** steps such as accommodation
    and commute back; non-mandatory steps: meals and attractions)
                      ⇒ flights, long-haul drives, first hotel)
           D_est = estimated cost of the current origin-dest
                   **self-drive / taxi** leg
                   (estimate if alt price unknown).
      If  (candidate_cost × travellers  >  R − Hmin)            OR   # would break budget
          (candidate_cost × travellers  > 2 × D_est):                  # ≥ 2× cheaper alt
         ⇒  "alt_operator"      # ask executor to try Drive / Taxi first
      truncated
<Meals:>:
Meals can be skipped when (1) in lower precedence with Flights and Self-Driving or Taxi; (2)
    Skipped to make room for accommodation and other budgets as well.
truncated
Q3.2 Is a concrete or inferred meal_time available?
       Skip if the derived time falls outside the canonical window.
<Hotel Check-in / Stay>:
Q3.3  OCCUPANCY FLEX FOR ROOM-TYPE LISTINGS
      • If the budget or min-night test fails, continue paging /
        back-tracking exactly as in Q3.0.

< Budget Aggregator>:truncated
Q7.0 First, determine the number of travellers mentioned (exclude kids if not specified or
    ambiguous).
Q7.1 Is Σ(daily_costs) ≤ total_trip_budget?
      → If FALSE and cheaper alternative exists for *current* soft step   ⇒ "next_result".
```

NATURALPLANS

```
<Format & Template: High Priority>
F0. STRICT HEADER
    PASS if the very first line equals:
         "Here is the trip plan for visiting the {N} European cities for {D} days:"
         with N, D matching the task; blank line follows.
    Else FAIL → backtrack (fix header).
F1. VISIT BLOCK SYNTAX
<Coverage & Day Arithmetic: High Priority>
truncated
C2. CONTIGUOUS DAY RANGES
    Let total days = D. Let the first block be s=1. For each block i with length k_i:
      • Visit line must be "**Day s-e:** ... for k_i days." where e = s + k_i − 1.
      • The following flight line must be "**Day e:** Fly from Prev to Next."
      • The next block must start at day s' = e (flight day equals the first day of the
    destination block).
    PASS if all blocks satisfy the recurrence and the final e equals D.
    Else FAIL → backtrack (repair day math).
```

τ-BENCH

```
1. INTERACTION MANAGEMENT
1.1 Single Task Focus
  • Only work on one primary task at a time (booking, modifying, cancelling, compensation).
  • If the user makes a new request mid-task, ask:
    ''You've asked to [new request] while we're working on [current task]. Would you like to
    pause [current task] to handle [new request] now, or finish first?''
    truncated
3. BOOKING FLIGHTS
3.1 Information Gathering (in order)
  1. Confirm User ID.
  2. Trip type (one-way or round-trip).
  8. Ask how many checked bags are needed.
3.5 Confirmation & Booking
  • Summarize all booking details and payment plan.
  • Ask ''Do you confirm booking with these details?''
  • Upon explicit ''yes,'' call the booking API.
  • After API returns, compare each returned field to requested values. If mismatch, apologize
    , correct, and retry or escalate.
```

Table 4: **Case studies of Dynamic Verification Policy (DVP) behavior across domains.** Each example shows how DVP responds to execution-time deviations by issuing control signals such as `backtrack`, `alt_operator`, or `next_result`. In TRAVELPLANNER, DVP prevents budget overflows by switching transport modes or skipping low-priority steps. In NATURALPLANS, it enforces formatting and day arithmetic (e.g., fixing header and visit block errors). In τ-BENCH, DVP corrects API misuse, reorders dialogue turns, or prompts clarification when tasks conflict. These examples highlight how prompt-optimized verifier rulebooks enable grounded, context-sensitive policy control at runtime.

D ADDITIONAL EXPERIMENTAL DETAILS

Feedback Collection Prompt. Below we present the prompt to generate plans.

```
You are a planner whose job is to break down a given user request into a
    set of small/atomic steps to finalize the user query. For example,
    given a complex query in natural language, you need to indentify the
    set of steps, constrained in user queries and break it down into
    smaller steps. The expected output format is provided at the end.

For planning purposes, you will be given following as an input:
(1) An optional list of APIs/tools/operators or agents which can be used
    to execute the atomic steps.
(2) The user query
(3) An optional set of constraints. If not provided, you may chose to
    skip this field.
(4) An optional plan schema which you can use to output the plan. If none
     provided, please output in a natural language set of steps.

### Start of APIs/Tools/Operators ###
{api_tools}
### End of APIs/Tools/Operators ###

The input query is:
### Begin User Query ###
{user_query}
### End User Query ###

### Start of Constraints ###
{constraints}
### End of Constraints ###

Your job is to output a plan in the following format:
{plan_schema}.

Expected outputs:
1) First, provide your step by step analysis using "Let's think step by
    step." to tackle the problem. Wrap your analysis in <analysis>...</
    analysis> tags.
2) Second, output any user constraints in the field wrapped inside tags <
    constraints>...</constraints>
3) Finally, output your plan wrapped inside tags <PLAN>...</PLAN> tags.

Always, strictly follow the provided output structure.
```

Rubric Generation Prompt. Below, we present the prompt to generate rubrics.

```
You are an expert in designing rubrics for grading any domain-specific
    tasks and your job is to design strict domain-specific rubrics based
    on few samples covering critical and necessary aspects of a domain.
    You need to generate exactly {num_rubrics} rubrics for each category.
     Remember, some of the taxonomy elements provided below are only
    applicable to structured plans or specific domains. Use your best
    judgment to pick the taxonomy elements for rubrics generation. The
    examples are as follows:

The examples to learn rubrics are as follows:
### Begin Query ###
{example_string}
### End Query ###

Please consider the following taxonomy for rubrics generation (consider
    only which are applicable)"
### Begin Taxonomy ###
{taxonomy}
### End Taxonomy ###
```

```
In addition, here are some task-specific constraints you should also
    consider:

### Begin Constraints ###
{additional_constraints}
### End Constraints ###

Please craft a domain-specific rubric, wrapped within tags <rubrics>...</
    rubrics> for plan verification in the following format:
```
{{
    "Parent_Category_1":
        {{
            "Child-Category 1" : "Rubric - 1 ",
            "Child-Category 2" : "Rubric - 2 ",
            ...
        }}
    ...
}}
```

Expected outputs:
1) First, provide your step by step analysis using "Let's think step by
    step." to tackle the problem. Wrap your analysis in <analysis>...</
    analysis> tags.
2) Second, output any user constraints that needs to be included in
    rubrics in the field wrapped inside tags <constraints>...</
    constraints>. You can also include any additional constraints for a
    viable plan.
3) A very important thing to consider and remember is that the process is
     for validation so the execution results are not provided in the plan
    . The plan will be sent for execution once the validation scores
    enough basd on your scores hence, do not expect any outputs yet!
4) Finally, output your rubrics wrapped inside tags <rubrics>...</rubrics
    > tags.
5) Do not ever change the name of the rubric items.

Always, strictly follow the provided output structure.
```

Rubric Generation Prompt. Below, we present the prompt to generate judge scores and feedback.

```
You are a judge whose job is to critically analyze and categorically
    analyze the user plan and the taxonomy and assign a binary score (0
    or 1) to every rubric item. Remember to only base your judgment on
    the plan, not the query itself, and to be very strict and specific.
    The information from the user query should not be used for judging
    and is for reference and grounding purposes only. You only need to
    judge based on the plan, not the user query and present actionable
    items at the end. A very important thing to consider and remember is
    that the process is for validation so the execution results are not
    provided in the plan. The plan will be sent for execution once the
    validation scores enough basd on your scores hence, do not expect any
     outputs yet!

The user query is:
### Begin User Query ###
{user_query}
### End User Query ###

Below are the set of APIs/tools/api/operators which you can be used for
    planning (we use APIs, Tools, and Operators interchangeably):
### Begin API/Tools/Operators ###
{api_tools}
```

```
### End API/Tools/Operators ###

The plan you have to verify is:
### Start of Proposed Plan ###
{plan}
### End Proposed Plan ###

The plan schema is:
### Begin Plan Schema ###
{plan_schema}
### End Plan Schema ###

The rubrics are as follows --- score only these, do not invent new ones:
### Begin Rubrics ###
{rubrics}
### End Rubrics ###

You can consider the following to understand related plans which are
    correct and will score high for similar queries:
### Begin Demonstrations ###
{icl}
### End Demonstrations ###

Return your result wrapped in <scores> ... </scores> tags, in the
    following format:
```
{{
  "Parent_Category_1": {{
    "Child-Category 1": {{
      "reasoning": "your detailed analysis of the plan",
      "score": 0/1
    }},
    "Child-Category 2": {{
      "reasoning": "your detailed analysis of the plan",
      "score": 1/0
    }}
  }},
  "Parent_Category_2": {{
    ...
  }},
  "Overall": {{
    "Overall Impression of the plan and actionable items to improve the
    plan": "provide your overall detailed analysis of the plan and in
    addition unambiguous and clear feedback or ationable items on how to
    improve it based on other judges' feedbacks.",
    "Overall score of the plan": "{{overall_score_percent}}"
  }}
}}

```

Expected outputs:
1) First, provide your step by step analysis using "Let's think step by
    step." to tackle the problem. Wrap your analysis in <analysis>...</
    analysis> tags.
2) Second, output any user constraints that needs to be used for rubric
    verification in the field wrapped inside tags <constraints>...</
    constraints>. You can also include any additional constraints for a
    viable plan.
3) Finally, output your scores wrapped inside tags <scores>...</scores>
    tags with **a valid JSON object.** Remember "overall score" should be
     a number, don't even add % or any other character.
4) Do not ever change the name of the rubric items.

Always, strictly follow the provided output structure.
```

Rubric Generation Prompt. Below, we present the prompt to generate new plans based on judge scores and feedback.

```
You are a planner whose job is to break down a given user request into a
    set of small/atomic steps to finalize the user query and improve over
     a plan which was given a low score based on judges' feedback. For
    example, given a complex query in natural language, you need to
    indentify the set of steps, constrained in user queries and break it
    down into smaller steps while keeping feedback in mind. The expected
    output format is provided at the end.

For planning purposes, you will be given following as an input:
(1) An optional list of APIs/tools/operators or agents which can be used
    to execute the atomic steps.
(2) The user query
(3) The current (low-scored) plan you need to modify.
(4) The feedback.
(5) An optional set of constraints. If not provided, you may chose to
    skip this field.
(6) An optional plan schema which you can use to output the plan. If none
     provided, please output in a natural language set of steps.
(7) Previous failed attempts which leads to incorrect outputs

### Start of APIs/Tools/Operators ###
{api_tools}
### End of APIs/Tools/Operators ###

The input query is:
### Begin User Query ###
{user_query}
### End User Query ###

The current plan is:
### Start of Plan to modify ###
{current_plan}
### End of Plan to modify ###

The feedback:
### Begin Feedback ###
{feedback}
### End Feedback ###

You can consider the following to understand related plans which are
    correct and will score high during validation:
### Begin Demonstrations ###
{icl}
### End Demonstrations ###

### Start of Constraints ###
{constraints}
### End of Constraints ###

Your job is to output a plan in the following format:
### Begin Plan Schema ###
{plan_schema}.
### End Plan Schema ###

### Start Previous Failed Attempts ###
{previous_attempts}
### End Previous Failed Attempts ###

Expected outputs:
```

```
1) First, provide your step by step analysis using "Let's think step by
    step." to tackle the problem. Wrap your analysis in <analysis>...</
    analysis> tags.
2) Second, output any user constraints in the field wrapped inside tags <
    constraints>...</constraints>
3) **Finally, always output your plan wrapped inside tags <PLAN>...</PLAN
    > tags.** Always, strictly follow the provided output structure. """
```