Aligning LLM-Generated Tasks with Physical Executability in Grounded Environments

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly used to generate task instructions for grounded agents, yet linguistic fluency does not guarantee physical or operational feasibility. We reveal an *executability gap*: LLMs often produce instructions that sound plausible but violate environment constraints (e.g., nonexistent entities or invalid preconditions). Mechanistically, we observe a *layer-wise preference shift* where early representations favor grounded candidates, while deeper layers increasingly promote linguistically coherent but less constrained continuations. We then introduce a *constraint-aware evaluation* protocol with execution-verifiable constraints, and propose a *training-free* generation-time intervention that injects execution-aware constraints without retraining the downstream agent. Across code, tool-use, and embodied benchmarks, this simple adjustment consistently improves executability and task success (e.g., VirtualHome executability 41.3→46.0, correctness 42.6→47.0). Our results suggest that aligning generation with environment-verifiable constraints is a key bottleneck for grounded task generation.

1 Introduction

Large language models (LLMs) have become a central interface for specifying goals and procedures for grounded agents, ranging from code generation and tool invocation to embodied planning in simulated or physical environments.(Huang et al., 2022a)(Schick et al., 2023)(Ahn et al., 2022)(Chen et al., 2021a) In many applications, an LLM is expected to translate a user intent into a concrete task description or an executable workflow, which is then carried out by a downstream executor (e.g., a compiler, an API runtime, or an embodied environment). While modern LLMs can produce instructions that are fluent, coherent, and syntactically well-formed, grounded execution imposes an additional requirement that is easy to overlook: the generated task must be *feasible* under environment-defined constraints.

Despite the apparent simplicity of this requirement, we observe a recurring failure mode in practice: LLMs frequently generate tasks that *sound right* but are *execution-invalid*(Huang et al., 2025)(Zhou et al., 2024)(Liu et al., 2022). Such instructions may reference nonexistent entities, call unavailable tools, violate interface specifications, or break prerequisite and state-transition rules (e.g., using an object before acquiring it, or placing an item into a container that has not been opened). These errors are not merely corner cases. They persist across task domains and remain noticeable even for large-scale models, suggesting that the bottleneck is not only downstream reasoning, but also a misalignment between language-driven generation and environment-grounded feasibility. We argue that this phenomenon can be understood from a constrained-generation perspective(Lu et al., 2021). Let x denote a generated task (instruction, program, or action sequence) conditioned on a context c, and let $\mathcal{F}(c)$ be the set of feasible tasks determined by execution semantics and environment constraints. Standard decoding implicitly seeks high-probability sequences under the language model $p_\theta(x \mid c)$, yet it does not explicitly enforce $x \in \mathcal{F}(c)$. As a result, the model may allocate substantial probability mass to linguistically frequent patterns that are weakly grounded, while feasible solutions occupy only a small portion of the distribution. In other words, improving linguistic modeling (e.g., scaling) does not necessarily increase the probability mass over $\mathcal{F}(c)$, because feasibility is defined by external semantics rather than surface-level plausibility(McKenzie et al., 2024). This tension also manifests internally: we find evidence that model preference can drift from constraint-compatible candidates toward more fluent continuations as representations be-

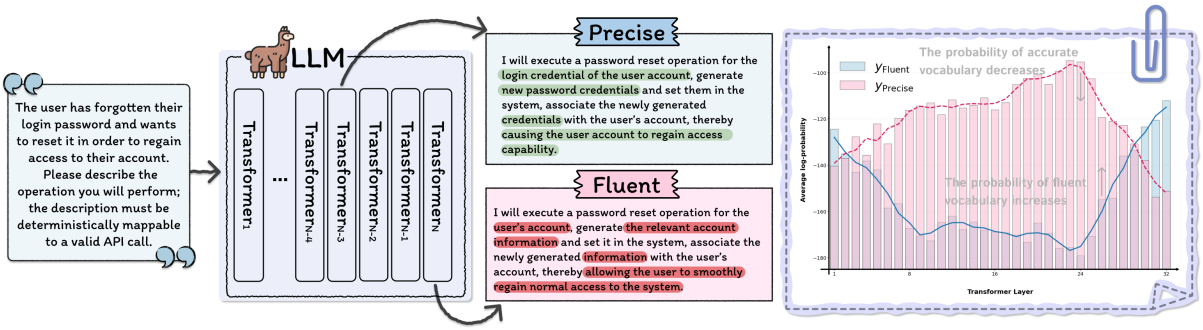

Figure 1: This figure illustrates how model preference shifts across layers between two types of lexical candidates: physically or operationally grounded tokens and linguistically fluent but less constrained ones. In early layers, the model favors tokens that better satisfy execution-level constraints, whereas in deeper layers it increasingly prefers more fluent expressions. This transition highlights a gradual shift from constraint-aware reasoning to language-driven coherence in the model's internal representations.

come deeper, indicating a systematic bias toward linguistic coherence when constraints are not made explicit at generation time.

Motivated by this view, our key insight is simple: grounded task generation should treat feasibility constraints as first-class signals during generation, rather than hoping they emerge from generic language modeling.(Huang et al., 2022b) Concretely, we (i) introduce a constraint-aware evaluation protocol that augments instances with execution-verifiable constraints, enabling a principled measurement of constraint compliance and executability; and (ii) propose a training-free, minimal generation-time intervention that injects execution-aware constraints without retraining the downstream agent.(Li et al., 2020) This design directly targets the mismatch between "what sounds plausible" and "what is executable", and can be plugged into diverse grounded settings.

Our contributions are summarized as follows:

- We identify and systematically characterize an *executability gap* in grounded task generation, where fluent instructions can be execution-invalid under environment constraints.
- We provide a mechanistic perspective suggesting a drift toward linguistic coherence when constraints are not enforced, offering an explanation for why scaling alone may not resolve infeasibility.
- We propose a constraint-aware evaluation protocol with execution-verifiable constraints, and a training-free generation-time constraint injection method that improves executability and task success across grounded benchmarks.

2 Related Works

2.1 LLMs for Grounded Task Generation

LLMs are increasingly deployed as high-level task generators for grounded agents, spanning program synthesis, tool-use orchestration, and embodied planning(Rozière et al., 2024)(Wang et al., 2023). In these settings, the model must translate user intent into an *executable* artifact (code, API-call traces, or action sequences) that is consumed by a downstream executor. While recent progress in instruction tuning and scaling improves fluency and local coherence, a persistent failure mode is the *executability gap*: generations can be linguistically plausible yet invalid under environment-defined semantics, e.g., violating API schemas, calling tools out of order, referencing nonexistent entities, or breaking state-dependent preconditions. Prior studies typically treat such errors as hallucinations or insufficient reasoning. In contrast, our work emphasizes a mechanistic perspective: even when constraints are present in context, feasibility-consistent preferences can be attenuated along depth, motivating explicit constraint signals during generation.

2.2 Training-Time Constraint Alignment

A common approach to improve constraint compliance is to embed feasibility into the model via training-time alignment, including instruction fine-tuning, domain adaptation, and preference-based optimization (e.g., RLHF-style objectives(Ouyang et al., 2022)(Qin et al., 2023)) with constraint-aware feedback. These methods can internalize frequent or well-specified constraints, but face two practical bottlenecks in grounded environments. First, retraining and repeated alignment

are costly, which limits rapid iteration when tools, APIs, or environment dynamics change. Second, grounded constraints are often *combinatorial and non-stationary*: feasibility depends on evolving states, temporal ordering, and hidden preconditions, making it difficult for a fixed training distribution to cover the long tail of runtime configurations. As a result, purely training-based alignment can overfit to observed constraint patterns and degrade when the executor or environment shifts.

2.3 Inference-Time Constraint Enforcement and Verifier-in-the-Loop Decoding

Complementary to training-time alignment, a growing line of work enforces feasibility at inference by coupling decoding with external structure or feedback, such as grammar-/schema-constrained decoding for code and tool calls(Scholak et al., 2021)(Poesia et al., 2022), compiler/interpreter checks, symbolic validators, or lightweight simulators for embodied action feasibility(Ni et al., 2023). These approaches share the principle of *executor-grounded verification*: using environment-verifiable signals to prune or re-rank candidate continuations. EXECGATE follows this paradigm but differs in two key aspects. First, it performs *token-level, online* constraint injection with a soft gating mechanism, approximating constrained decoding as a product-of-experts between the LLM distribution and an environment verifier. Second, to keep latency practical, it adopts a verify-on-demand strategy over a small top-K candidate set, enabling broad applicability across code, tool-use, and embodied settings without retraining the downstream agent.

3 Methodology

In this section, we formally address the challenge of aligning Large Language Model (LLM) (Llama Team, 2024)generation with physical executability. We first formulate the grounded task generation problem as a constrained decoding objective in §3.1. In §3.2, we investigate the internal mechanisms of LLMs, identifying a phenomenon we term *Layer-wise Preference Drift*, where models progressively lose track of constraints in deeper layers. Motivated by these findings, we introduce EXECGATE in §3.3, a generation-time intervention that injects execution-aware constraints via probabilistic gating. Finally, we provide a theoretical interpretation of our method and detail its domain-specific instantiations in §3.5.

3.1 Preliminaries: The Constrained Decoding Objective

Autoregressive Generation. Let c denote the generation context, which encompasses the user request and environment-specific information such as API definitions, object inventories, and state descriptors. A standard LLM generates a sequence $x = (x_1, \ldots, x_T)$ autoregressively, modeling the probability of the next token x_t given the prefix $x_{<t}$:

$$p_\theta(x \mid c) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}, c), \tag{1}$$

where θ represents the model parameters. At each step, the model produces logits $z_t \in \mathbb{R}^{|\mathcal{V}|}$ over the vocabulary $\mathcal{V}$, converting them to probabilities via the softmax function.

The Executability Gap. While standard LLMs optimize for linguistic fluency, grounded environments impose strict, environment-verifiable constraints. We formalize these constraints as a set of predicates $\{\phi_j\}_{j=1}^{m}$, where $\phi_j(x, c) \in \{0, 1\}$ indicates whether sequence x satisfies the j-th constraint (e.g., entity existence, API schema compliance). The *feasible set* $\mathcal{F}(c)$ is defined as the collection of sequences that satisfy all constraints:

$$\mathcal{F}(c) = \{x \mid \forall j \in [m],\ \phi_j(x, c) = 1\}. \tag{2}$$

Standard decoding implicitly seeks high-probability sequences under $p_\theta(x|c)$ but does not explicitly enforce $x \in \mathcal{F}(c)$. This leads to an *executability gap*: models often allocate substantial probability mass to linguistically plausible but execution-invalid sequences.

Ideal Constrained Objective. Ideally, grounded generation should sample from the target posterior $p^\star(x|c)$, which reweights the base model distribution by a feasibility indicator $\mathbb{I}_{\text{feas}}(x, c) = \prod_j \phi_j(x, c)$:

$$p^\star(x \mid c) \ \propto\ p_\theta(x \mid c) \cdot \mathbb{I}_{\text{feas}}(x, c). \tag{3}$$

Solving Eq. 3 directly is intractable due to the vast search space. EXECGATE aims to approximate this objective efficiently during inference.

3.2 Analysis Motivation: Layer-wise Preference Drift

Why do powerful models fail to respect $\mathbb{I}_{\text{feas}}$ implicitly? We hypothesize that the tension between

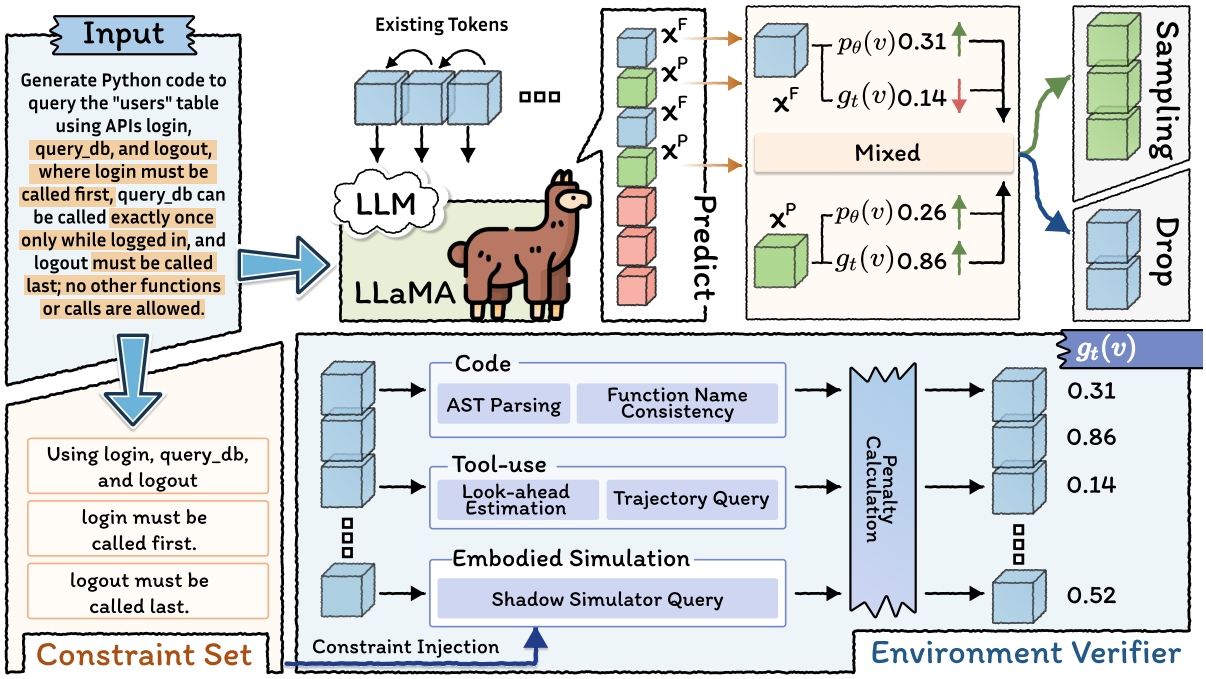

Figure 2: Given an input with explicit execution-grounded constraints, the LLM proposes candidate next tokens at each step. An environment verifier—instantiated per domain (e.g., AST parsing for code, trajectory/lookup checks for tool use, or shadow simulation for embodied tasks)—incrementally evaluates candidates and produces a gating score $g_t(v)$ / penalty $r_t(v)$. These signals are fused with the base model probability $p_\theta(v)$ to reweight the decoding distribution, so sampling favors constraint-satisfying continuations while pruning infeasible ones.

"linguistic plausibility" and "physical feasibility" manifests internally within the transformer layers.

To verify this, we employ a logit-lens analysis. Let $h_t^{(\ell)}$ be the hidden state at layer ℓ. We decode the intermediate token distribution $p_t^{(\ell)}$ by projecting $h_t^{(\ell)}$ via the unembedding matrix W_U. We compare the model's preference for two distinct types of candidates:

1. **Precise (x^P):** A constraint-compatible candidate (grounded).
2. **Fluent (x^F):** A linguistically coherent but weakly grounded candidate (ungrounded).

We quantify the preference drift via the average log-probability gap $\Delta^{(\ell)}$:

$$\Delta^{(\ell)} = \frac{1}{T}\sum_{t=1}^{T}\left[\log p_t^{(\ell)}(x_t^P) - \log p_t^{(\ell)}(x_t^F)\right]. \tag{4}$$

Insight: As illustrated in our analysis (Figure 1), $\Delta^{(\ell)}$ tends to decrease in deeper layers. This indicates a *preference drift*: early layers favor constraint-aware tokens, but deeper layers increasingly promote linguistically smooth but unconstrained continuations. This suggests that feasibility signals are attenuated by the language modeling objective, necessitating external re-injection at the decoding stage.

3.3 ExecGate: Generation-Time Constraint Injection

To bridge the executability gap, we propose ExecGate, a training-free decoding mechanism. It approximates the posterior $p^\star$ by modulating the next-token distribution with an explicit feasibility gate. The process follows Algorithm 1.

Efficient Incremental Verification (Lines 3–5, Alg. 1). Evaluating constraints over the entire vocabulary is computationally prohibitive. We adopt a *verify-on-demand* strategy. At step t, we restrict our attention to the top-K candidates $\mathcal{K}$ proposed by the base model. For each candidate $v \in \mathcal{K}$, we run an environment verifier on the incremental prefix $(x_{<t} \oplus v)$ to check for constraint violations. We define a cumulative penalty score $r_t(v)$ based on the weighted sum of violations:

$$r_t(v) = \sum_{j=1}^{m} w_j \left(1 - \phi_j(x_{<t} \oplus v, c)\right), \tag{5}$$

where $w_j \geq 0$ controls the importance of constraint j.

Soft Gating and Hard Masking (Line 6, Alg. 1). We transform the penalty into a probabilistic gate $g_t(v)$. For deterministic constraints (e.g., syntax or tool availability), we apply a hard mask to eliminate provably invalid candidates. For soft or partial constraints, we use an exponentially decaying weight:

$$g_t(v) = \exp\big(-\lambda\, r_t(v)\big) \cdot \prod_{j \in \mathcal{J}_{hard}} \phi_j(x_{<t} \oplus v, c), \tag{6}$$

where λ acts as an inverse temperature parameter regulating the strictness of the constraints.

Posterior Approximation (Lines 8–9, Alg. 1). The final distribution $\tilde{p}_t$ is obtained by reweighting the base logits with the gate:

$$\tilde{p}_t(v) = \frac{p_\theta(x_t = v \mid x_{<t}, c) \cdot g_t(v)}{Z_t}, \tag{7}$$

where Z_t is the partition function. This mechanism implements an online approximation to the constrained posterior Eq. 3, effectively "correcting" the preference drift observed in standard models.

3.4 Theoretical View

Our formulation in Eq. 7 can be rigorously interpreted through the lens of Energy-Based Models (EBMs). Let the energy of a token v be defined as $E(v) = E_{\text{LM}}(v) + \lambda E_{\text{Phys}}(v)$, where $E_{\text{LM}}(v) = -\log p_\theta(v)$ represents the linguistic cost, and $E_{\text{Phys}}(v) = r_t(v)$ represents the physical violation cost. EXECGATE effectively samples from the Gibbs distribution $p(v) \propto e^{-E(v)}$. This perspective highlights that our method is not merely a heuristic, but a principled Product-of-Experts approximation combining a learned linguistic expert (p_θ) and a symbolic physical expert (ϕ).

3.5 Instantiation and Verifier Implementation

While the EXECGATE framework is general, the concrete mechanism of the verifier ϕ_j depends on the domain-specific state representation. We detail the implementation for three key settings:

Code Generation (Static Analysis): Here, ϕ_j acts as an incremental parser. For a candidate token v, we form the partial code $x_{<t} \oplus v$. The verifier checks if this partial code violates the grammar of the target language (e.g., Python). Crucially, we employ **incremental abstract syntax tree (AST) parsing** to detect syntax errors early (e.g., unclosed parentheses, indentation errors) and API schema matching to ensure function arguments align with defined signatures (e.g., preventing a string argument where an integer is required).

Algorithm 1 EXECGATE: Generation-Time Constraint Injection

Require: context c, model p_θ, constraints $\{\phi_j\}$, weights $\{w_j\}$, strength λ, candidate size K

1: Initialize $x_{<1} = \emptyset$
2: **for** $t = 1$ to T **do**
3: Compute base probs $p_\theta(\cdot \mid x_{<t})$ and select $\mathcal{K} \leftarrow \text{TopK}(p_\theta, K)$
4: **for** each candidate $v \in \mathcal{K}$ **do**
5: Evaluate constraints $\phi_j(x_{<t} \oplus v, c)$
6: Compute penalty $r_t(v) = \sum_j w_j(1 - \phi_j)$ ▷ Eq. 5
7: Compute gate $g_t(v) = \exp(-\lambda r_t(v))$ ▷ Eq. 6
8: Reweight $\tilde{p}_t(v) \propto p_\theta(v) \cdot g_t(v)$ ▷ Eq. 7
9: Sample $x_t \sim \tilde{p}_t$ and update prefix $x_{<t+1} \leftarrow x_{<t} \oplus x_t$
10: **return** Sequence x

Tool-Use Planning (Schema Validation): The constraints are derived from the API documentation. When the model generates a tool call (e.g., 'Tool(arg=...'), ϕ_j performs a **lookahead check**. If the candidate v starts an argument value, the verifier checks it against the allowed type enum or range defined in the schema. For dependency constraints (e.g., "login before query"), ϕ_j maintains a lightweight execution trace; if 'query' is proposed but 'login' is absent from the trace, $\phi_{order}(x_{<t} \oplus v)$ returns 0, triggering a penalty.

Embodied Planning (State Simulation): This setting requires dynamic state tracking. We run a **shadow simulator** that mirrors the environment state. For an action candidate v (e.g., 'grab(apple)'), ϕ_j queries the simulator state: *Is the agent reachable to the apple? Is the agent's hand empty?* If the preconditions are not met in the current simulated state, the action is marked as physically infeasible ($\phi = 0$), and the penalty r_t is applied immediately, preventing the agent from hallucinating impossible moves.

4 Experiments

4.1 Experiment Setup

Benchmark To evaluate the ability of the model to generate executable workflows under constraint guidance, we consider a set of downstream tasks covering code generation, tool usage, and embod-

Table 1: **Performance comparison across different backbone models.** Results are reported on code generation, tool-use, and embodied reasoning benchmarks. Improvements over the base model are marked in red.

Model	HumanEval		MBPP		API-Bank	API-Bench	Nexus	VirtualHome		
	HE	HE+	MBPP	MBPP+				Exec	LCS	Corr
Qwen2.5-7B	52.8	46.2	73.6	59.6	80.6	32.4	48.7	64.2	24.3	61.3
+ EXECGATE	**54.7**$_{+1.9}$	**49.6**$_{+3.4}$	**74.8**$_{+1.2}$	**61.4**$_{+1.8}$	**81.5**$_{+0.9}$	**34.0**$_{+1.6}$	**49.1**$_{+0.4}$	**65.8**$_{+1.6}$	**25.6**$_{+1.3}$	**63.7**$_{+2.4}$
Qwen2.5-14B	54.6	48.6	76.4	61.8	82.7	36.3	52.3	71.4	28.6	68.2
+ EXECGATE	**56.2**$_{+1.6}$	**51.9**$_{+3.3}$	**77.3**$_{+0.9}$	**63.0**$_{+1.2}$	**84.0**$_{+1.3}$	**37.5**$_{+1.2}$	**52.9**$_{+0.6}$	**73.7**$_{+2.3}$	**30.7**$_{+2.1}$	**71.4**$_{+3.2}$
Qwen2.5-72B	56.3	50.8	81.9	68.3	89.6	39.6	56.7	76.8	32.3	74.3
+ EXECGATE	**57.8**$_{+1.5}$	**53.4**$_{+2.6}$	**82.6**$_{+0.7}$	**69.7**$_{+1.4}$	**90.2**$_{+0.6}$	**40.4**$_{+0.8}$	**57.0**$_{+0.3}$	**78.9**$_{+2.1}$	**34.0**$_{+1.7}$	**77.1**$_{+2.8}$
LLaMA3-8B	32.6	27.6	53.4	42.4	70.4	9.1	35.6	46.2	18.4	49.3
+ EXECGATE	**34.8**$_{+2.2}$	**30.4**$_{+2.8}$	**54.8**$_{+1.4}$	**44.5**$_{+2.1}$	**71.6**$_{+1.2}$	**12.9**$_{+3.8}$	**36.5**$_{+0.9}$	**48.1**$_{+1.9}$	**20.0**$_{+1.6}$	**51.4**$_{+2.1}$
LLaMA3-70B	46.2	41.8	67.2	56.2	76.4	28.4	43.7	58.4	22.6	54.2
+ EXECGATE	**48.2**$_{+2.0}$	**45.1**$_{+3.3}$	**68.4**$_{+1.2}$	**58.0**$_{+1.8}$	**77.8**$_{+1.4}$	**30.3**$_{+1.9}$	**44.3**$_{+0.6}$	**60.4**$_{+2.0}$	**24.4**$_{+1.8}$	**56.6**$_{+2.4}$

ied execution. For code generation, the model is required to produce executable programs from natural language descriptions, where violations of syntax rules, API specifications, or implicit constraints result in execution failure. We evaluate this setting using **HumanEval**(Chen et al., 2021b) and **MBPP**(Austin et al., 2021), as well as their constrained variants, **HumanEval+** and **MBPP+**, which introduce stricter constraints to assess the model's compliance ability. For tool usage tasks, the model must generate sequences of tool invocations that satisfy interface specifications and dependency constraints; errors such as incorrect tool selection, parameter mismatch, or invalid invocation order lead to task failure. We evaluate this setting on **API-Bank**(Li et al., 2023),**API-Bench**(Patil et al., 2023) and **Nexus**(team, 2023). For embodied tasks, the model is required to generate executable action sequences under explicit physical and temporal constraints. We follow the experimental setup of Wenlong Huang et al(Huang et al., 2022a). and adopt the VirtualHome platform (Puig et al., 2020) to evaluate whether the generated plans satisfy constraints related to object existence, spatial relations, and action preconditions.

Baseline To systematically evaluate the generalizability of **EXECGATE** across different model architectures, we select two representative open-source foundation models, **Qwen2.5**(Qwen et al., 2025) and **LLaMA 3**(Llama Team, 2024), as comparative backbones and apply EXECGATE under a unified experimental setting. Beyond cross-architecture evaluation, we further assess the robustness of EXECGATE across models of varying parameter scales. Specifically, we consider three model sizes (7B, 14B, and 72B) within the Qwen2.5 family and two model sizes (8B and 70B) within the LLaMA 3 family. This design enables a comprehensive analysis of the transferability and general effectiveness of EXECGATE along both architectural diversity and model scale dimensions.

4.2 Main Results

As shown in the results(Table 1), our proposed **EXECGATE** consistently achieves stable and consistent performance gains across different backbone architectures (Qwen2.5 and LLaMA 3) and model scales, demonstrating strong transferability and generalization capability. Specifically, EXECGATE yields consistent improvements on standard code generation benchmarks (HumanEval and MBPP), and achieves even more pronounced gains on their constraint-enhanced variants (HE+ and MBPP+), such as +3.4 on HE+ for Qwen2.5-7B and +3.3 for LLaMA3-70B. These results indicate that EXECGATE effectively mitigates models' over-reliance on surface-level language patterns and enhances their ability to adhere to explicit constraints during generation.

On tool-oriented benchmarks (API-Bank, API-Bench, and Nexus), EXECGATE also delivers systematic improvements, with particularly notable gains on the more compositional and execution-sensitive API-Bench benchmark (e.g., +3.8 on LLaMA3-8B). This suggests that enforcing execution-aware constraints effectively reduces invalid tool invocations and mismatched action sequences, narrowing the gap between syntactic plausibility and actual executability.

More importantly, in the embodied environment VirtualHome, EXECGATE consistently improves Executability, LCS, and Correctness metrics (typically by +1–3 points across models), directly val-

Table 2: **Performance comparison across different backbone models under explicit constraint-aware settings.** Results are reported on code generation, tool-use, and embodied reasoning benchmarks. Improvements over the corresponding base models are highlighted in red, demonstrating the effectiveness of incorporating execution-level constraints into the generation process.

Model	HumanEval	MBPP	API-Bank	API-Bench	Nexus	VirtualHome		
						Exec	LCS	Corr
Qwen2.5-7B	30.6	42.3	53.6	21.4	31.4	41.3	13.4	42.6
+ EXECGATE	**32.8**$_{+2.2}$	**46.3**$_{+4.0}$	**57.4**$_{+3.8}$	**23.9**$_{+2.5}$	**33.5**$_{+2.1}$	**46.0**$_{+4.7}$	**15.3**$_{+1.9}$	**47.0**$_{+4.4}$
Qwen2.5-14B	34.2	46.2	55.3	24.6	34.2	47.2	16.8	46.8
+ EXECGATE	**37.0**$_{+2.8}$	**50.1**$_{+3.9}$	**58.9**$_{+3.6}$	**26.7**$_{+2.1}$	**36.9**$_{+2.7}$	**51.4**$_{+4.2}$	**17.3**$_{+1.5}$	**50.8**$_{+4.0}$
Qwen2.5-72B	36.8	48.2	59.4	26.8	37.6	51.8	19.6	52.4
+ EXECGATE	**39.2**$_{+2.4}$	**51.8**$_{+3.6}$	**62.6**$_{+3.2}$	**29.2**$_{+2.4}$	**39.9**$_{+2.3}$	**55.7**$_{+3.9}$	**20.7**$_{+1.1}$	**56.0**$_{+3.6}$
LLaMA3-8B	20.6	32.7	41.7	2.6	24.2	31.3	8.4	37.2
+ EXECGATE	**24.2**$_{+3.6}$	**37.1**$_{+4.4}$	**45.7**$_{+4.0}$	**6.1**$_{+3.5}$	**26.4**$_{+2.2}$	**36.1**$_{+4.8}$	**10.0**$_{+1.6}$	**41.7**$_{+4.5}$
LLaMA3-70B	24.8	26.2	32.6	13.4	28.4	43.6	14.8	44.5
+ EXECGATE	**28.0**$_{+3.2}$	**30.3**$_{+4.1}$	**36.1**$_{+3.5}$	**16.5**$_{+3.1}$	**31.2**$_{+2.8}$	**47.9**$_{+4.3}$	**16.2**$_{+1.4}$	**48.7**$_{+4.2}$

idating our central hypothesis: task generation driven purely by language modeling objectives tends to overlook physical and environmental constraints, whereas explicitly injecting execution-aware constraints during generation substantially alleviates this mismatch. As a result, the performance bottleneck shifts from the agent's reasoning capability to the alignment between generated plans and their real-world executability.

4.3 Constraint-Based Evaluation

To systematically evaluate a model's ability to comply with explicit environmental constraints, we construct a constraint-aware evaluation subset derived from the original dataset. Specifically, we randomly sample a subset of instances and augment them with explicitly defined, execution-grounded constraints, while preserving the original task semantics. These constraints are formulated such that they can be directly verified by the execution environment, including but not limited to: restricting available API calls to a predefined set, enforcing the existence of referenced entities in the environment, and requiring action sequences to satisfy prerequisite and state-transition conditions (e.g., an object must be acquired before being placed, or a container must be opened before an item can be retrieved). By comparing model performance under unconstrained and constrained settings, we assess the extent to which model behavior is governed by executable constraints rather than surface-level language patterns. This evaluation protocol enables a principled analysis of whether models can adapt their generation strategies in response to explicit feasibility requirements, thereby providing a more faithful measure of their true reasoning and execution capabilities.

The results(Table 2) indicate that under settings with stronger constraints and stricter executability requirements, the proposed method more effectively encourages models to adhere to explicit constraints, which is consistent with our original motivation. In particular, stronger constraints typically require more structured and fine-grained reasoning to encode prerequisite relations and state dependencies, making baseline models more susceptible to being misled by superficially plausible but execution-invalid patterns.

Taking highly constrained scenarios as an example, on the VirtualHome benchmark, the proposed approach leads to consistent improvements in executability (e.g., from 41.3 to 46.0 on Qwen2.5-7B), while simultaneously improving overall task correctness (from 42.6 to 47.0). Similarly, on API-Bench—where errors frequently arise from invalid tool usage or incorrect action ordering—models with weaker baselines benefit more substantially (e.g., an improvement from 2.6 to 6.1 on LLaMA3-8B). These results demonstrate that by explicitly incorporating execution-aware constraints,

4.4 Ablation Study

To assess the independent contribution of each component in EXECGATE, we conduct component-wise ablations under identical settings—fixing the backbone model, data, prompts, and decoding hyperparameters (e.g., temperature, Top-K, and maximum generation length)—and evaluate all vari-

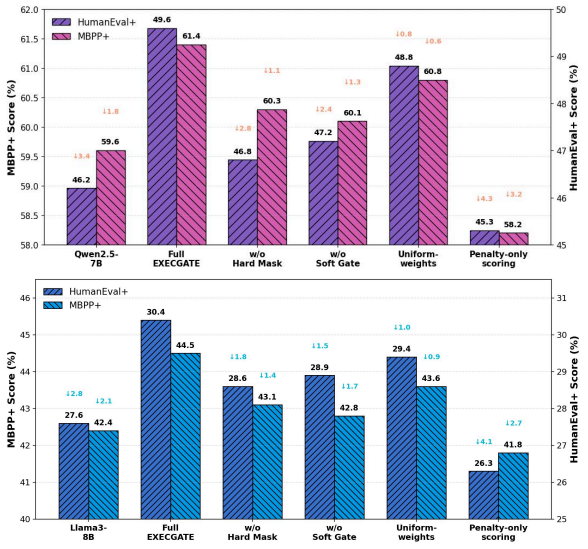

Figure 3: **Ablation results of EXECGATE.** Results on HumanEval+ and MBPP+ with Qwen2.5-7B and LLaMA-3-8B show that combining hard masking and soft gating is critical for performance, while removing either component leads to degradation, underscoring the importance of fusion-based constraint-aware decoding.

ants with the same protocol. **Full EXECGATE** performs constraint verification for candidate tokens at each decoding step to obtain a penalty score $r_t(v)$, and reweights the next-token distribution via *hard mask* plus *soft gate*. We further consider: (1) **w/o Hard Mask (only soft)**: removing the hard pruning and retaining only the continuous penalty term $\exp(-\lambda r_t(v))$; (2) **w/o Soft Gate (only hard)**: applying only hard-constraint filtering without the soft penalty; (3) **Uniform-weights**: using an equal-weight aggregation $r_t(v) = \sum_j (1-\phi_j)$ (i.e., $w_j = 1$); and (4) **Penalty-only scoring**: discarding the base-model probabilities and selecting/reranking candidates solely according to $r_t(v)$. We evaluate these variants on QWEN2.5-7B and LLAMA3-8B across HUMANEVAL+, MBPP+, API-BANK, and API-BENCH.

The ablation study(Figure 3) indicates that the *synergy* between *hard mask* and *soft gate* is a primary driver of EXECGATE's gains. On QWEN2.5-7B, removing the hard mask (only soft) drops HUMANEVAL+/MBPP+ to $46.8/60.3$ (vs. Full: $-2.8/-1.1$), while removing the soft gate (only hard) yields $47.2/60.1$ ($-2.4/-1.3$). A consistent degradation is observed on LLAMA3-8B, where the corresponding scores fall to $28.6/43.1$ ($-1.8/-1.4$) and $28.9/42.8$ ($-1.5/-1.7$), respectively. These results suggest that continuous penalties alone are insufficient to reliably prune *certainly infeasible* candidates, whereas hard filtering alone lacks the granularity to discriminate and rank *borderline* candidates; combining both achieves a better trade-off between constraint satisfaction and generation quality.

Moreover, constraint importance weighting further improves discriminability. Switching to uniform aggregation (Uniform-weights) leads to a small but consistent regression: QWEN2.5-7B reaches $48.8/60.8$ (vs. Full: $-0.8/-0.6$) and LLAMA3-8B reaches $29.4/43.6$ ($-1.0/-0.9$). This suggests that different constraint types (e.g., ordering dependencies, argument validity, entity consistency) are not equally critical for executability, and prioritizing more consequential violations provides stronger guidance.

5 Limitations.

While our approach consistently improves constraint satisfaction at generation time, several aspects deserve further exploration. First, our framework assumes access to an executability signal (e.g., lightweight verifiers or environment checks); extending it to more implicit, underspecified, or partially observable constraints is a promising direction. Second, the current instantiation performs incremental verification during decoding; although it is designed to be lightweight and plug-and-play, efficiency can be further improved via caching, verifier distillation, or early-exit gating. Finally, constraint injection naturally trades off strict feasibility and expressive freedom; developing more adaptive schedules or data-driven calibration of the gating strength is left for future work.

6 Conclusion

In this work, we identify and characterize the executability gap in grounded task generation, revealing a mechanistic preference drift where LLMs progressively prioritize linguistic fluency over physical feasibility in deeper layers. To bridge this gap, we propose EXECGATE, a training-free inference-time intervention that dynamically injects execution-aware constraints via a product-of-experts mechanism. By synergizing hard masks for deterministic rules and soft gates for dynamic states, our method consistently improves executability and correctness across code generation, tool use, and embodied planning benchmarks without requiring parameter updates. Our findings suggest that for grounded agents

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, and 26 others. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *Preprint*, arXiv:2204.01691.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021b. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022a. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *Preprint*, arXiv:2201.07207.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. 2022b. Inner monologue: Embodied reasoning through planning with language models. *Preprint*, arXiv:2207.05608.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *Preprint*, arXiv:2304.08244.

Zhongyang Li, Xiao Ding, Ting Liu, J. Edward Hu, and Benjamin Van Durme. 2020. Guided generation of cause and effect. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, page 3629–3636. International Joint Conferences on Artificial Intelligence Organization.

Ruibo Liu, Jason Wei, Shixiang Shane Gu, Te-Yen Wu, Soroush Vosoughi, Claire Cui, Denny Zhou, and Andrew M. Dai. 2022. Mind's eye: Grounded language model reasoning through simulation. *Preprint*, arXiv:2210.05359.

AI @ Meta Llama Team. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Neurologic decoding: (un)supervised neural text generation with predicate logic constraints. *Preprint*, arXiv:2010.12884.

Ian R. McKenzie, Alexander Lyzhov, Michael Pieler, Alicia Parrish, Aaron Mueller, Ameya Prabhu, Euan McLean, Aaron Kirtland, Alexis Ross, Alisa Liu, Andrew Gritsevskiy, Daniel Wurgaft, Derik Kauffman, Gabriel Recchia, Jiacheng Liu, Joe Cavanagh, Max Weiss, Sicong Huang, The Floating Droid, and 8 others. 2024. Inverse scaling: When bigger isn't better. *Preprint*, arXiv:2306.09479.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. *Preprint*, arXiv:2302.08468.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *Preprint*, arXiv:2305.15334.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *Preprint*, arXiv:2201.11227.

Xavier Puig, Tianmin Shu, Shuang Li, Zilin Wang, Joshua B. Tenenbaum, Sanja Fidler, and Antonio Torralba. 2020. Watch-and-help: A challenge for social perception and human-ai collaboration. *Preprint*, arXiv:2010.09890.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *Preprint*, arXiv:2109.05093.

Nexusflow.ai team. 2023. Nexusraven: Surpassing the state-of-the-art in open-source function calling llms.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *Preprint*, arXiv:2305.16291.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

A Datasets

HumanEval. HUMANEVAL is a widely adopted benchmark for evaluating code generation and program synthesis. It consists of hand-crafted Python programming problems paired with unit tests that automatically verify functional correctness. Each task requires the model to generate a complete function implementation that satisfies the given specification. Execution correctness is determined by whether the generated code passes all unit tests, making HUMANEVAL a standard testbed for assessing reasoning-driven code generation beyond surface-level syntax.

HumanEval+. HUMANEVAL+ extends HUMANEVAL by introducing additional execution-level constraints. Beyond passing unit tests, generated programs must also satisfy stricter syntactic and semantic requirements, such as valid control-flow structure and adherence to predefined API usage rules. This variant is designed to explicitly evaluate a model's ability to comply with execution-verifiable constraints rather than relying solely on functional correctness.

MBPP. MBPP (Mostly Basic Programming Problems) focuses on short, entry-level Python programming tasks that test fundamental programming concepts, including loops, conditionals, and basic data structures. Compared to HUMANEVAL, tasks in MBPP are generally shorter and more localized, providing a complementary evaluation of execution correctness under simpler problem formulations.

MBPP+. MBPP+ augments MBPP with explicit constraint annotations to assess compliance behavior. In addition to producing functionally correct code, the model must respect stricter execution constraints, such as function signature consistency, restricted operation sets, and syntactic validity throughout generation. Violations of these constraints result in execution failure, enabling a finer-grained evaluation of constraint adherence.

API-Bank. API-BANK is a tool-use benchmark that evaluates a model's ability to generate valid API invocation sequences. Each task provides a natural language goal along with a set of available APIs defined by explicit schemas. Successful execution requires correct API selection, valid parameter instantiation, and adherence to interface specifications. Errors such as calling unavailable APIs or supplying mismatched argument types are treated as execution failures.

API-Bench. API-BENCH emphasizes compositional and dependency-aware tool usage. Tasks often require generating multi-step API invocation sequences that satisfy ordering and dependency constraints, such as invoking prerequisite APIs before downstream calls. This benchmark is particularly sensitive to invalid invocation order and partial constraint violations, making it suitable for evaluating execution-aware planning under strict interface semantics.

Nexus. NEXUS provides complex tool-use scenarios that integrate multiple APIs with heterogeneous schemas. In addition to basic interface correctness, tasks involve cross-tool dependencies and state tracking across calls. The benchmark evaluates whether generated tool-use plans can be executed end-to-end without violating schema, type, or dependency constraints, highlighting the gap between linguistically plausible and executable action sequences.

VirtualHome. VIRTUALHOME is an embodied planning benchmark set in a simulated household environment. The model is required to generate sequences of symbolic actions that can be executed by an embodied agent. Each action is governed by explicit physical and temporal constraints, including object existence, spatial reachability, and state-dependent preconditions (e.g., an object must be acquired before being placed, and containers must be opened before interaction). A plan is considered valid only if it can be executed without violating any environment-defined constraints, making VIRTUALHOME a rigorous testbed for evaluating physical executability.

B Robustness to Verifier Precision

To demonstrate that the effectiveness of EXECGATE arises from the general paradigm of constraint injection rather than from the sophistication of any particular verifier, we investigate the model's sensitivity to *verifier precision*. Specifically, on code generation benchmarks (HUMANEVAL+ and MBPP+), we replace the default high-precision *Incremental AST Parser* with two simplified constraint strategies while keeping all other components unchanged. All variants are evaluated using the QWEN2.5-7B backbone.

Heuristic Regex (Medium Precision). This variant employs a lightweight verifier based on regular expressions, enforcing only coarse structural patterns such as indentation consistency and basic keyword constraints. It does not perform full syntactic parsing or maintain an explicit abstract syntax tree.

Vocabulary Masking (Low Precision). This variant adopts a minimal verifier that filters out tokens that are invalid in Python source code (e.g., arbitrary non-code symbols), but performs no structural or semantic checks beyond vocabulary-level validity.

We compare these simplified verifiers against both the baseline model and the full AST-based EXECGATE, allowing us to isolate the contribution of constraint injection itself from the accuracy of the underlying verifier.

Table 3: **Performance sensitivity to verifier implementation (Qwen2.5-7B).** We evaluate the impact of verifier precision on code generation benchmarks. Δ denotes the absolute improvement over the unconstrained baseline.

Verifier Implementation	Precision	HumanEval+	Δ	MBPP+	Δ
Baseline (No Constraints)	–	46.2	–	59.6	–
Vocabulary Masking	Low	48.4	+2.2	60.5	+0.9
Heuristic Regex	Medium	49.0	+2.8	61.0	+1.4
Incremental AST (Default)	**High**	**49.6**	**+3.4**	**61.4**	**+1.8**

The results clearly indicate that EXECGATE is insensitive to the specific implementation of constraints. Remarkably, even the coarse Heuristic Regex strategy achieves performance parity with the rigorous Incremental AST (49.0% vs 49.6% on HE+), and significantly outperforms the unconstrained baseline (46.2%). Even the simplest Vocabulary Masking captures the majority of the performance gain (+2.2%). This finding confirms that the primary driver of success is the EXECGATE framework itself—specifically, the mechanism of injecting external feedback to correct preference drift—rather than the complexity or perfection of the constraint verifier. As long as the system provides a directional feasibility signal, the model can effectively align its generation.

C Sensitivity to Candidate Set Size (K)

To determine the optimal search space required to recover physically feasible tokens, we analyzed the performance sensitivity to the candidate set size K (as defined in Algorithm 1, Line 3). Using the Qwen2.5-7B backbone on HumanEval+ (HE+) and VirtualHome benchmarks, we varied K within the set $\{1, 5, 10, 20, 50\}$ while keeping the constraint strength fixed at $\lambda = 1.0$. This experiment tests the hypothesis that due to "layer-wise preference drift", valid grounded tokens are often suppressed to lower probability ranks by the base model. Therefore, $K = 1$ (equivalent to standard greedy decoding) serves as the baseline, and we measure how expanding the search scope enables the retrieval of these suppressed but executable candidates.

Table 4: **Effect of candidate size K on executability and latency.** Results are reported on HUMANEVAL+ (Pass@1) and VIRTUALHOME (Executability), with inference latency measured relative to greedy decoding (K=1).

Candidate Size (K)	HumanEval+	VirtualHome	Latency (Rel.)
$K = 1$ (Greedy)	46.2	61.3	1.00×
$K = 5$	46.5	62.0	1.12×
K = 10 (Default)	**49.6**	**63.7**	1.25×
$K = 20$	49.8	64.1	1.48×
$K = 50$	49.9	64.3	2.15×

The results exhibit a clear upward trend, indicating that a moderate expansion of the search space is sufficient to bridge the executability gap. When $K = 1$, the verifier has no alternative tokens to select from, and performance remains at the baseline level (HE+ = 46.2%). As K increases to 10, we observe a substantial performance improvement (HE+ rises to 49.6%), suggesting that physically feasible tokens are typically present among the top-10 predictions but are often overshadowed by more linguistically fluent candidates due to internal model biases. Beyond this point, further increasing K yields diminishing returns: for example, at $K = 50$, HE+ improves only marginally to 49.9%, while inference latency increases significantly. These results indicate that $K = 10$ achieves a favorable Pareto trade-off, delivering strong executability gains while maintaining efficient inference.

D Comparison with Iterative Self-Correction Strategies

To demonstrate the efficiency and efficacy of EXECGATE compared to prompt-based repair methods, we benchmarked it against an Iterative Self-Correction baseline on the HumanEval+ (Code Generation) dataset using Qwen2.5-7B. The Self-Correction baseline follows a "Generate-Verify-Refine" loop: if the generated code fails the AST/Schema check, the error message is fed back

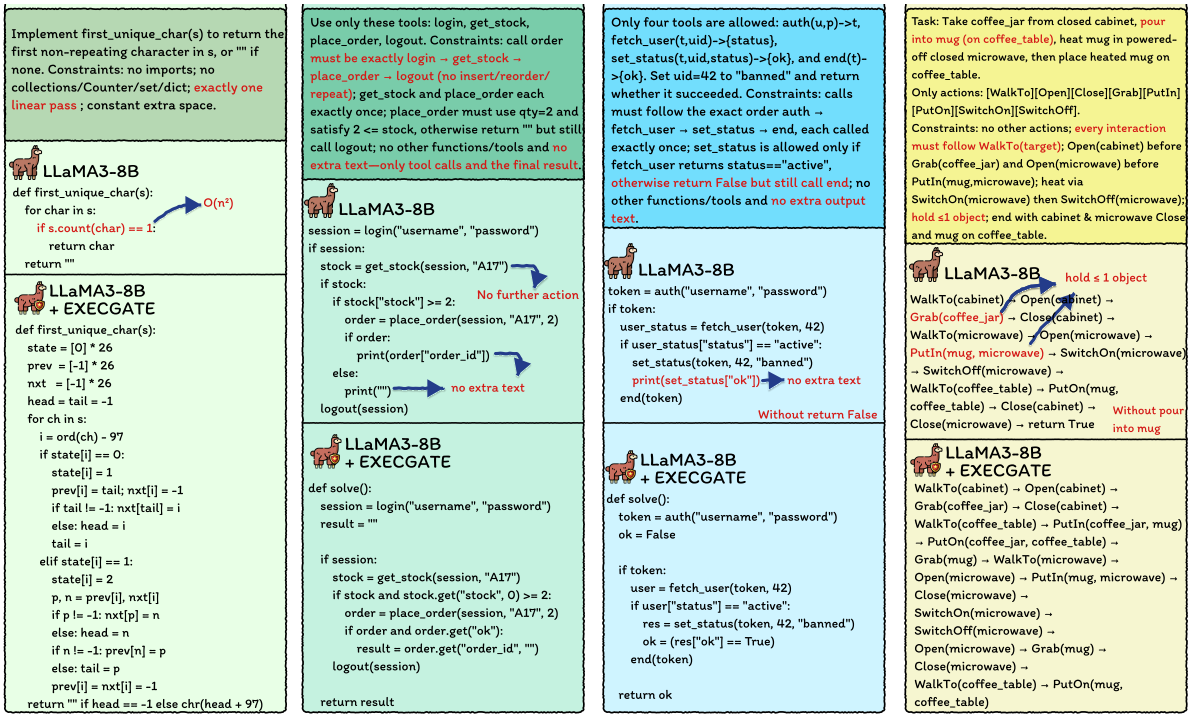

Figure 4: **Qualitative comparisons of constraint compliance across coding, tool-use, and embodied tasks.** The base LLaMA3-8B frequently violates explicit constraints (e.g., hidden $O(n^2)$ scans, illegal API call order or extra outputs, and invalid VirtualHome action preconditions/holding limits), whereas EXECGATE enforces the specified constraints and produces executable, verifier-consistent solutions.

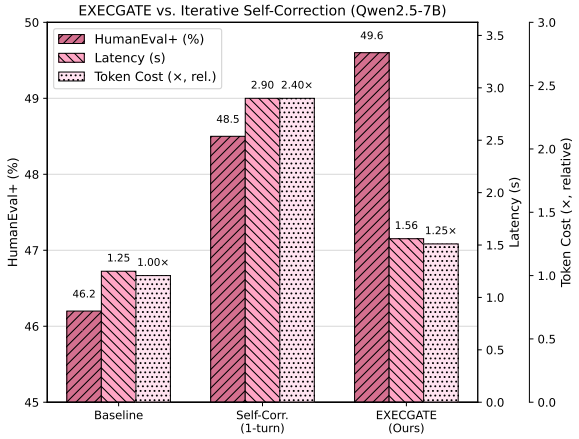

Figure 5: **EXECGATE vs. iterative self-correction (Qwen2.5-7B).** Comparison in terms of executability, inference latency, and token cost. Token cost is reported relative to standard greedy generation.

to the model via a prompt to regenerate the solution (allowing up to 1 refinement turn). The experiment highlights a fundamental advantage of EXECGATE: efficiency. While Self-Correction significantly improves performance over the baseline (46.2% → 48.5%) by reacting to errors, it comes at a steep price—nearly doubling the latency and 2.4x token cost due to the need for regeneration and processing long error contexts. In contrast, EXECGATE achieves slightly superior performance (49.6%) by preventing errors before they occur. By shaping the probability distribution during the initial decoding pass, EXECGATE avoids the wasteful "generate-fail-retry" cycle, offering a much more scalable solution for real-time grounded tasks where latency and compute budgets are constrained.

E Qualitative Analysis

Figure 4 presents representative failure cases of the base model and the corresponding fixes enabled by EXECGATE under explicit, execution-checkable constraints. Across *code generation*, *tool-use*, and *embodied planning* settings, we observe a consistent pattern: the base model tends to satisfy the *surface task intent* while silently violating *hard constraints* that are only detectable by a verifier/executor, whereas EXECGATE reliably steers decoding toward verifier-consistent action/code sequences.

Code generation: hidden constraint violations. In the "first unique character" example, the base model outputs a seemingly correct solution but relies on `s.count(char)` inside a loop, which induces a hidden $O(n^2)$ scan and violates the *exactly-*

one linear pass constraint. In contrast, EXECGATE produces a verifier-compliant implementation that maintains a constant-size state and avoids disallowed constructs, illustrating that constraint injection can correct *complexity- and structure-level* violations that are non-obvious from functional outputs alone.

Tool-use: protocol errors and output contamination. For the ordering task, the base model exhibits typical protocol drift: it either introduces extra outputs (e.g., `print`) or fails to return the required value, and may skip required steps under conditional branches, breaking the mandated call order. With EXECGATE, the generated program follows the exact tool invocation schema (order, cardinality, and argument constraints) and cleanly returns the final result without extraneous side effects, demonstrating improved adherence to *interaction protocols*.

Embodied planning: precondition failures and state inconsistency. In the VirtualHome-style coffee-making task, the base model often omits prerequisite actions (e.g., manipulating the mug without first reaching/grabbing it, or violating the one-object holding limit) and may leave containers in invalid terminal states. EXECGATE corrects these errors by enforcing action preconditions and terminal-state constraints (e.g., `Open` before `Grab`/`PutIn`, `SwitchOn` then `SwitchOff`, and closing containers at the end), yielding executable sequences consistent with environment dynamics.

Takeaway. These cases highlight that the gains of EXECGATE stem from *explicit constraint injection during decoding* rather than the sophistication of any particular verifier: by continually aligning generation with execution-grounded constraints, EXECGATE mitigates protocol drift and improves reliability in settings where "plausible" generations are insufficient.