

Distribution-Aware Constrained Code Generation: Fast FSM Decoding with AST-Guided Recovery

Achraf Hafsaoui
Technical University of Darmstadt
Darmstadt, Germany

Abstract

Constrained Decoding (CD) improves code generation by masking invalid tokens, but current methods face a trade-off between latency, structural admissibility, and semantic intent preservation. Fast FSM-based decoders provide efficient syntactic control but lack semantic awareness, while AST-based methods capture richer program structure at higher cost. This paper proposes a tiered Fast-Path/Slow-Path architecture: an $O(1)$ FSM enforces grammar-level admissibility, while an asynchronous AST supports selective recovery. A distribution-aware tripwire monitors the valid probability mass ρ_t left by the FSM mask and triggers rollback when the admissible region becomes misaligned with the model's unconstrained distribution. The system then resumes from a stable AST checkpoint with a checkpoint-local branch penalty, preserving fast syntactic control while reducing constraint-induced semantic drift.

Keywords

Constrained Decoding, Large Language Models, Code Generation, Semantic Drift, Distributional Alignment, AST-Guided Recovery

ACM Reference Format:

Achraf Hafsaoui. 2026. Distribution-Aware Constrained Code Generation: Fast FSM Decoding with AST-Guided Recovery. In . ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Large Language Models (LLMs) generate code probabilistically, while programming languages require deterministic syntactic structure. Constrained Decoding (CD) addresses this mismatch by masking invalid tokens during generation. However, fast grammar-based decoders can preserve syntax while still distorting the model's intended continuation: when the FSM-admissible region contains little unconstrained probability mass, generation is forced into a low-probability path.

This paper formalizes this tension as the **Constrained Decoding Trilemma**: practical systems struggle to jointly optimize inference latency, structural admissibility, and semantic intent preservation. We propose a **Fast-Path/Slow-Path Architecture** that keeps syntactic masking on an $O(1)$ FSM Fast Path, while using an asynchronous AST Slow Path only when distributional drift is detected. A valid-mass tripwire monitors ρ_t and triggers rollback to a stable AST checkpoint when $\rho_t \leq \tau_t$.

Our contributions are: (i) a trilemma formulation of constrained code generation, (ii) a valid-mass tripwire that detects constraint-induced distributional drift, and (iii) an AST-guided rollback mechanism that restores generation from stable program checkpoints without relaxing syntactic guarantees.

2 Background

2.1 The Reliability Gap in Auto-Regressive Code Generation

The foundational architecture of contemporary Large Language Models (LLMs) is inherently auto-regressive. During code synthesis, these models generate sequences incrementally, where the probability of predicting the next token, x_t , is conditioned entirely on the sequence of preceding tokens, $x_{<t}$. Mathematically, the decoding process seeks to model the joint probability distribution of a token sequence $X = (x_1, x_2, \dots, x_N)$ as the product of conditional probabilities:

$$P(X) = \prod_{t=1}^N P(x_t \mid x_{<t})$$

While this stochastic, probability-driven approach is exceptionally effective for natural language processing, as human communication is naturally tolerant of ambiguity, structural variations, and minor grammatical deviations, it introduces a reliability gap when applied to formal programming languages [6]. Unlike natural language, source code operates under strict, deterministic rules governed by parsers, type checkers, and compilers. A single misplaced bracket, an incorrectly typed variable, or an indentation error may result in a compilation failure or runtime error. Because standard auto-regressive models do not possess an intrinsic, formal understanding of abstract syntax trees (ASTs), execution contexts, or static typing environments, their decoding process is effectively "blind" to the overarching logic and state of the program they are generating. They optimize for the statistically most probable next token based on their training corpus, rather than the logically and syntactically sound next operation.

Consequently, base models are highly susceptible to logic "hallucinations" in a programming context. Common manifestations of this reliability gap include the generation of syntactically invalid constructs, the referencing of variables that are strictly out of scope, or the invocation of deprecated, hallucinated, or non-existent Application Programming Interfaces (APIs). Because the likelihood of a compound failure scales with the length of the output, the risk of generation errors generally increases with sequence length. This inherent friction between stochastic generation and deterministic execution necessitates the integration of formal external interventions, establishing the core motivation for constrained decoding.

2.2 The Mechanics of Constrained Decoding (CD)

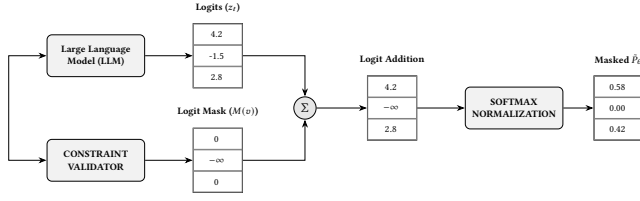


Figure 1: Constrained Decoding pipeline

To mitigate the reliability gap, Constrained Decoding (CD) intervenes directly in the model’s auto-regressive generation phase. Rather than altering the underlying neural architecture or fine-tuning the model weights, CD acts as a real-time filter that dynamically restricts the output space at each generation step.

In a standard unconstrained generation step, an LLM produces a vector of raw scores, or logits z_t , for every token v in its vocabulary V . These logits are converted into a probability distribution via the softmax function:

$$P(x_t = v \mid x_{<t}) = \frac{\exp(z_{t,v})}{\sum_{v' \in V} \exp(z_{t,v'})}$$

Constrained decoding, depicted in Figure 1, fundamentally alters this distribution by introducing a stateful constraint function or validator. Let V_{valid} represent the subset of tokens in V that satisfy the target language’s grammatical or semantic rules given the prefix $x_{<t}$. The CD mechanism constructs a binary mask $M(v)$ defined in log-space:

$$M(v) = \begin{cases} 0, & \text{if } v \in V_{valid} \\ -\infty, & \text{if } v \notin V_{valid} \end{cases}$$

This mask is added directly to the output logits prior to the softmax computation:

$$\tilde{P}(x_t = v \mid x_{<t}) = \text{softmax}(z_{t,v} + M(v))$$

By forcing the probability of all invalid tokens to exactly zero, the model is mathematically compelled to sample only from structurally sound continuations.

The efficacy of this masking heavily depends on the underlying search strategy. In **Greedy Decoding**, the system simply selects the highest-probability valid token at each step ($x_t = \text{argmax} \tilde{P}$). While computationally inexpensive, greedy search is strictly monotonic; it cannot recover if a sequence of valid tokens eventually leads to a state where V_{valid} becomes empty or semantically irrelevant.

To expand the search horizon, frameworks like PICARD [7] implement constrained **Beam Search**. Instead of a single path, beam search maintains the top k most probable sequence hypotheses concurrently. At each time step, the constraint mask is applied across all active beams, pruning branches that violate formal grammar. While beam search offers a broader exploration of the valid state space than greedy decoding, it remains fundamentally unidirectional. This limitation sets the stage for advanced non-linear recovery mechanisms, such as dynamic backtracking, which attempt to salvage generation when all active beams drift from the model’s semantic intent.

2.3 The Evolution of Constraints: From Syntax to Semantics

The trajectory of constrained decoding in automated code generation is defined by a necessary transition from stateless structural filtering to stateful semantic integration. Initial frameworks focused predominantly on lexical and syntactic validation to mitigate the unreliability of base LLMs. By restricting the auto-regressive generation process to valid paths defined by regular expressions or strict context-free grammars (CFGs), early systems guaranteed that the generated sequence would parse correctly according to the target language’s baseline syntax rules.

However, baseline syntactic correctness does not preclude catastrophic logical errors; a syntactically flawless program can still invoke undefined variables, pass incorrect arguments to functions, or violate strict type constraints. Recognizing the limitations of purely syntactic enforcement, recent literature highlights a necessary paradigm shift towards semantic-aware generation. As articulated in contemporary research on projectional decoding frameworks [1], modern constraint mechanisms are actively extending beyond context-free grammars to address deep semantic validity.

This new phase involves transitioning from stateless token masking to stateful generation environments. Instead of merely querying whether a token is grammatically permissible, semantic-aware decoders evaluate whether the token maintains the logical integrity of the broader codebase [1]. This allows for the enforcement of context-sensitive rules, specifically targeting particular classes of semantic constraints such as guaranteed type-safe code generation.

While this semantic shift significantly reduces complex compilation and runtime errors, it simultaneously introduces a new frontier of friction: the rigidity of semantic rules. As constraints become increasingly strict and globally aware, they act as absolute programmatic overrides. When an LLM’s probabilistic reasoning conflicts with a rigid semantic constraint, the constraint inadvertently overrides the model’s underlying reasoning. This tension between enforcing complex semantic validity and preserving the model’s functional goal directly precipitates the modern challenge of intent distortion.

3 Analysis: The Evolution of Constrained Decoding

The landscape of Constrained Decoding (CD) has undergone a rapid evolution. Initially conceived as a mechanism for simple syntactic filtering, the field quickly fractured into competing methodologies. This section traces that evolution by evaluating the foundational greedy methods, the shift toward intent-preserving backtracking, and the rise of strict structural alternatives. Ultimately, a critical analysis of the literature reveals a persistent **Trilemma in Constrained Decoding**: frameworks struggle to simultaneously balance **inference latency**, **structural guarantees**, and **intent preservation**.

3.1 Foundations of Constrained Decoding

The genesis of formal constraints in Large Language Model (LLM) inference is primarily rooted in the development of incremental parsing and semantic filtering, exemplified by the seminal works PICARD [7] and Synchronesh [6].

PICARD operates on the premise that the reliability of text-to-SQL systems can be significantly enhanced by integrating an incremental parser directly into the auto-regressive decoding process. Methodologically, it employs a constrained beam search where, at each time step, the parser evaluates the partial sequence to prune tokens that would violate the target language’s context-free grammar. While PICARD demonstrates high accuracy on benchmarks like Spider, its conclusion that un-tuned models can outperform fine-tuned counterparts via structural guidance is marred by a critical omission: the framework lacks a mechanism to handle context-sensitive constraints effectively. Because it relies on a greedy integration of the parser, PICARD can successfully enforce syntax while remaining blind to deeper semantic errors. Its depth is further limited by its specificity to SQL; the methodology does not inherently scale to general-purpose programming languages where the state space of valid continuations is significantly more volatile.

In contrast, SynchroMesh expands this paradigm by introducing Constrained Semantic Decoding (CSD), predicated on the premise that models should be restricted to a set of programs that are valid both syntactically and semantically (e.g., ensuring referenced variables are within scope). Its methodology utilizes a completion engine and a vocabulary-masking function to bridge the gap between the LLM’s token distribution and the target language’s grammar. Although SynchroMesh achieves impressive results in domain-specific languages, its methodology assumes a high-fidelity oracle (referred to as Completion Engine) is always available. A significant inconsistency in its evaluation is the lack of consideration for scenarios where the constraint mask, in its attempt to ensure validity, forces the model into a low-probability region of its latent space that no longer aligns with the user’s prompt.

Ultimately, while both PICARD and SynchroMesh established the relevance of token-level masking, they share a common limitation: they treat decoding as a unidirectional, greedy traversal. By assuming that a valid prefix will always lead to a valid and semantically sound conclusion, these foundations fail to account for dead ends that can arise during constrained generation. This oversight results in a rigid enforcement of rules that often comes at the expense of the model’s inherent reasoning capabilities.

3.2 AdapTrack: Mitigating Intent Distortion

AdapTrack [3] identifies a critical flaw in prior work termed “intent distortion.” The authors argue that forcing an LLM to satisfy a constraint often overrides its semantic intent. Specifically, when an LLM’s preferred output path is blocked by a constraint, token-level masking forces the model to select a highly unnatural token simply to satisfy the rule, resulting in a deviation that irreversibly corrupts the remaining generation sequence.

Their methodology introduces adaptive backtracking through rejection sampling. AdapTrack maintains, for previously explored prefixes, estimates of the probability that a valid complete sequence can still be reached. As new constraint information is discovered, these estimates and the corresponding next-token distributions are updated. Rejection sampling then determines whether an earlier token choice should be retained; if it is rejected, the decoder selects an alternative token at that prefix and discards the resulting suffix. The method is designed to preserve the language model’s relative

probabilities over valid complete sequences. The authors evaluate it on the specialized API-migration datasets *TfV1* and *TfV1Real*, where it improves functional correctness.

A critical evaluation of AdapTrack reveals that its primary strength is its distribution-preserving formulation of constrained generation. However, recovery remains organized around token prefixes rather than program structure: rollback points are not selected according to declarations, scopes, or other AST boundaries. Although short backtracking distances capture much of the benefit on the evaluated API-migration tasks, it remains unclear whether token-oriented recovery scales equally well to longer and more deeply nested generation contexts. Moreover, repeated rejection sampling and branch revision can increase decoding cost when valid completions are difficult to reach. The proposed architecture therefore trades AdapTrack’s exact distributional guarantee for low-overhead common-case decoding and structurally guided recovery.

3.3 Structural Validation via ASTs and Graph Models

In contrast to the token-level backtracking of **AdapTrack** [3], a distinct branch of contemporary research has attempted to enforce deep programming logic through dynamic structural constraints.

ChopChop [5] operates on the premise that token-level regular expressions or context-free grammars are fundamentally insufficient for enforcing deep programming logic. Its methodology introduces a programmable framework that constructs Abstract Syntax Trees (ASTs) dynamically during the decoding process, allowing users to define high-level semantic predicates. By analyzing the structural properties of the partial AST rather than raw token sequences, ChopChop can enforce complex non-local properties such as variable scoping and initialization invariants. However, the depth of this approach is offset by a critical architectural omission: the paper does not adequately address the latency scaling factor. Building and validating partial ASTs at every token emission interval introduces substantial computational overhead. Furthermore, standard AST generation assumes a complete, well-formed input string. When applied to incremental auto-regressive output, the parser must continually synthesize “dummy” closures to validate incomplete abstract syntax trees, further increasing the computational overhead. Consequently, because the methodology lacks a graceful degradation strategy when the underlying language grammar is highly ambiguous or evolving, it can introduce bottlenecks in real-time inference.

To mitigate the latency of synchronous AST construction, recent dynamic parsing frameworks such as **Projectional Decoding** [1] have introduced asynchronous semantic validation. Rather than repeatedly halting generation to build full ASTs, Projectional Decoding maintains a “partial graph model” in the background during text generation. This enables the system to continuously validate deeper semantic rules alongside standard syntactic checks without severely interrupting the primary auto-regressive token stream.

While Projectional Decoding successfully optimizes the management of dynamic ASTs, relying on graph-based validation inherently maintains a high computational ceiling. Furthermore, enforcing strict semantic graphs can still rigidly block the model’s desired reasoning path if the graph state becomes misaligned with

the model’s natural language context, ultimately failing to fully resolve intent distortion. The computational limits of dynamic graph validation thus forced a parallel paradigm shift in the field toward pre-compiled, constant-time validation models.

3.4 High-Efficiency FSM Engines and the Latency Breakthrough

As evidenced by the computational overhead of dynamic AST construction, inference latency remains a critical bottleneck. Evaluating deep structural constraints at every generation step t often results in a time complexity of $O(|x_{<t}|)$ per token. To resolve this latency axis, the field has recently undergone a paradigm shift toward pre-compiling constraints into highly optimized Finite-State Machines (FSMs).

Shifting the focus to static typing, Mündler et al. [4] tackled the premise that syntactic correctness is meaningless without type safety. Their methodology pioneered the use of a prefix automaton—functioning essentially as a high-efficiency Finite-State Machine (FSM)—compiled from target type definitions (such as TypeScript interfaces). By computing the precise set of valid next tokens based on the current type environment, their system proactively prevents the model from generating type-violating expressions. However, the prefix automata assume a static, fully defined type environment, which struggles to adapt to the dynamic APIs and loosely defined variables common in real-world software engineering.

Building upon the foundational mechanics of Mündler’s prefix automata, recent inference engines have generalized automaton-based constraints to achieve low validation overhead. Frameworks such as **SGLang** [8] and the **XGrammar** family (including the early-2026 release of **XGrammar-2** [2]) utilize “compressed FSMs” or pushdown automata as their core completion engines. By mathematically reducing context-free grammar validation to a closed-form state transition, these engines maintain the active state s_t of the generation process. Given the model vocabulary V , the engine maps s_t to an allowable token subset with approximately constant-time state lookup in the common case. This enables low-overhead logit masking, substantially reducing the structural overhead of constrained decoding. SGLang, for instance, demonstrates that FSM-guided generation can even surpass the speed of unconstrained baselines via jump-forward decoding along deterministic transition paths.

Crucially, this efficiency relies on the reduction of the target grammar to a Deterministic Context-Free Grammar (DCFG). This reduction inherently strips away non-local semantic context, meaning FSMs cannot enforce higher-order constraints such as generic type instantiation or dynamic API resolution. The resulting architecture is fast precisely because it is semantically impoverished. And this semantic impoverishment can contribute to intent distortion during the auto-regressive decoding cycle. At any generation step t , the LLM’s unconstrained probability distribution provides a local proxy for its preferred continuation. Because the FSM operates strictly on pre-compiled local transitions, it is entirely blind to this global reasoning.

When the LLM strongly prefers a structurally inadmissible token, the FSM applies an absolute $-\infty$ mask to the intended token.

The masked softmax function is then renormalized over the remaining structurally valid subset, forcing the model to sample from the remaining structurally valid subset—even if those tokens fall into the lowest percentiles of the model’s original distribution. As explicitly demonstrated by *AdapTrack* [3], because the token generated at step t becomes the immutable context for step $t + 1$, this forced, unnatural token selection can shift the subsequent autoregressive context toward a semantically unstable trajectory, causing the subsequent generation to hallucinate or completely derail.

3.5 The Constrained Decoding Trilemma

The evolution of the literature exposes a persistent design tension in real-time constrained generation: systems are typically forced to trade off inference latency, syntactic admissibility, and semantic intent preservation. We refer to this tension as the *Constrained Decoding Trilemma*. The central difficulty is that the mechanisms that provide fast structural control are usually too local to represent semantic intent, while mechanisms that capture richer program context are too expensive to execute synchronously at every decoding step. To formalize this tension, we parameterize the performance of a constrained decoding system C along three axes: Latency ($\mathcal{L}$), Structural Admissibility ($\mathcal{S}$), and Intent Preservation ($\mathcal{I}$).

- (1) **Latency ($\mathcal{L}$):** The computational overhead introduced per generated token. For real-time auto-regressive inference, a practical constrained decoder should keep the token-critical path close to constant time, i.e., $\mathcal{L} \approx O(1)$ in the common case.
- (2) **Structural Admissibility ($\mathcal{S}$):** The ability to prevent the decoder from emitting tokens that violate the target grammar state. We define $\mathcal{S}(X) \in \{0, 1\}$ with respect to the grammar encoded by the constraint system, where $\mathcal{S}(X) = 1$ indicates that the generated sequence X remains admissible under the compiled grammar.
- (3) **Intent Preservation ($\mathcal{I}$):** The degree to which the constrained decoder preserves the model’s unconstrained continuation trajectory. Since developer intent is not directly observable during decoding, we approximate local model intent through the unconstrained next-token distribution $P_\theta(\cdot \mid x_{<t})$. A constraint strongly distorts this trajectory when the admissible region contains little of the model’s unconstrained probability mass.

Under this formulation, the key quantity connecting structural constraints and intent preservation is the valid probability mass

$$\rho_t = \sum_{v \in V_{\text{valid}}(s_t)} P_\theta(v \mid x_{<t}),$$

where $V_{\text{valid}}(s_t)$ is the set of tokens admitted by the constraint system in state s_t . High ρ_t indicates that the structural constraint remains compatible with the model’s preferred continuation space. Low ρ_t indicates that the decoder is being forced to renormalize over a low-probability subset of the vocabulary, creating a risk of semantic drift. Importantly, low ρ_t is not treated as proof of semantic error, but as a direct operational signal that the constraint is forcing the decoder away from the model’s unconstrained continuation.

The trilemma can therefore be understood as a recurring pattern of pairwise optimization:

Scenario 1: Optimizing S and I (Dynamic Semantic Graphing). Frameworks such as *ChopChop* and related semantic-aware methods use richer program representations, such as partial ASTs or graph-based state, to preserve semantic context while enforcing structural constraints. This improves semantic alignment and structural admissibility, but it requires evaluating deeper program context during generation. As a result, the per-token cost can scale with the length or structural complexity of the prefix, weakening the latency axis: $\mathcal{L} \rightarrow O(|x_{<t}|)$ or worse.

Scenario 2: Optimizing $\mathcal{L}$ and I (Unconstrained Generation). Standard probabilistic decoding keeps latency low and preserves the model’s unconstrained distribution because no external mask is applied. In this case, the constrained and unconstrained distributions coincide, so no constraint-induced distributional drift is introduced. However, without grammar-level enforcement, the model may emit syntactically invalid or structurally inadmissible code, especially as sequence length increases.

Scenario 3: Optimizing $\mathcal{L}$ and S (High-Efficiency FSMs). High-efficiency automaton-based systems such as SGLang and XGrammar enforce grammar-level admissibility through log-space masking in approximately constant time [2, 8]. This preserves the latency and structural axes, but the FSM remains semantically impoverished: it cannot evaluate whether the remaining valid tokens still align with the model’s intended continuation. When ρ_t becomes small, the decoder is forced to sample from a structurally valid but distributionally low-probability region, increasing the risk of semantic drift.

This analysis suggests that the trilemma is not merely a matter of choosing stronger constraints, but of deciding where different forms of reasoning should occur. A monolithic decoder that performs all syntactic and semantic checks synchronously risks high latency, while a purely local FSM decoder risks semantic drift. This motivates a tiered architecture: a fast syntactic path for common-case admissibility, paired with a selective recovery path that is activated only when the FSM constraint becomes distributionally misaligned with the model’s unconstrained continuation.

4 Proposed Approach

As established in the literature review, existing constrained decoding methods expose a persistent trade-off between latency, syntactic admissibility, and semantic intent preservation. Rather than optimizing one axis in isolation, we propose a tiered architecture that separates low-latency structural validation from selective intent recovery.

To achieve this, we propose a hybrid framework that integrates the $O(1)$ execution speed of Finite-State Machine (FSM) completion engines [8] with AST-guided rollback and distribution-aware drift detection. Pure FSM decoding asks only whether a token is admissible; the proposed architecture additionally asks whether the admissible region still contains enough of the model’s probability mass to support a plausible continuation.

4.1 Architectural Overview: The Fast-Path/Slow-Path Paradigm

To address the practical tension described by the Constrained Decoding Trilemma, we propose a tiered validation architecture that

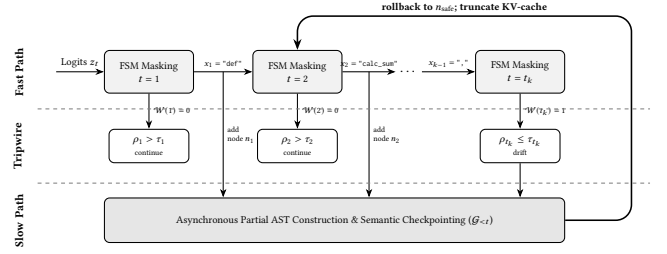


Figure 2: Fast-Path/Slow-Path architecture

separates constraint enforcement from intent recovery. The goal is not to eliminate the trilemma in the strict worst case, but to reduce its practical impact by moving expensive semantic reasoning outside the token-critical path. In standard generation, the system relies on a synchronous finite-state constraint engine for constant-time masking. Only when the constraint mask appears to strongly distort the model’s unconstrained distribution does the architecture synchronize with a slower structural recovery mechanism.

This design is organized around a *Fast-Path/Slow-Path Paradigm*. The Fast Path is responsible for enforcing local syntactic admissibility at every decoding step, while the Slow Path maintains a delayed but richer representation of the generated program structure. The two components are depicted in Figure 2: a synchronous Finite-State Machine (FSM) acts as the primary completion engine in the upper layer, while an asynchronous Abstract Syntax Tree (AST) monitor runs as a background parser in the lower layer. Between them, a distortion monitor acts as a control interface, deciding when the Fast Path should continue normally and when recovery should be triggered.

4.1.1 The Fast Path: $O(1)$ Synchronous Execution. During standard auto-regressive generation, the system exclusively utilizes the Fast Path. Let V represent the LLM’s vocabulary and $x_{<t}$ denote the sequence of generated tokens up to step t . The LLM outputs an unconstrained logit vector $z_t \in \mathbb{R}^{|V|}$, which yields the unconstrained probability distribution:

$$P_\theta(x_t = v \mid x_{<t}) = \text{softmax}(z_t)$$

In parallel, the FSM completion engine maintains an active formal state s_t . Given this state, the FSM computes the set of structurally admissible next tokens,

$$V_{\text{valid}}(s_t) \subseteq V,$$

where a token v is admissible if consuming it leads to a defined transition in the compiled automaton. The transition function δ updates the formal state after a token has been emitted:

$$s_{t+1} = \delta(s_t, x_t).$$

The Fast Path enforces syntactic admissibility through a binary log-space mask:

$$M_{\text{FSM}}(v) = \begin{cases} 0, & \text{if } v \in V_{\text{valid}}(s_t), \\ -\infty, & \text{otherwise.} \end{cases}$$

The final token x_t is sampled from the masked distribution

$$\tilde{P}_\theta(x_t = v \mid x_{<t}) = \text{softmax}(z_t + M_{\text{FSM}})_v.$$

This ensures that tokens outside the FSM-admissible set receive zero probability mass. As shown in Figure 2, normal decoding therefore consists of repeated FSM masking steps, for example when emitting structurally expected tokens such as `def`, an identifier, or a delimiter. Since the valid-token set is obtained from the current automaton state rather than by reparsing the full prefix, the Fast Path approximates constant-time structural validation in the common case.

4.1.2 The Slow Path: Asynchronous Semantic Monitoring. To mitigate the intent distortion that can arise from semantically impoverished DCFG-based masking, the architecture concurrently maintains a Slow Path. As the FSM drives token-level sampling, a lightweight, asynchronous parser constructs a partial Abstract Syntax Tree, denoted as $\mathcal{G}_{<t}$, reflecting the macro-structural state of the code (e.g., active variable scopes, parent classes, and unresolved closures).

Because $\mathcal{G}_{<t}$ is maintained asynchronously, it remains outside the token-critical path during ordinary decoding. The Slow Path therefore functions as a background structural memory that can be queried when the Fast Path encounters a high-conflict state. In Figure 2, this corresponds to the lower AST-monitoring layer receiving committed tokens from the upper FSM layer, while remaining outside the main decoding loop unless a distortion signal is raised.

This separation gives the architecture its central theoretical advantage. In the common case, decoding inherits the low latency of FSM-based masking. In exceptional cases, where the mask substantially conflicts with the model’s unconstrained distribution, the Slow Path provides the structural information needed for controlled rollback and recovery.

4.2 Detecting the Conflict: FSM Tripwires and Distributional Drift

The critical control mechanism of the Fast-Path/Slow-Path architecture is the detector that decides when ordinary FSM-based masking is sufficient and when AST-guided recovery should be invoked. The FSM engine evaluates admissibility only with respect to the compiled grammar state. It can determine whether a token is structurally valid, but it does not observe how strongly the language model preferred the regions of the vocabulary removed by the mask. To bridge this gap, we introduce a distribution-aware monitor, termed the *FSM Tripwire*, which estimates the degree of conflict between the model’s unconstrained next-token distribution and the FSM-admissible continuation set.

At generation step t , the LLM produces an unconstrained logit vector z_t over the vocabulary V , inducing the probability distribution

$$P_\theta(x_t = v \mid x_{<t}) = \text{softmax}(z_t)_v.$$

Given its current formal state s_t , the FSM defines the admissible continuation set

$$V_{\text{valid}}(s_t) \subseteq V.$$

The central quantity used by the tripwire is the amount of unconstrained probability mass that remains available after applying the FSM constraint. We define this *valid probability mass* as

$$\rho_t = \sum_{v \in V_{\text{valid}}(s_t)} P_\theta(x_t = v \mid x_{<t}).$$

The value ρ_t measures the compatibility between the model’s unconstrained distribution and the current structural constraint. If ρ_t is high, the FSM still permits a large portion of the model’s preferred continuation space, so masking is unlikely to cause severe semantic deviation. If ρ_t is low, the FSM forces the decoder to sample from a region that the model assigned little probability mass to, indicating a likely conflict between structural admissibility and the model’s intended trajectory.

We refer to this conflict as *distributional drift*. Distributional drift occurs when the constrained continuation space departs substantially from the model’s unconstrained decoding trajectory. This serves as an operational approximation of semantic drift: since the developer’s true intent is not directly observable during decoding, the system estimates local model intent through the geometry of the unconstrained probability distribution.

To make the detector adaptive, we condition its sensitivity on the model’s uncertainty. We compute the Shannon entropy of the unconstrained distribution:

$$H_t = - \sum_{v \in V} P_\theta(x_t = v \mid x_{<t}) \log P_\theta(x_t = v \mid x_{<t}).$$

The normalized entropy is defined as

$$\hat{H}_t = \frac{H_t}{\log |V|},$$

where $\hat{H}_t \in [0, 1]$. Low entropy indicates that the model’s probability mass is concentrated in a small set of continuations, while high entropy indicates a flatter and more uncertain distribution.

The tripwire threshold is defined as an entropy-dependent lower bound on the valid probability mass:

$$\tau_t = \beta + \alpha(1 - \hat{H}_t),$$

where α and β are tunable hyperparameters controlling the sensitivity of the detector. This formulation makes the tripwire stricter when the model is confident. If entropy is low, a small value of ρ_t is more concerning because the FSM is excluding continuations that the model strongly preferred. If entropy is high, the detector becomes more permissive because the model itself has not committed strongly to a narrow continuation region.

The FSM Tripwire is then defined as

$$\mathcal{W}(t) = \mathbb{I}[\rho_t \leq \tau_t].$$

When $\mathcal{W}(t) = 1$, the system interprets the current decoding state as a high-conflict point between the model distribution and the FSM constraint. Rather than forcing the model to immediately sample from the remaining low-probability valid tokens, the synchronous Fast Path is paused. The partial output prefix is then handed to the AST-guided Slow Path, which identifies a stable rollback point before generation resumes. In this way, the tripwire acts as the control interface between constant-time syntactic enforcement and selective semantic recovery.

For diagnostic purposes, the highest-probability unconstrained token can also be recorded:

$$v^* = \arg \max_{v \in V} P_\theta(x_t = v \mid x_{<t}).$$

If $v^* \notin V_{\text{valid}}(s_t)$, the FSM has removed the model’s most preferred local continuation. However, the primary tripwire condition remains based on ρ_t , since semantic drift is better captured by the

total probability mass surviving the constraint than by any single token alone.

4.3 Resolving the Conflict: AST-Guided Intent Recovery

When the tripwire condition $\mathcal{W}(t) = 1$ is triggered, the decoder has reached a state in which the FSM-admissible continuation set contains too little of the model's unconstrained probability mass. Directly sampling from the remaining valid tokens would preserve syntactic admissibility, but risks pushing the model into a low-probability continuation region and thereby amplifying semantic drift. To avoid this, the architecture activates the *Slow Path*: an AST-guided recovery mechanism that rewinds generation to a structurally meaningful checkpoint before decoding resumes.

The Slow Path is built around an incremental, non-blocking parser inspired by the partial graph models of Projectional Decoding [1]. As the FSM Fast Path commits tokens to the output stream, these tokens are asynchronously dispatched to a background parser. Rather than requiring a complete parse at every decoding step, the parser maintains a partial Abstract Syntax Tree, denoted as $\mathcal{G}_{<t}$, representing the macro-structural state of the generated prefix. When the parser encounters an incomplete syntactic region, such as an open function declaration or an unresolved block, it inserts a *frontier node* n_f into the graph. A frontier node acts as a placeholder for an open scope, allowing the parser to track incomplete program structure without relying on expensive dummy closures.

The partial AST is updated asynchronously through the graph update function

$$\mathcal{G}_{<t+1} = \mathcal{U}(\mathcal{G}_{<t}, x_t).$$

Since $\mathcal{U}$ runs outside the token-critical path, the FSM does not wait for AST construction before computing the next token mask. The Slow Path therefore functions as an asynchronous structural memory: it continuously consumes committed tokens and updates the partial AST in the background, but it is not queried during ordinary decoding. Only when the tripwire identifies a high-conflict state does the Fast Path synchronize with this background representation to select a stable rollback checkpoint.

Once $\mathcal{W}(t) = 1$, recovery requires more than reverting by a fixed number of tokens. A naive rollback to $t - k$ may cut through a function signature, variable declaration, argument list, or block scope, producing a prefix that is syntactically valid only in a superficial sense but structurally unstable for regeneration. Instead, the proposed method uses the partial AST to identify the nearest checkpoint from which the model can plausibly resume a coherent continuation.

To connect the sequential token stream with the partial AST, we maintain a mapping

$$\phi : i \mapsto n_j,$$

which associates each committed token index i with the AST node $n_j \in \mathcal{G}_{<t}$ that it contributes to. Since the tripwire is evaluated before committing a new token at step t , the conflict is localized to the current generation context rather than to a newly emitted token. We therefore define the conflict node as the active terminal or frontier node associated with the current prefix:

$$n_c = \phi(t - 1).$$

Recovery proceeds by traversing the ancestor path of n_c and selecting the deepest node that satisfies a stability predicate Ψ . The predicate is intended to capture both syntactic coherence and suitability for regeneration. We define $\Psi(n) = 1$ if the node satisfies the following conditions: (i) it corresponds to a syntactically coherent prefix boundary, (ii) the FSM state can be reconstructed by replaying the token prefix ending at that node, and (iii) the node represents a meaningful structural checkpoint, such as a completed statement, a function signature, a block initialization, or another scope boundary from which generation can resume without severing an active declaration.

The recovery target is then defined as

$$n_{\text{safe}} = \arg \max_{n \in \text{Ancestors}(n_c)} \text{Depth}(n) \quad \text{subject to} \quad \Psi(n) = 1.$$

Once n_{safe} is identified, the system maps it back to its corresponding token index,

$$t_{\text{safe}} = \max\{i \mid \phi(i) = n_{\text{safe}}\},$$

and rolls back the auto-regressive context to the prefix

$$x_{\leq t_{\text{safe}}}.$$

The generated sequence, partial AST, decoding index, KV-cache, and FSM state are restored consistently to this checkpoint. The FSM state is reconstructed by replaying the retained prefix through the automaton, after which the logits are recomputed for position $t_{\text{safe}} + 1$. This ensures that regeneration resumes from a prefix that is both structurally admissible and suitable for recovery.

To prevent the decoder from immediately re-entering a branch that previously led to a conflict, we maintain a checkpoint-local failed-choice set. For each rollback checkpoint n_{safe} , let

$$V_{\text{failed}}(n_{\text{safe}})$$

denote the set of first token choices previously emitted at position $t_{\text{safe}} + 1$ whose resulting continuations later triggered the tripwire. If the discarded branch began with token $x_{t_{\text{safe}}+1}$, the set is updated as

$$V_{\text{failed}}(n_{\text{safe}}) \leftarrow V_{\text{failed}}(n_{\text{safe}}) \cup \{x_{t_{\text{safe}}+1}\}.$$

During regeneration from the same checkpoint, the logits at the recovery junction are modified as

$$z'_{t_{\text{safe}}+1}(v) = z_{t_{\text{safe}}+1}(v) - \lambda \cdot \mathbb{I}[v \in V_{\text{failed}}(n_{\text{safe}})].$$

The penalty therefore targets only branch-entry choices known to have produced a low-compatibility continuation from this checkpoint, rather than penalizing every token that appeared somewhere in the discarded suffix. Unlike an absolute ban, it discourages repeated exploration of failed branches while still leaving the full FSM-admissible search space available.

By replacing fixed-depth token retraction with AST-guided rollback, the Slow Path addresses semantic drift without abandoning syntactic control. The FSM continues to enforce hard structural admissibility, while the partial AST provides a meaningful checkpoint from which the decoder can recover when the valid continuation set becomes distributionally misaligned with the model's preferred trajectory. Thus, recovery is not merely a structural repair step; it is a controlled attempt to restore alignment by returning generation to the nearest stable program boundary.

5 Theoretical Evaluation Across the Trilemma Axes

To analyze the proposed Fast-Path/Slow-Path architecture, we evaluate its theoretical behavior across the three axes of the Constrained Decoding Trilemma: inference latency, structural admissibility, and intent preservation.

5.1 Axis 1: Inference Latency and Computational Complexity

The primary operational flaw of dynamic semantic frameworks such as ChopChop is their synchronous evaluation overhead: if deep structural or semantic constraints are evaluated at every decoding step, the per-token cost can scale with the length or complexity of the generated prefix. Conversely, token-level backtracking systems such as AdapTrack introduce potentially expensive recovery procedures when the decoder must explore alternative branches after a constraint conflict.

Our architecture reduces this cost by separating the common decoding path from the exceptional recovery path. Let T_{Fast} denote the computational time of the FSM state transition and mask construction. Since the valid-token set is obtained from the current automaton state rather than by reparsing the full prefix, this cost is treated as constant in the decoding-critical path:

$$T_{\text{Fast}} = O(1).$$

To model the recovery cost more precisely, let p_t denote the largest token index that has already been incorporated into the asynchronous partial AST at decoding step t . We define the parser lag as

$$\ell_t = t - p_t.$$

The intended latency advantage of the Slow Path depends on the assumption that incremental AST maintenance can keep this lag bounded in the common case.

When the tripwire activates, the system may need to wait for the parser to incorporate any missing suffix of length ℓ_t , synchronize with the partial AST, locate the safe rollback node n_{safe} , truncate the KV-cache, and reconstruct the FSM state. The practical recovery cost is therefore modeled as

$$T_{\text{Slow}} = O(\ell_t) + O(D_{\text{AST}}) + \Delta_{\text{sync}} + T_{\text{KV}},$$

where D_{AST} is the depth of the relevant AST ancestor path, Δ_{sync} denotes synchronization overhead, and T_{KV} denotes the cost of KV-cache truncation or restoration.

The expected per-token cost can then be written in terms of the tripwire activation probability:

$$E[T] = (1 - P(\mathcal{W}(t) = 1)) \cdot T_{\text{Fast}} + P(\mathcal{W}(t) = 1) \cdot T_{\text{Slow}}.$$

With the distributional tripwire introduced in Section 4.2, this activation probability corresponds to

$$P(\mathcal{W}(t) = 1) = P(\rho_t \leq \tau_t),$$

where ρ_t is the valid probability mass left by the FSM constraint and τ_t is the entropy-dependent drift threshold.

In the common case, the FSM-admissible continuation set remains sufficiently aligned with the model's unconstrained distribution, so $\rho_t > \tau_t$ and the Slow Path is not invoked. Under the

additional assumptions that parser lag remains bounded and tripwire activations are sparse, the expected cost is dominated by the Fast Path:

$$E[T] \approx T_{\text{Fast}} = O(1).$$

Thus, the architecture preserves constant-time decoding behavior in the common case while allowing bounded, selective recovery when distributional drift is detected. This does not eliminate the worst-case cost of recovery: if tripwire activations become frequent, parser lag grows with the prefix length, or KV-cache rollback is inefficient, the latency advantage weakens. Rather, the design shifts expensive structural recovery out of the synchronous token path and invokes it only when the FSM constraint becomes distributionally misaligned with the model's intended continuation.

5.2 Axis 2: Structural Admissibility Guarantees

While stochastic generation inherently risks syntax errors, constrained decoding frameworks are designed to prevent the decoder from emitting tokens that violate the target grammar. In our architecture, this responsibility is assigned exclusively to the Fast Path. The Slow Path may guide rollback and recovery, but it never bypasses the FSM mask. This separation ensures that semantic recovery remains subordinate to syntactic admissibility.

As established in Section 3.4, DCFG-based and automaton-based validators are semantically limited: they cannot in general enforce global program properties such as type correctness, API validity, or data-flow consistency. However, they are well suited for enforcing baseline syntactic structure at decoding time. Our architecture therefore uses the FSM not as a complete semantic validator, but as a hard syntactic guardrail that remains active during both ordinary decoding and regeneration after rollback.

We formalize the target language's baseline syntax as a grammar

$$G = (N, \Sigma, R, S),$$

where N is the set of non-terminal symbols, Σ is the terminal alphabet, R defines the production rules, and S is the start symbol. The Fast Path compiles this grammar into an automaton-style completion engine. Let the internal state of this engine at generation step t be s_t , with transition function δ . At each step, the validator computes the structurally admissible token subset

$$V_{\text{valid}}(s_t) = \{v \in V \mid \delta(s_t, v) \text{ is defined}\}.$$

The FSM mask is then applied to the unconstrained logits:

$$M_{\text{FSM}}(v) = \begin{cases} 0, & \text{if } v \in V_{\text{valid}}(s_t), \\ -\infty, & \text{otherwise.} \end{cases}$$

The resulting constrained distribution is

$$\tilde{P}_\theta(x_t = v \mid x_{<t}) = \text{softmax}(z_t + M_{\text{FSM}})_v.$$

Since tokens outside $V_{\text{valid}}(s_t)$ receive probability zero after masking, the decoder cannot sample a token that is inadmissible with respect to the compiled grammar state. Thus, under the assumption that the grammar and token-to-grammar interface correctly encode the target syntax, the Fast Path preserves syntactic admissibility throughout generation.

The same principle applies during AST-guided recovery. A potential weakness of naive token-count backtracking is that reverting to an arbitrary index $t - k$ may cut through a declaration, expression,

or lexical unit, leaving the decoder in a prefix state that is difficult to reconcile with the automaton. Our architecture avoids this by rolling back only to a node n_{safe} satisfying the stability predicate $\Psi(n_{\text{safe}}) = 1$. In particular, Ψ requires that the corresponding prefix be replayable through the FSM.

Let t_{safe} denote the token index associated with n_{safe} . The recovery prefix must satisfy

$$\exists s_{\text{safe}} \text{ such that } \delta^*(S, x_{\leq t_{\text{safe}}}) = s_{\text{safe}},$$

where δ^* is the extended transition function obtained by replaying the retained prefix from the initial state. After rollback, the FSM is reinitialized to s_{safe} , and all subsequent tokens are again generated under the mask M_{FSM} .

Consequently, the Slow Path does not weaken the structural guarantees of the Fast Path. Its role is to select a structurally meaningful rollback point, not to relax the grammar. The architecture therefore preserves syntactic admissibility while allowing semantic recovery to occur only at prefixes that can be reconstructed as valid FSM states.

5.3 Axis 3: Semantic Intent Preservation

The main weakness of high-efficiency FSM-based constrained decoding is not syntactic reliability, but semantic rigidity. Because the FSM only encodes admissible grammar transitions, it cannot determine whether the remaining valid tokens still align with the model's intended continuation. To evaluate this failure mode, we model local intent through the LLM's unconstrained next-token distribution $P_\theta(\cdot | x_{<t})$. Semantic intent is therefore not assumed to be directly observable; instead, it is approximated by the probability mass assigned by the model before structural masking is applied.

In a pure FSM decoder, the constrained distribution is obtained by applying the FSM mask to the unconstrained logits:

$$\tilde{P}_\theta(x_t = v | x_{<t}) = \text{softmax}(z_t + M_{\text{FSM}})_v.$$

The distortion introduced by this mask can be characterized through the amount of unconstrained probability mass that survives the constraint. Let

$$\rho_t = \sum_{v \in V_{\text{valid}}(s_t)} P_\theta(v | x_{<t})$$

denote the valid probability mass. Since the constrained distribution is obtained by renormalizing the surviving probabilities by ρ_t , the corresponding renormalization pressure can be written as

$$\kappa_t = -\log \rho_t.$$

When ρ_t is high, the FSM preserves most of the model's preferred continuation space and κ_t remains small. When ρ_t is low, the surviving tokens must be strongly renormalized, indicating that the decoder is being forced into a region that the model originally considered unlikely. This does not prove that the blocked continuation was semantically correct, but it provides a direct operational measure of conflict between the model distribution and the structural constraint.

The distributional tripwire introduced in Section 4.2 detects this situation before the decoder samples from the distorted distribution. The condition

$$\mathcal{W}(t) = 1 \iff \rho_t \leq \tau_t$$

acts as an early warning signal that the FSM-admissible region contains too little of the model's unconstrained probability mass.

Once the tripwire fires, the architecture avoids immediately sampling from this low-mass region. Instead, it activates the AST-guided Slow Path defined in Section 4.3 and rolls generation back to the nearest stable checkpoint n_{safe} . This improves intent preservation in two ways. First, it prevents the decoder from continuing directly through a strongly renormalized continuation space. Second, it returns the model to a meaningful program boundary where alternative continuations can be explored without cutting through an active declaration, expression, or scope.

At the recovery junction, the checkpoint-local penalty defined in Section 4.3 is applied:

$$z'_{t_{\text{safe}}+1}(v) = z_{t_{\text{safe}}+1}(v) - \lambda \cdot \mathbb{I}[v \in V_{\text{failed}}(n_{\text{safe}})].$$

This discourages branch-entry choices that previously led to conflict while preserving the remaining FSM-admissible search space.

Consequently, the proposed architecture preserves semantic intent in an operational, distributional sense: it monitors whether the FSM-valid region remains compatible with the model's unconstrained continuation, interrupts generation when compatibility falls below an adaptive threshold, and resumes decoding from a stable program checkpoint. This does not guarantee recovery of the developer's true intent, which is not directly observable during decoding. However, it mitigates the central failure mode of pure FSM decoding: blindly renormalizing over a syntactically valid but distributionally implausible continuation set.

6 Limitations

While the Fast-Path/Slow-Path architecture theoretically addresses the tension between latency, syntactic admissibility, and semantic intent preservation, its practical implementation introduces several limitations. These limitations mainly concern tripwire calibration, synchronization overhead, KV-cache management, and the availability of robust incremental parsers.

6.1 Hyperparameter Sensitivity

The architecture relies on three scalar hyperparameters: α and β control the tripwire threshold

$$\tau_t = \beta + \alpha(1 - \hat{H}_t),$$

while λ controls the checkpoint-local branch penalty

$$z'_{t_{\text{safe}}+1}(v) = z_{t_{\text{safe}}+1}(v) - \lambda \cdot \mathbb{I}[v \in V_{\text{failed}}(n_{\text{safe}})].$$

These parameters determine how aggressively the system detects distributional drift and how strongly it avoids previously failed branches during regeneration. If α or β is too high, the system may trigger unnecessary Slow Path activations, weakening the latency advantage of the Fast Path. If they are too low, the decoder may behave like a pure FSM system and continue through low-mass valid regions. Similarly, if λ is too small, regeneration may repeat the same failed trajectory; if it is too large, the decoder may over-penalize plausible continuations and force unnatural alternatives.

Thus, the method requires task- or language-specific calibration. Future work should investigate adaptive tuning strategies that adjust α , β , and λ based on the local distribution shape, the observed

valid probability mass ρ_t , and the frequency of repeated recovery events.

6.2 Slow-Path Implementation Dependencies

The expected-time analysis in Section 5 assumes that the Slow Path can be queried with bounded overhead. In practice, this depends on three implementation conditions: bounded parser lag, efficient KV-cache rollback, and reliable incremental parsing.

First, the asynchronous parser must remain close to the token stream. Let p_t denote the latest token index incorporated into the partial AST and let $\ell_t = t - p_t$ be the parser lag. If ℓ_t remains small, recovery only requires limited catch-up before querying a checkpoint. However, if CPU resources are saturated, the target language is difficult to parse incrementally, or tripwire activations are frequent, ℓ_t may grow with the prefix length and weaken the latency advantage. The decoder must also synchronize with the parser to retrieve n_{safe} , introducing a non-zero synchronization cost Δ_{sync} .

Second, AST-guided recovery requires rolling back the autoregressive context to $x_{\leq t_{\text{safe}}}$. This is not merely a textual truncation: the discarded tokens' KV-cache entries must also be removed or invalidated. If the serving engine does not support efficient cache truncation, prefix reuse, or tree/radix-based KV-cache management, rollback may require recomputing part of the prefix and become a major bottleneck.

Finally, the quality of recovery depends on the reliability of the incremental parser and the stability predicate Ψ . Languages with complex preprocessing, macro expansion, dynamic scoping, indentation-sensitive syntax, or context-dependent grammar rules may make partial AST construction less reliable. If the parser assigns incorrect frontier nodes or unstable boundaries, the selected node n_{safe} may remain replayable through the FSM but still be semantically unsuitable for recovery. Thus, the practical effectiveness of the Slow Path depends directly on bounded parser lag, efficient cache management, and robust incremental parsing.

6.3 Semantic Intent Remains Only Approximately Observable

A final limitation is that the architecture does not directly observe the developer's true intent. The distributional tripwire treats the unconstrained model distribution as an operational proxy for local semantic intent. This is useful for detecting when FSM masking strongly distorts the model's preferred continuation, but the model's preferred continuation may itself be incorrect, hallucinated, or inconsistent with the user prompt.

Consequently, low valid probability mass ρ_t indicates a conflict between the model and the constraint system, not necessarily proof that the FSM is blocking a semantically correct continuation. The Slow Path can mitigate semantic drift caused by rigid masking, but it cannot guarantee semantic correctness in the stronger sense of functional correctness, type correctness, or alignment with the developer's full specification. These stronger notions would require additional signals, such as type checkers, unit tests, execution feedback, or repository-level semantic analysis.

7 Conclusion

This paper addressed the tension between inference latency, syntactic admissibility, and semantic intent preservation in constrained code generation. We proposed a Fast-Path/Slow-Path architecture in which a synchronous FSM enforces grammar-level admissibility, while an asynchronous partial AST supports selective recovery when distributional drift is detected.

The central mechanism is the valid probability mass ρ_t , which measures how much of the model's unconstrained distribution remains available after FSM masking. When $\rho_t \leq \tau_t$, the system rolls back to a stable AST checkpoint and resumes generation with a checkpoint-local branch penalty. This allows the decoder to preserve the speed and structural rigor of FSM-based masking while avoiding the main failure mode of pure FSM decoding: forcing generation through syntactically valid but distributionally implausible continuations.

By separating syntactic enforcement from semantic recovery, the proposed architecture provides a practical path toward constrained decoding systems that are fast, structurally reliable, and less prone to intent distortion. Future work should investigate adaptive tripwire calibration, efficient KV-cache rollback, and robust incremental parsing across more complex programming languages.

References

- [1] Boqi Chen, José Antonio Hernández López, and Aren A. Babikian. 2026. Projectional Decoding: Towards Semantic-Aware LLM Generation. arXiv:2605.30054 [cs.SE] <https://arxiv.org/abs/2605.30054>
- [2] Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. 2026. XGrammar-2: Dynamic and Efficient Structured Generation Engine for Agentic LLMs. In *Proceedings of the ACM Conference on AI and Agentic Systems (CAIS '26)*. ACM, 1009–1022. doi:10.1145/3786335.3813124
- [3] Yuan Li, Jia Li, Ge Li, and Zhi Jin. 2025. AdapTrack: Constrained Decoding without Distorting LLM's Output Intent. *arXiv preprint arXiv:2510.17376* (2025). doi:10.48550/arxiv.2510.17376
- [4] Niels Münder, Jingxuan He, Haotian Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proceedings of the ACM on Programming Languages* (2025). doi:10.1145/3729274
- [5] Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Lorin D'Antoni. 2025. Chop-Chop: a Programmable Framework for Semantically Constraining the Output of Language Models. arXiv:2509.00360 [cs.PL] <https://arxiv.org/abs/2509.00360>
- [6] 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. arXiv:2201.11227 [cs.LG] <https://arxiv.org/abs/2201.11227>
- [7] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. arXiv:2109.05093 [cs.CL] <https://arxiv.org/abs/2109.05093>
- [8] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] <https://arxiv.org/abs/2312.07104>

A Decoding Procedure

Algorithm 1 Tiered Constrained Decoding

Require: LLM, FSM state s_1 , partial AST $\mathcal{G}_{<1}$, thresholds α, β , penalty λ

Ensure: FSM-admissible output X , or failure if no accepting completion is found

```

1: Initialize  $t \leftarrow 1, X \leftarrow \emptyset$ 
2: while  $t \leq N$  and  $s_t$  is not accepting do
3:   Compute logits  $z_t$  and unconstrained distribution  $P_\theta(\cdot \mid x_{<t})$ 
4:   Compute  $V_{\text{valid}}(s_t)$  and valid probability mass  $\rho_t$ 
5:   Compute  $\tau_t = \beta + \alpha(1 - \hat{H}_t)$ 
6:   if  $\rho_t \leq \tau_t$  then
7:     Query the asynchronous AST and identify conflict node
        $n_c = \phi(t - 1)$ 
8:     Select  $n_{\text{safe}} = \arg \max_{n \in \text{Ancestors}(n_c)} \text{Depth}(n)$  subject to
        $\Psi(n) = 1$ 
9:     Set  $t_{\text{safe}} = \max\{i \mid \phi(i) = n_{\text{safe}}\}$ .
10:    Set  $v_{\text{branch}} \leftarrow x_{t_{\text{safe}}+1}$ 
11:    Update  $V_{\text{failed}}(n_{\text{safe}}) \leftarrow V_{\text{failed}}(n_{\text{safe}}) \cup \{v_{\text{branch}}\}$ 
12:    Restore  $X \leftarrow x_{\leq t_{\text{safe}}}$  and  $\mathcal{G}_{<t} \leftarrow \mathcal{G}_{\leq n_{\text{safe}}}$ 
13:    Set  $t \leftarrow t_{\text{safe}} + 1$ 
14:    Truncate the KV-cache and reconstruct the FSM state
        $s_t$ 
15:    Recompute logits  $z_t$  from the restored prefix  $x_{<t}$ 
16:    Apply  $z_t(v) \leftarrow z_t(v) - \lambda \mathbb{I}[v \in V_{\text{failed}}(n_{\text{safe}})]$ 
17:  end if
18:  Compute  $V_{\text{valid}}(s_t)$  and apply the FSM mask  $M_{\text{FSM}}$ 
19:  Sample  $x_t$  from  $\text{softmax}(z_t + M_{\text{FSM}})$ 
20:  Append  $x_t$  to  $X$  and update  $s_{t+1} = \delta(s_t, x_t)$ 
21:  Dispatch  $x_t$  to the asynchronous AST parser
22:   $t \leftarrow t + 1$ 
23: end while

```
