Article

Not peer-reviewed version

Cascaded Neurosymbolic Code Generation for Niche DSLs: Preserving Chain-of-Thought in Grammar-Constrained Decoding

Rubén Ruiz-Torrubiano *, Himanshu Buckchash, Sarita Paudel, Deepak Dhungana

Posted Date: 12 June 2026

doi: 10.20944/preprints202606.1011.v1

Keywords: neurosymbolic AI; large language models; domain-specific languages; code generation; constraint programming; grammar-constrained decoding

 doi:10.20944/preprints202606.1011.v1

Article

Cascaded Neurosymbolic Code Generation for Niche DSLs: Preserving Chain-of-Thought in Grammar-Constrained Decoding

Rubén Ruiz-Torrubiano *, Himanshu Buckchash, Sarita Paude and Deepak Dhungana

Department of Science and Technology, IMC University of Applied Sciences Krems, Krems, Austria
* Correspondence: ruben.ruiz@imc.ac.at

Abstract

Domain-Specific Languages (DSLs) are essential in software engineering for safely expressing complex domain logic. However, Large Language Models (LLMs) struggle to generate syntactically and semantically correct code for niche DSLs due to sparse representation in pre-training corpora. While Grammar-Constrained Decoding (GCD) resolves syntactic hallucinations by masking logits through a formal Context-Free Grammar (CFG), empirical evidence shows that strict GCD disrupts the autoregressive Chain-of-Thought (CoT) reasoning of modern models, frequently forcing them into irreversible semantic dead-ends. To overcome the friction between internal neural reasoning and external symbolic constraints, we propose a Dual-Phase Cascaded Neurosymbolic framework. In the first phase, the model is provided with dynamically injected grammar rules and is permitted to reason unconstrained, producing an optimistic code draft. If the draft fails native compiler checks, the system enters a second phase: it preserves the successful semantic reasoning from Phase 1 but re-generates the code under strict GCD enforcement. This cascaded architecture utilizes the formal FSM not as an adversarial constraint, but as a localized syntax repair engine guided by the model's own prior reasoning. We construct a comprehensive benchmark of 100 MiniZinc constraint programming tasks and evaluate our approach using a *pass@k* metric with a strict semantic LLM judge. Our findings demonstrate that this dual-phase "think-then-constrain" approach significantly outperforms zero-shot, pure few-shot, and pure GCD baselines, achieving highly reliable, training-free code generation for unseen DSLs.

Keywords: neurosymbolic AI; large language models; domain-specific languages; code generation; constraint programming; grammar-constrained decoding

1. Introduction

Large Language Models (LLMs) have precipitated a paradigm shift in software engineering, demonstrating unprecedented capabilities in code generation, translation, and debugging [1,2]. For general-purpose programming languages such as Python, Java, and C++, these models benefit from massive representation in pre-training corpora, allowing them to internalize both syntactic rules and complex semantic patterns. However, modern software architecture frequently relies on Domain-Specific Languages (DSLs) to safely and concisely express specialized domain logic. From hardware description (e.g., Verilog) and constraint programming (e.g., MiniZinc [3]) to proprietary enterprise configuration languages, DSLs enforce strict structural rules that prevent domain-specific errors. Unfortunately, LLMs consistently struggle to generate reliable code for these niche DSLs. Due to data sparsity, models frequently hallucinate syntax, invent non-existent operators, or fail to respect the rigid typing rules required by domain compilers. Because continuously fine-tuning foundation models for every new or proprietary DSL is economically unviable and technically cumbersome [4], training-free (zero-shot and few-shot) generation strategies remain highly desirable.

 doi:10.20944/preprints202606.1011.v1

To address syntactic hallucinations without fine-tuning, the neurosymbolic integration of Grammar-Constrained Decoding (GCD) has emerged as a leading technique [5]. By compiling a formal Context-Free Grammar (CFG) or Extended Backus-Naur Form (EBNF) into a Finite State Machine (FSM), GCD dynamically masks the LLM's output logits at each generation step. This physically prevents the model from generating any sequence of tokens that violates the target language's syntax. While GCD effectively guarantees a 100% syntactic pass rate, recent empirical observations reveal a severe limitation when applied to modern reasoning models: a degradation of semantic accuracy.

Advanced LLMs, particularly those optimized via Reinforcement Learning for reasoning (e.g., DeepSeek-R1, OpenAI o1), achieve high semantic accuracy by utilizing long-form Chain-of-Thought (CoT) generation [6,7]. They "think out loud," planning variables, logic, and constraints before emitting final code. Standard GCD architectures force the LLM to begin generating strictly valid grammar tokens immediately, stripping the model of its unconstrained reasoning space. Consequently, the model generates code blindly and greedily. If it makes a poor semantic choice early in the generation (e.g., declaring a variable as a boolean instead of an integer), the strict FSM forces it to complete the syntactically legal sequence, irreversibly driving the output into a "semantic dead-end" that fails subsequent compiler verification.

To overcome the friction between internal neural reasoning and external symbolic constraints, this paper proposes a *Dual-Phase Cascaded Neurosymbolic Framework*. Instead of applying constraints indiscriminately, our approach isolates semantic planning from syntactic enforcement. In Phase 1, the model is provided with the target EBNF grammar via the prompt but is permitted to reason unconstrained (e.g., within a `<think>` block), producing an optimistic draft of the code. This draft is instantly evaluated using native DSL compiler checks. If the optimistic draft passes (the "Fast Path"), the generation successfully concludes in $\mathcal{O}(1)$ time. If the draft exhibits syntactic hallucinations, the system enters Phase 2: it preserves the successful, unconstrained semantic reasoning from Phase 1, appends it to the context window, and re-generates the code under strict GCD enforcement.

The explicit contributions of this paper are as follows:

- We identify and empirically demonstrate the "Semantic-Syntax Trade-off" in modern reasoning LLMs, showing how strict Grammar-Constrained Decoding disrupts Chain-of-Thought planning and induces semantic dead-ends in niche DSLs.
- We present a novel Dual-Phase Cascaded Framework that implements Optimistic Bypassing. By preserving unconstrained reasoning from a failed draft and using it to guide a strict GCD fallback, our architecture effectively utilizes the symbolic grammar as a localized syntax repair engine.
- We construct and openly release a comprehensive benchmark dataset comprising 100 natural language to MiniZinc constraint programming tasks of varying complexities.
- We conduct a rigorous empirical evaluation using a *pass@k* metric, scored by a strict semantic LLM judge and native compiler. Our results demonstrate that the dual-phase "think-then-constrain" approach significantly outperforms zero-shot, pure few-shot, and pure GCD baselines for unseen DSL code generation.

The remainder of this article is organized as follows. Section 2 reviews related literature on LLM code generation, reasoning models, and grammar-constrained decoding. Section 3 details the proposed Dual-Phase Cascaded architecture, including the prompt injection of formal EBNF structures and the optimistic bypassing mechanism. Section 4 describes the experimental setup, introduces the MiniZinc benchmark, and presents the comparative results against established baselines. Finally, Section 5 summarizes our findings and outlines directions for future research.

2. Previous Work

On ordinary code generation tasks, in general LLMs are often good at producing code that is at least syntactically well formed, especially for popular languages and strong code-tuned models [1,2,8,9]. The wider view in the literature is that larger code models implicitly learn syntax rules well

 doi:10.20944/preprints202606.1011.v1

enough that outright syntax errors are becoming rarer, even in relatively small code-specialized models [10].

Most recent studies on code generation for domain-specific languages focus on comparing different LLM configurations (pre-trained vs. fine-tuned, zero-shot vs. few-shot), LLM-based approaches (RAG vs. self-debugging vs. reinforcement learning), and DSLs versus general-purpose languages. In [11,12], the authors compared fine-tuned LLMs for generating Verilog code in the context of hardware design. They found that fine-tuning was able to increase the capability of the models to produce syntatically correct code. Fine-tuning vs. optimized RAG was investigated in [13] in the context of automation task representation, showing that both approaches are prone to frequently generating syntactically erroneous code. The authors of [14] apply a small language model (SLM) fine-tuned using reinforcement learning to fix syntactic errors in LLM-generated code, and evaluate their approach for the Ansible, Bash and SQL languages. Their approach was shown to achieve superior performance than other repair-based approaches. A general framework for assessing the effectiveness of LLMs for DSL code generation is proposed in [15]. The authors find that, in general, the performance of LLMs for generating code in general-purpose programming languages like Python is better than for constraint programming (OCL, Alloy). They propose improvements in the code generation process like code repair or multiple attempts that can improve the quality of the generated code.

The authors of [16] argue that existing pretrained models still lack true understanding of code syntax, failing to match simple baselines focused on keywords and offsets. Compared with traditional generators, the summary is that LLMs now frequently achieve high syntactic correctness on common code, but they still do not offer the same consistency or guarantees, and syntax failures remain common enough to matter in practice [4]. The clearest pattern in the DSL and low-resource literature is that syntactic correctness gets worse as the target language moves away from mainstream training data. Very low-resource programming languages are a direct example: models were found to struggle to compose syntactically valid programs in languages not represented in pretraining, and for the UCLID5 formal verification language, a special method called SPEAC generated syntactically correct programs more often than retrieval and fine-tuning baselines[17]. Hardware and other technical DSL-like code targets show the same tradeoff. For RTL and Verilog, prior work suggests strong LLMs can often handle the surface syntax because HDL looks somewhat like ordinary programming languages, but this does not translate into reliable overall generation without domain-specific processing and fine-tuning [12,18,19].

Recent work has focused on grammar constrained decoding (GDC) as one specific way of enforcing syntactic constraints at the LLM decoding stage [5,20]. In it essence, GDC approaches mask token completions during generations which do not correspond to the provided grammar, usually in EBNF form. This can be done, for instance, by formulating the completions allowed by the grammar as a finite state machine (FSM) [5]. However, GDC can distort the LLM's generation distribution, biasing the decoding process towards outputs which are grammatically correct but with a likelihood that is not proportional to the ones that would have been selected by the LLM, resulting in poor semantic quality [21].

In general, producing syntactically correct code represents only a pre-condition for semantic correctness. Some papers note that deep learning and LLM systems usually can produce syntactically correct code more often than they can produce code which is fully aligned with the task description [22,23]. This aspect remains underexplored in the current literature. One of the few works in this regard is [24], where the authors focus on enforcing semantic constraints by integrating token-level Monte Carlo Tree Search (MCTS) with Answer Set Grammars (ASG). They show that their approach allows small pretrained LLMs to outperform starte-of-the-art reasoning models and guarantee semantic validity.

In the context of this existing literature, our proposed Dual-Phase Cascaded architecture addresses a critical intersection of these open challenges. While prior works rely on computationally expensive fine-tuning or token-level search to enforce syntax and semantics [12,24], we aim to provide

 doi:10.20944/preprints202606.1011.v1

a lightweight, training-free alternative. Specifically, our approach directly addresses the semantic degradation induced by standard GCD [21]. By isolating unconstrained semantic planning (Chain-of-Thought) from syntactic enforcement (the FSM logits mask), our framework prevents the grammar constraint from distorting the LLM's natural generation distribution during the critical reasoning phase. Furthermore, by introducing an optimistic fast-path verified by native compilers, our method guarantees syntactic validity while actively prioritizing semantic alignment, bridging the gap between rigid formal constraints and the fluid reasoning capabilities of modern foundation models.

3. Methodology

The core premise of our approach is to decouple semantic planning from syntactic enforcement. By allowing modern reasoning-optimized Large Language Models (LLMs) to construct a semantic plan without constraints, and subsequently enforcing strict syntactic adherence only when necessary, we construct a robust, training-free neurosymbolic architecture. This section details the Dual-Phase Cascaded Framework, detailing the prompt formulation, the optimistic bypassing mechanism, and the Grammar-Constrained Decoding (GCD) fallback.

3.1. In-Context Symbolic Grounding

A fundamental challenge in zero-shot or few-shot code generation for Domain-Specific Languages (DSLs) is the model's reliance on pre-trained biases, which often contradict the specific rules of the target DSL. To mitigate this, our framework employs *In-Context Symbolic Grounding*. Rather than relying on the LLM to infer the syntax from examples alone, we explicitly inject the raw Extended Backus-Naur Form (EBNF) grammar directly into the system prompt.

Alongside the EBNF grammar, the prompt is enriched with a semantic operator cheat sheet (Action Aliasing) and a curated set of representative few-shot examples. By forcing the LLM to observe the exact derivation rules prior to generation, we align the model's internal representations with the strict constraints it will encounter. This ensures that when the LLM begins its Chain-of-Thought (CoT) reasoning, it actively traces its logic through the provided formal grammar, mapping natural language intents directly to valid non-terminal expansions.

3.2. Phase 1: Unconstrained Reasoning and Optimistic Bypassing

The first phase of the cascaded architecture capitalizes on the native strengths of reasoning LLMs (e.g., DeepSeek-R1). The model is instructed to encapsulate its step-by-step planning inside explicit `<think>...</think>` tags before outputting the actual code. During this phase, generation is completely unconstrained; standard autoregressive decoding is used. This allows the model to freely explore the semantic space, define variable relations, and plan optimization bounds without being artificially truncated by a logits mask.

Following the reasoning block, the model generates an initial code draft. To maximize computational efficiency, we introduce *Optimistic Bypassing*. The generated draft is immediately subjected to two deterministic gatekeepers:

1. **Syntactic Gate:** A fast Context-Free Grammar parser (e.g., Lark) verifies that the generated string perfectly conforms to the EBNF structure.
2. **Semantic Gate:** The draft is passed to the native DSL compiler in type-checking mode. Because the compiler evaluates the code globally, it instantly flags any type mismatches, uninitialized variables, or invalid operator overloads.

If the draft passes both gates, the system recognizes that the unconstrained LLM successfully handled both the semantics and the syntax. The generation halts, returning the valid code in $\mathcal{O}(1)$ generation steps. This "Fast Path" prevents wasting computational resources on strict constraints when the model's native capability is sufficient.

 doi:10.20944/preprints202606.1011.v1

3.3. Phase 2: Grammar-Constrained Syntax Repair

If the optimistic draft fails either the syntactic or semantic gate, it indicates that while the model may have reasoned correctly, its translation into code suffered from structural hallucinations. In standard pipelines, such failures result in discarded outputs. In our cascaded framework, we discard the malformed code but *preserve the model's internal reasoning*.

The successful `<think>` block generated in Phase 1 is captured and appended to the context window, acting as a highly detailed, task-specific semantic blueprint. The system then initiates Phase 2: generating the code block under strict Grammar-Constrained Decoding. A Finite State Machine (FSM) compiled from the DSL's EBNF grammar masks the output logits at every decoding step. If the LLM assigns a high probability to an invalid token (e.g., hallucinating a Python-style `and` instead of the DSL's `/\`), the FSM forces the logit to $-\infty$, compelling the model to select the most probable *valid* token.

Because the LLM's Context Window already contains its own step-by-step reasoning, it does not "fight" the GCD mechanism. Instead, the constraint acts as a localized syntax repair engine, seamlessly mapping the LLM's preserved semantic intent onto the mathematically guaranteed syntactic structure.

3.4. Algorithmic Formalization

The complete Dual-Phase Cascaded generation process is formalized in Algorithm 1.

Algorithm 1 Dual-Phase Cascaded Neurosymbolic Generation

Require: Natural language intent I, EBNF Grammar G, Semantic Aliases A, Few-Shot Examples E
Ensure: Syntactically and semantically viable DSL code

1: $P_{base} \leftarrow \text{BuildPrompt}(I, G, A, E)$ ▷ In-Context Symbolic Grounding
2: $P_{phase1} \leftarrow P_{base} +$ `<think>`
▷ *Phase 1: Unconstrained CoT & Optimistic Draft*
3: $Out_1 \leftarrow \text{LLM\_GenerateUnconstrained}(P_{phase1})$
4: $Reasoning, DraftCode \leftarrow \text{ParseOutput}(Out_1)$
▷ *Optimistic Bypassing (Fast Path)*
5: **if** $DraftCode \neq$ **null then**
6: $\quad IsSyntaxValid \leftarrow \text{CFG\_Parse}(DraftCode, G)$
7: $\quad IsSemanticallyValid \leftarrow \text{NativeCompilerCheck}(DraftCode)$
8: $\quad$ **if** $IsSyntaxValid$ **and** $IsSemanticallyValid$ **then**
9: $\quad\quad$ **return** $DraftCode$ ▷ Successfully bypassed Phase 2
10: $\quad$ **end if**
11: **end if**
▷ *Phase 2: Constrained Syntax Repair*
12: $P_{phase2} \leftarrow P_{phase1} + Reasoning +$ `</think>dsl`
13: $FSM \leftarrow \text{CompileFSM}(G)$ ▷ Build masking automaton from grammar
14: $FinalCode \leftarrow \text{LLM\_GenerateConstrained}(P_{phase2}, FSM)$
15: **return** $FinalCode$

Algorithm 1 formalizes the execution flow of this cascaded architecture. The procedure begins by assembling a comprehensive, DSL-agnostic prompt that grounds the LLM using the formal EBNF grammar, semantic aliases, and few-shot examples (Lines 1–2). In Phase 1, the model is queried without logit constraints, producing both its step-by-step Chain-of-Thought reasoning and an optimistic code draft (Lines 4–5). This draft is immediately evaluated by deterministic structural and semantic gatekeepers; if it passes both the CFG parser and the native compiler, the algorithm successfully takes the fast path and returns the code, bypassing further computation (Lines 5–11). However, if the draft fails due to syntactic or structural hallucinations, the algorithm initiates Phase 2. The valid reasoning extracted from Phase 1 is preserved and appended to the context window, while a compiled finite state machine strictly masks the LLM's token generation (Lines 12–14). This forces the final output to mathematically conform to the DSL's syntax, using the model's own unconstrained planning to seamlessly guide the constrained repair.

 doi:10.20944/preprints202606.1011.v1

4. Empirical Evaluation

To rigorously assess the effectiveness of the proposed Dual-Phase Cascaded framework, we designed an experimental pipeline comparing our approach against prevailing LLM-driven code generation strategies. The objective of this evaluation is to quantify the extent to which preserving Chain-of-Thought (CoT) reasoning mitigates the semantic dead-ends induced by strict Grammar-Constrained Decoding (GCD). All the code, benchmarks and utilities used for the empirical evaluation are publicly available[1]. The experiments were carried out on a GPU server with 256 GB RAM, two AMD EPYC SP3 8-core (3.7GHz) and two NVIDIA L40S GPUs.

4.1. Benchmark Dataset Construction

Mainstream code generation evaluation suites, such as HumanEval or MBPP, are heavily biased toward general-purpose programming languages and algorithmic tasks. They are ill-suited for measuring an LLM's capability to generate structurally rigid, niche Domain-Specific Languages. Consequently, we constructed a novel benchmark dataset comprising 100 distinct natural language programming tasks mapped to the MiniZinc constraint programming language. We choose MiniZinc for two reasons: first, it is a niche DSL which is underrepresented in the training corpus of smaller reasoning LLMs in comparison to general-purpose programming languages like Python or C++. This observation was supported empirically by preliminary zero-shot tests using the `Qwen2.5-Coder-1.5B-Instruct` model (see Table 1 for the full results). Second, current state-of-the-art flagship LLMs like `Qwen3.5` already possess a good understanding of MiniZinc code, which allows us to use this model as a semantic LLM judge in the evaluation process. By using a smaller reasoning model that is not proficient in MiniZinc, we make sure that the code generation process is guided by the provided grammar instead of pretrained knowledge.

The benchmark tasks exhibit a progressive gradient of complexity, encompassing simple variable declarations, set definitions, array aggregations, logical implications, and complex optimization objectives (minimizing or maximizing specific variables). Each task in the dataset consists of a natural language intent and a verified "golden" MiniZinc solution, providing a ground-truth reference for automated evaluation.

4.2. Experimental Setup and Baselines

All generation experiments were conducted using `Qwen2.5-Coder-1.5B-Instruct`, an open-weights reasoning model, parameterized to simulate a resource-constrained, local-first software engineering environment. To establish a comprehensive comparative analysis, we evaluated the proposed architecture against three distinct baseline configurations, all utilizing the exact same underlying LLM:

Baseline 1: Zero-Shot Autoregressive

The LLM is prompted solely with the user intent and instructed to output raw MiniZinc code without any provided grammar rules, examples, or constraints. This establishes the absolute baseline capability of the foundation model's pre-training.

Baseline 2: One-Shot (Unconstrained)

The LLM is provided with the structural EBNF grammar, semantic operator aliases, and a single high-quality CoT example. The model is allowed to output its `<think>` reasoning block, but the final code generation is left unconstrained. This evaluates the efficacy of pure prompt engineering.

Baseline 3: One-Shot (GCD Only)

The LLM is provided with the same grammar and example, but is strictly prohibited from outputting a `<think>` block. The generation is immediately subjected to the strict Outlines FSM logits

[1] https://github.com/IMC-UAS-Krems/mcts-niche-dsl

 doi:10.20944/preprints202606.1011.v1

processor. This baseline isolates the performance of standard, state-of-the-art GCD architectures that prioritize syntax over unconstrained planning.

Proposed Method: Dual-Phase Cascaded GCD

As formalized in Section 3, the model generates an unconstrained CoT reasoning block and an optimistic draft. If the draft fails, the system executes a GCD fallback guided by the preserved reasoning. Critically, we never mention MiniZinc in the prompts used for our approach, using the name "MaskedLanguage" instead. This ensures that our method is using only the knowledge provided in the prompts instead of retrieving any previous pretrained knowledge. All the previous baselines do mention MiniZinc explicitly, putting our method purposely at disadvantage initially to get a tight estimate of the method's performance.

4.3. Evaluation Metrics: pass@k and Automated Semantic Judging

Because LLM generation is inherently probabilistic, evaluating a single greedy output often misrepresents a model's true capability. We evaluate all methods using the standard *pass@k* metric with $k = 1, 3, 5$, which measures the probability that at least one out of k generated candidate programs successfully solves the task. To induce the variance required for diverse sampling, we utilized multinomial sampling with a temperature of $T = 0.6$ across all generation calls.

For a generated candidate to be classified as a "Pass", it must successfully navigate three strict, sequential evaluation gates:

1. **Syntactic Gate:** The generated string must be successfully parsed by the formal Lark CFG parser, verifying absolute structural compliance.
2. **Semantic Compiler Gate:** The code is passed to the native MiniZinc compiler using the `-model-check-only` flag. This deterministic check instantly rejects uninitialized variables, out-of-bounds array accesses, and semantic type mismatches.
3. **Functional Intent Gate (LLM-as-a-Judge):** Because code can compile perfectly but fail to solve the requested problem, we implement a strict semantic judge. We employ a larger, independent local model (`Qwen3.5`) via Ollama and prompted with a highly strict CoT evaluation rubric. The judge compares the generated code against the golden solution from the benchmark, analyzing variable mapping, constraint logic, and optimization directions. The judge returns a normalized score between 0.0 and 1.0; a candidate is only marked as successful if it achieves a score of ≥ 0.85.

4.4. Results and Discussion

The experimental results for the MiniZinc code generation benchmark are summarized in Table 1. The evaluation rigorously assesses syntactic correctness, compilation viability, and semantic alignment through the *pass*@1, *pass*@3, and *pass*@5 metrics. The data clearly demonstrate the superiority of the proposed Dual-Phase architecture, while also revealing profound insights into the interaction between modern reasoning LLMs and strict symbolic constraints.

Table 1. Evaluation of code generation strategies on the MiniZinc benchmark. Results denote the *pass@k* accuracy (%), defined as syntactic, semantic, and compiler-verified success. The Δ columns represent the relative improvement of the proposed Dual-Phase architecture over the respective baseline at each k.

Method	**pass@1**	Δ	**pass@3**	Δ	**pass@5**	Δ
Zero-Shot	8.0	+687.5%	18.0	+283.3%	21.0	+261.9%
One-Shot (No GCD)	47.0	+34.0%	62.0	+11.3%	68.0	+11.8%
One-Shot (GCD Only)	8.0	+687.5%	8.0	+762.5%	9.0	+744.4%
Dual-Phase (Proposed)	63.0	-	69.0	-	76.0	-

 doi:10.20944/preprints202606.1011.v1

4.4.1. The Semantic Collapse of Strict GCD

Perhaps the most striking finding from the benchmark is the catastrophic performance degradation of the **One-Shot (GCD Only)** baseline. Despite being provided with the exact same in-context example and grammar description as the unconstrained baseline, the GCD-only approach achieved a mere 8.0% *pass*@1 and plateaued at 9.0% for *pass*@5.

This starkly validates our core hypothesis regarding the "Semantic-Syntax Trade-off." Because the GCD-only baseline physically prevents the model from outputting its native `<think>` reasoning block—forcing it to immediately output constrained EBNF tokens—the model is stripped of its ability to perform Chain-of-Thought (CoT) planning. Consequently, the FSM logits processor forces the model to make greedy syntactic choices without a global semantic plan. The resulting code is 100% syntactically perfect but semantically nonsensical, failing the compiler's type-checks or the semantic judge's intent verification. This demonstrates that for modern, reasoning-optimized models (e.g., DeepSeek-R1, Qwen), applying structural constraints at the expense of CoT reasoning is highly detrimental.

4.4.2. Unconstrained Reasoning vs. Zero-Shot Pre-training

The **Zero-Shot** baseline exhibited similarly poor performance (8.0% *pass*@1, 21.0% *pass*@5), confirming that the 1.5B parameter model lacks the pre-trained density to natively map natural language to the niche MiniZinc syntax.

Conversely, the **One-Shot (No GCD)** approach yielded a massive performance leap, achieving 47.0% at *pass*@1 and reaching 68.0% at *pass*@5. By providing a single example and, most importantly, allowing the model to utilize its `<think>` reasoning block, the LLM successfully planned the semantic bounds, variables, and constraints. The failures in this baseline were predominantly caused by minor syntactic hallucinations (e.g., mixing Python-style operators with MiniZinc operators), where the unconstrained generation drifted slightly out of bounds.

4.4.3. The Efficacy of the Dual-Phase Architecture

The proposed **Dual-Phase** framework significantly outperformed all baselines across every metric. By isolating semantic planning (Phase 1) from syntactic enforcement (Phase 2), the system achieved a *pass*@1 accuracy of 63.0% and a maximum *pass*@5 accuracy of 76.0%.

The relative improvements (Δ) highlight the exact value of the cascaded approach. Compared to the strongest baseline (One-Shot No GCD), the Dual-Phase architecture achieved a remarkable **+34.0% relative improvement in** *pass*@1 **accuracy**. This indicates that for single-sample generation—where computational efficiency is paramount—the optimistic bypassing and subsequent GCD fallback effectively "rescued" a third of the generations that would have otherwise failed due to trivial syntax errors. Because the fallback mechanism is explicitly guided by the preserved `<think>` reasoning from Phase 1, the FSM logits mask acts as a precise syntax repair engine rather than an adversarial constraint.

4.4.4. Scaling with Sampling Diversity (*pass*@*k*)

As the sampling budget increases from $k = 1$ to $k = 5$, the performance of the unconstrained One-Shot approach and the Dual-Phase approach scales logarithmically. The Dual-Phase framework consistently defines the upper bound, maintaining an $+11.8\%$ relative improvement over the unconstrained baseline at *pass*@5. The narrowing of the relative gap at higher k values is expected; with enough stochastic sampling temperature, unconstrained models will eventually "guess" the correct syntax. However, the Dual-Phase architecture dramatically accelerates this convergence, proving that strict neurosymbolic grounding yields far higher reliability per computational cycle.

In the context of this existing literature, our proposed Dual-Phase Cascaded architecture addresses a critical intersection of these open challenges. While prior works rely on computationally expensive fine-tuning or token-level search to enforce syntax and semantics [12,24], we aim to provide a lightweight, training-free alternative. Specifically, our approach directly addresses the semantic degradation induced by standard GCD [21]. By isolating unconstrained semantic planning (Chain-of-

 doi:10.20944/preprints202606.1011.v1

Thought) from syntactic enforcement (the FSM logits mask), our framework prevents the grammar constraint from distorting the LLM's natural generation distribution during the critical reasoning phase. Furthermore, by introducing an optimistic fast-path verified by native compilers, our method guarantees syntactic validity while actively prioritizing semantic alignment, bridging the gap between rigid formal constraints and the fluid reasoning capabilities of modern foundation models.

5. Conclusions and Future Work

In this paper, we addressed the enduring challenge of generating reliable, syntactically, and semantically correct code for niche Domain-Specific Languages (DSLs) using Large Language Models. We identified a critical friction point in modern reasoning-optimized LLMs: while Grammar-Constrained Decoding (GCD) successfully eliminates syntactic hallucinations, blindly applying strict constraints disrupts the model's native Chain-of-Thought (CoT) planning, frequently trapping the generation in irreversible semantic dead-ends.

To resolve this "Semantic-Syntax Trade-off," we introduced a Dual-Phase Cascaded Neurosymbolic Framework. By injecting formal EBNF rules directly into the context window, we grounded the model symbolically. Phase 1 permits the model to reason unconstrained and output an optimistic draft; if this draft passes native compiler checks, the generation concludes efficiently. If the draft fails, Phase 2 preserves the successful unconstrained reasoning and utilizes it to guide a strict GCD fallback. Our empirical evaluation on a newly constructed benchmark of 100 MiniZinc constraint programming tasks demonstrated that this "think-then-constrain" approach significantly outperforms zero-shot, unconstrained one-shot, and pure GCD baselines. The dual-phase architecture effectively re-purposes the formal grammar from an adversarial restriction into a localized, reasoning-guided syntax repair engine, achieving highly reliable code generation without the computational burden of model fine-tuning.

5.1. Limitations and Threats to Validity

Despite the promising results, this study presents certain limitations. Regarding *internal validity*, our empirical evaluation relies on a specific class of reasoning models (e.g., Qwen2.5-Instruct). While this demonstrates the architecture's viability on consumer-grade hardware, the interaction between GCD and larger, proprietary frontier models (e.g., OpenAI o1) remains unexplored. Furthermore, while our LLM-as-a-judge implements a strict evaluation rubric grounded by parsed ASTs, it remains a heuristic proxy; it is not a complete substitute for formal functional verification against mathematical unit tests.

Regarding *external validity*, our experiments were constrained to MiniZinc. While MiniZinc is highly representative of complex, declarative DSLs, the scalability of the optimistic bypassing mechanism to highly imperative DSLs or multi-file domain environments requires further investigation. Finally, the Phase 2 GCD implementation relies on the *Outlines* finite-state machine compilation, which imposes an initial processing overhead and restricts the expressiveness of the CFG to strictly regular or right-recursive structures to avoid memory exhaustion.

5.2. Future Research Directions

The insights derived from this cascaded architecture open several promising avenues for future research. A primary direction involves integrating search-based reasoning back into the generation pipeline. While pure token-level Monte Carlo Tree Search (MCTS) proved too computationally expensive and disjointed for initial generation in our early experiments, *Localized MCTS* holds immense potential as an advanced repair mechanism. Future work will explore transitioning from a greedy GCD fallback to a localized MCTS that explores alternative syntax derivations guided by compiler-error feedback.

Additionally, we plan to extend this framework to a multi-agent paradigm, where one reasoning agent generates the semantic CoT, a symbolic parser identifies specific structural failure indices, and an MCTS agent systematically searches for valid localized repairs. Finally, expanding the benchmark

 doi:10.20944/preprints202606.1011.v1

to encompass a wider variety of industrial DSLs—such as hardware description languages and proprietary configuration formats—will further validate the generalizability of in-context symbolic grounding coupled with cascaded constraints.

Author Contributions: Conceptualization, R.R. and D.D. and S.P. and H.B.; methodology, R.R. and D.D. and S.P. and H.B.; validation, R.R..; investigation, R.R. and D.D. and S.P. and H.B.; data curation, R.R.; writing—original draft preparation, R.R.; writing—review and editing, R.R. and D.D. and S.P. and H.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The benchmark dataset and code underlying this article is available in the following public repository https://github.com/IMC-UAS-Krems/mcts-niche-dsl.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv.org* **2021**.
2. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code Llama: Open Foundation Models for Code. *arXiv.org* **2023**.
3. Nethercote, N.; Stuckey, P.J.; Becket, R.; Brand, S.; Duck, G.J.; Tack, G. MiniZinc: Towards a Standard CP Modelling Language. In Proceedings of the Principles and Practice of Constraint Programming – CP 2007; Bessière, C., Ed., Berlin, Heidelberg, 2007; pp. 529–543. https://doi.org/10.1007/978-3-540-74970-7_38.
4. Poesia, G.; Polozov, A.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; Gulwani, S. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In Proceedings of the International Conference on Learning Representations, 2022.
5. Willard, B.T.; Louf, R. Efficient Guided Generation for Large Language Models. *arXiv.org* **2023**.
6. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; brian ichter.; Xia, F.; Chi, E.H.; Le, Q.V.; Zhou, D. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In Proceedings of the Advances in Neural Information Processing Systems; Oh, A.H.; Agarwal, A.; Belgrave, D.; Cho, K., Eds., 2022.
7. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.L.; Cao, Y.; Narasimhan, K.R. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In Proceedings of the Thirty-seventh Conference on Neural Information Processing Systems, 2023.
8. Xue, T.; Li, X.; Azim, T.; Smirnov, R.; Yu, J.; Sadrieh, A.; Pahlavan, B. Multi-Programming Language Ensemble for Code Generation in Large Language Model. *ArXiv* **2024**, *abs/2409.04114*.
9. Sarker, L.; Downing, M.; Desai, A.; Bultan, T. Assessing, Exploiting, and Mitigating Syntactic Robustness Failures in LLM-Based Code Generation, 2026. arXiv:2404.01535 [cs.SE], https://doi.org/10.1145/3793655.3793727.
10. Liang, Q.; Zhang, Z.; Sun, Z.; Lin, Z.; Luo, Q.; Xiao, Y.; Chen, Y.; Zhang, Y.; Zhang, H.; Zhang, L.; et al. Grammar-Based Code Representation: Is It a Worthy Pursuit for LLMs? In Proceedings of the Findings of the Association for Computational Linguistics: ACL 2025; Che, W.; Nabende, J.; Shutova, E.; Pilehvar, M.T., Eds., Vienna, Austria, 2025; pp. 15640–15653. https://doi.org/10.18653/v1/2025.findings-acl.807.
11. Thakur, S.; Ahmad, B.; Fan, Z.; Pearce, H.; Tan, B.; Karri, R.; Dolan-Gavitt, B.; Garg, S. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023, pp. 1–6. ISSN: 1558-1101, https://doi.org/10.23919/DATE56975.2023.10137086.
12. Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan-Gavitt, B.; Karri, R.; Garg, S. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Transactions on Design Automation of Electronic Systems* **2024**, *29*, 46:1–46:31. https://doi.org/10.1145/3643681.
13. Bassamzadeh, N.; Methani, C. A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation, 2024. arXiv:2407.02742 [cs.SE], https://doi.org/10.48550/arXiv.2407.02742.

 doi:10.20944/preprints202606.1011.v1

14. Fu, D.J.; Gupta, A.; Councilman, A.; Grove, D.; Wang, Y.X.; Adve, V. SLMFix: Leveraging Small Language Models for Error Fixing with Reinforcement Learning, 2025. arXiv:2511.19422 [cs.SE], https://doi.org/10.48550/arXiv.2511.19422.
15. Delgado, D.; Burgueño, L.; Clarisó, R. A framework for assessing the capabilities of code generation of constraint domain-specific languages with large language models. *Journal of Systems and Software* **2026**, *238*, 112871. https://doi.org/10.1016/j.jss.2026.112871.
16. Shen, D.; Chen, X.; Wang, C.; Sen, K.; Song, D. Benchmarking Language Models for Code Syntax Understanding. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2022.
17. Mora, F.; Wong, J.; Lepe, H.; Bhatia, S.; Elmaaroufi, K.; Varghese, G.; Gonzalez, J.; Polgreen, E.; Seshia, S.A. Synthetic Programming Elicitation for Text-to-Code in Very Low-Resource Programming and Formal Languages. *Advances in Neural Information Processing Systems 37* **2024**.
18. Gao, M.; Zhao, J.; Lin, Z.; Ding, W.; Hou, X.; Feng, Y.; Li, C.; Guo, M. AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs. *2024 IEEE 42nd International Conference on Computer Design (ICCD)* **2024**, pp. 162–169.
19. Lu, Y.; Liu, S.; Zhang, Q.; Xie, Z. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)* **2023**, pp. 722–727.
20. Geng, S.; Josifoski, M.; Peyrard, M.; West, R. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In Proceedings of the Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing; Bouamor, H.; Pino, J.; Bali, K., Eds., Singapore, 2023; pp. 10932–10952. https://doi.org/10.18653/v1/2023.emnlp-main.674.
21. Park, K.; Wang, J.; Berg-Kirkpatrick, T.; Polikarpova, N.; D' Antoni, L. Grammar-Aligned Decoding. In Proceedings of the Advances in Neural Information Processing Systems; Globerson, A.; Mackey, L.; Belgrave, D.; Fan, A.; Paquet, U.; Tomczak, J.; Zhang, C., Eds. Curran Associates, Inc., 2024, Vol. 37, pp. 24547–24568. https://doi.org/10.52202/079017-0774.
22. Wen, H.; Zhu, Y.; Liu, C.; Ren, X.; Du, W.; Yan, M. Fixing Function-Level Code Generation Errors for Foundation Large Language Models **2025**. arXiv:2409.00676 [cs.SE], https://doi.org/10.48550/arXiv.2409.00676.
23. Liu, C.; Bao, X.; Zhang, H.; Zhang, N.; Hu, H.; Zhang, X.; Yan, M. Guiding ChatGPT for Better Code Generation: An Empirical Study. In Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, pp. 102–113. ISSN: 2640-7574, https://doi.org/10.1109/SANER60148.2024.00018.
24. Albinhassan, M.; Madhyastha, P.; Russo, A. $\texttt{SEM-CTRL}$: Semantically Controlled Decoding. *Transactions on Machine Learning Research* **2025**.