Proceedings of the ASME 2026
International Design Engineering Technical Conferences &
Computers and Information in Engineering Conference
IDETC-CIE 2026
August 23–26, 2026, Houston, TX, USA

DETC2026-XXXX

RELIABLE NATURAL-LANGUAGE TO SYSMLV2 TRANSLATION VIA COMPILER-DRIVEN ITERATIVE REFINEMENT

Chance LaVoie[1],[*], Eladio Andujar Lugo[1], Levent Burak Kara[1]

[1]Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA

ABSTRACT

SysMLv2 is a textual, compilable modeling language, so generated output is only useful if it passes deterministic compiler checks. One-shot large language model generation frequently produces plausible but uncompilable SysMLv2, motivating compiler-gated refinement rather than one-pass synthesis.

This paper studies a syntax-only compiler-in-the-loop pipeline on the full SysMBench prompt set (IDs 1–151), evaluated across four model configurations. For each prompt/model trial, we compare baseline single-shot output (iteration 1 only) against the final output after iterative generate–compile–repair using `syside check` *as the oracle. Across 604 trials, first-shot compilation succeeds on 51.16% (309/604; 95% Wilson CI: 47.18–55.13%), while the compiler-in-the-loop final output reaches 100.00% eventual compilation (604/604; 95% Wilson CI: 99.37–100.00%), a gain of 48.84 percentage points with zero unresolved prompts. Failed-first-shot cases recover fully (295/295). These results show that deterministic compiler feedback is the primary control signal for reliable natural-language–to–SysMLv2 generation and that single-shot generation leaves a large, recoverable syntactic error surface.*

Keywords: SysMLv2, model-based systems engineering, large language models, compiler-in-the-loop, natural language to model translation, syntactic validity

1. INTRODUCTION

Model-Based Systems Engineering (MBSE) relies on formal, machine-checkable artifacts to support early design iteration, requirements traceability, and verification. The release of Systems Modeling Language version 2 (SysMLv2) [1] strengthens this workflow... strengthens this workflow by introducing a fully textual modeling language with a well-defined grammar and an executable toolchain, enabling models to be parsed, compiled, and version-controlled like software.

[*]Corresponding author: chancel@cmu.edu
Documentation for `asmeconf.cls`: Version 1.45, February 20, 2026.

This shift creates an opportunity for automated model synthesis: engineers could describe a system in natural language and obtain a SysMLv2 model that can be compiled and analyzed. Large language models (LLMs) are a natural candidate for this translation task, given their success at generating structured code from natural-language specifications. However, a model that is "almost correct" is not useful in practice: even small syntax or metamodel errors prevent compilation and therefore block rendering, visualization, and downstream analysis. Reliable translation therefore requires *syntactic validity*, not just plausible-looking output.

The current status quo is that LLMs can often produce readable pseudo-code or SysML-like text. Pseudo-code is not sufficient for MBSE: SysML is intended to serve as a *compilable* and *visualizable* system representation, and compilation is the gateway to rendering, visualization, and analysis inside engineering toolchains. If an LLM output is not syntactically correct, it cannot be compiled and therefore cannot be rendered into the diagrams and model views that engineers use to understand and communicate system structure and behavior. As a result, the practical value of AI-assisted modeling hinges on producing syntactically valid SysMLv2, not informal approximations.

In many code-generation domains, syntactic reliability can be improved by training or fine-tuning on large corpora of valid programs. For SysMLv2, such data is currently scarce: there is no large, broadly available dataset of compiler-verified models, and the language itself is evolving. Consequently, one-shot LLM generations often mix SysML v1 patterns with SysMLv2 constructs or violate the SysMLv2 grammar, leading to outputs that read convincingly but fail deterministic checks. This data scarcity motivates a method that improves correctness *without* modifying model weights.

This paper presents a compiler-in-the-loop framework that couples an off-the-shelf LLM with a deterministic verifier in an iterative generate–compile–repair loop. At each iteration, the model proposes a complete SysMLv2 candidate; the verifier returns precise diagnostics (error types and locations); and a controller converts these diagnostics into targeted revision instructions. The loop terminates only when zero errors are reported, yielding syntactic validity by construction without training, fine-tuning, or specialized hardware.

We evaluate the approach on all 151 SysMBench prompts and compare two conditions for each prompt/model pair: single-shot compilation at iteration 1 versus final compilation after compiler-in-the-loop refinement. Results show that deterministic compiler feedback is the primary driver of dependable generation: iterative refinement converts many first-pass failures into compilable SysMLv2 while single-shot generation alone leaves a large recoverable error surface.

2. RELATED WORK

Our review focuses on prior efforts in automated SysML v2 generation, grammar-constrained structured synthesis, and compiler-guided iterative refinement, with an emphasis on approaches addressing reliable syntactic correctness in low-data modeling languages.

2.1. LLM-Based SysML v2 Model Generation

Recent work has explored using large language models to generate SysML v2 models from natural language. Bouamra et al. [2] propose SysTemp, a multi-agent framework that decomposes the generation task into requirement extraction, template-based skeleton construction, and grammar-level parsing feedback. Their system employs a ParserAgent to detect syntactic errors and iteratively repair generated models, achieving 80% convergence across five evaluation scenarios. This work demonstrates that structural scaffolding and agent decomposition improve grammar conformity in sparse-data settings. However, syntactic correctness is defined at the grammar level and convergence is reported empirically rather than enforced as a termination invariant.

Cibrián et al. [3] introduce an agent-based framework combining retrieval-augmented generation (RAG) with ANTLR-based grammar validation for SysML v2 synthesis. Their approach reports 100% syntactic validity across 20 curated prompts under grammar-level parsing. While this represents a significant advance in structured generation reliability, correctness is enforced through grammar parsing rather than full compiler validation. Because grammar-level validation ensures context-free structural conformity but does not enforce full static semantics under the official toolchain, grammar-valid models may still fail compilation in production environments. neccessary but not sufficient Together, these efforts establish that iterative validation substantially improves syntactic success in SysML v2 generation. However, prior systems define correctness at the grammar level and evaluate on limited scenario sets. Our work builds on these insights while shifting the correctness oracle from grammar parsing to the official SysIDE compiler and scaling evaluation to benchmark-level scenarios.

2.2. Grammar-Constrained and Template-Based Structured Synthesis

Structured synthesis approaches aim to reduce hallucinations in LLM output by constraining generation via templates or grammar rules. SysTemp [2] explicitly uses a template generator based on Jinja2 to construct syntactically compliant model skeletons prior to completion. Grammar-constrained decoding and post-generation parsing similarly reduce token-level structural invalidity.

While grammar validation ensures adherence to context-free rules, it does not enforce type resolution, cross-reference integrity, constraint satisfaction, or toolchain compatibility. Thus, grammar-valid artifacts may remain unusable within industrial MBSE workflows. Our approach differs by treating grammar conformity as necessary but insufficient and requiring full compiler acceptance prior to termination.

2.3. Compiler-in-the-Loop and Verifier-Guided Generation

Prior work in neural code generation has explored leveraging compiler diagnostics to improve the compilability of model outputs. For example, Wang et al. [4] propose a multi-stage refinement framework that uses compiler feedback to iteratively revise generated programs and increase compilation success rates. Such approaches demonstrate that deterministic compiler signals can serve as effective supervisory feedback in programming-language settings. However, these systems operate in mature programming ecosystems and treat compilation success as an empirical objective rather than as a structural termination invariant.

Beyond compiler-feedback approaches in code generation, iterative generation guided by deterministic verifiers has also been studied under counterexample-guided inductive synthesis and execution-based refinement paradigms. Grubišić et al. [5] demonstrate that LLVM compiler feedback can serve as a supervisory signal for refining LLM-generated intermediate representation. Their work shows that deterministic compilation signals constrain output space and improve structural validity in programming languages.

While these approaches establish the feasibility of compiler-aware refinement in software domains, they differ in both objective and context from model-based systems engineering. Compiler feedback in such systems is typically used to improve optimization quality or increase compilation probability, rather than to enforce strict termination conditions. Moreover, these methods operate in programming languages with extensive training corpora and stable ecosystems.

In contrast, our work applies compiler-in-the-loop refinement to SysMLv2, a newly standardized modeling language with sparse representation in LLM training data and strict toolchain requirements. Rather than treating compilation results as heuristic feedback, we enforce zero-error compilation under the SysMLv2 compiler (invoked via `syside check`) as a termination invariant. This elevates syntactic correctness from an empirical metric to a property guaranteed by construction, aligning generation reliability with production MBSE toolchain criteria rather than grammar-level approximations.

3. METHODOLOGY

3.1. Evaluation Scope and Dataset

This campaign evaluates only *syntactic correctness* under deterministic compiler checking. We use SysMBench [6] prompt IDs 1–151 and analyze four provider/model runs available in this repository: OpenAI Codex 5.2 (`gpt-5.2-codex`) [7], Anthropic Sonnet 4.6 (`claude-sonnet-4-6`) [8], DeepSeek Reasoner (`deepseek-reasoner`) [9], and Mistral Large (`mistral-large-latest`) [10]. This yields 604 prompt-level trials (151 prompts × 4 models).

All claims in this section and the Results section are limited to compiler-level compilability. We do not report or claim semantic adequacy, behavioral fidelity, or design quality.

3.2. Baseline and Pipeline Definitions

For each prompt/model trial we define two outcomes from the same generate–compile–repair trajectory:

1. **Baseline (single-shot):** compilation status at iteration 1 only.
2. **Pipeline (compiler-in-the-loop):** compilation status of the final iteration after iterative repair.

The iterative loop is compiler-gated: the model proposes SysMLv2 text, the SysIDE checker (`syside check`) returns diagnostics, and the next iteration is conditioned on those diagnostics until success or stopping cap.

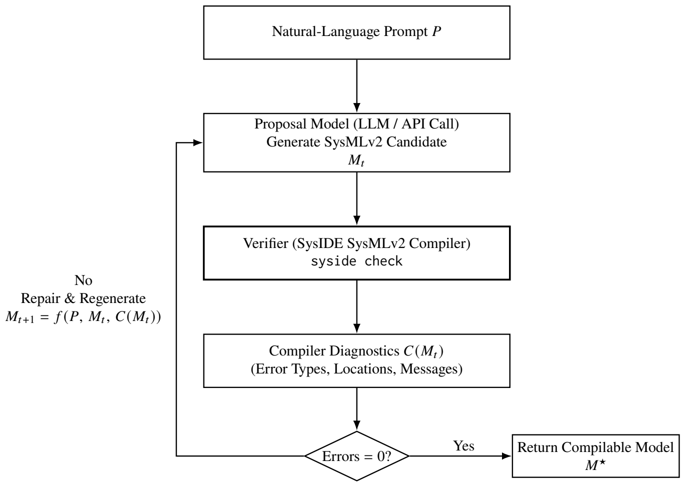
Natural-Language Prompt P
Proposal Model (LLM / API Call)
Generate SysMLv2 Candidate
M_t
Verifier (SysIDE SysMLv2 Compiler)
syside check
Compiler Diagnostics $C(M_t)$
(Error Types, Locations, Messages)
Errors = 0?
Yes
Return Compilable Model
$M^\star$
No
Repair & Regenerate
$M_{t+1} = f(P, M_t, C(M_t))$

FIGURE 1: Compiler-in-the-loop generate–compile–repair loop used for all prompt/model trials.

3.3. Compiler Oracle and Stopping Rule

The verifier is the SysIDE SysMLv2 compiler/checker [11]. A prompt is marked successful only when a run reaches zero compiler errors (successful compile signal in run logs/manifests). For each prompt we use the persisted run selected by its `<id>_refine_manifest.json`; associated per-iteration logs are read from the referenced `run_log.json`.

3.4. Extracted Metrics

We extract deterministic prompt-level and iteration-level metrics from raw artifacts.

Prompt-level metrics include: `first_shot_pass`, `eventual_pass`, `iterations_run`, `iterations_to_success`, `unresolved_within_cap`, first/final error counts, cumulative error volume, recovery indicator (failed first shot then passed), runtime, and token totals when available.

Iteration-level metrics include: `pass_at_iteration`, `error_count`, `error_families_json`, iteration runtime, and iteration token usage when available.

Compiler error families are parsed directly from diagnostics using the pattern `error (<family>):`. Warning families are tracked separately and do not affect success/failure labels.

3.5. Statistical Analysis

Let n be prompt count in a group (model-specific or pooled). We report:

1. First-shot pass rate, first-shot fail rate, eventual pass rate, unresolved rate.
2. Absolute gain (percentage-point improvement): eventual pass minus first-shot pass.
3. Relative gain: $(p_{\text{final}} - p_{\text{first}})/p_{\text{first}}$.
4. Iterations-to-success distribution (mean, median, standard deviation, quantiles, maximum).
5. Wilson 95% confidence intervals for key proportions.
6. Bootstrap 95% confidence interval for mean iterations-to-success (fixed seed 20260220; 10,000 resamples).
7. Paired baseline-vs-pipeline significance via McNemar test.

All statistics are deterministic given the same artifacts and random seed.

3.6. Reproducibility Pipeline

Reproducible analytics are implemented in:

- `paper/results/scripts/extract_syntax_metrics.py`
- `paper/results/scripts/compute_syntax_stats.py`
- `paper/results/scripts/make_syntax_tables.py`
- `paper/results/scripts/make_syntax_figures.py`
- `paper/results/scripts/run_syntax_campaign.sh`

A single command,

```
bash paper/results/scripts/run_syntax_campaign.sh
```

regenerates all CSV outputs, tables, and figures, then performs a hash-stability determinism check on key statistics files.

4. RESULTS

4.1. Overall Compiler-Gated Syntactic Outcomes

Across all 604 prompt-level trials (151 prompts for each of four models), first-shot compilation succeeded for 309/604 cases (51.16%; 95% Wilson CI: 47.18–55.13%). Under compiler-in-the-loop refinement, eventual compilation succeeded for 604/604 cases (100.00%; 95% Wilson CI: 99.37–100.00%). The absolute gain from baseline single-shot to final pipeline output was 48.84 percentage points (relative gain 95.47%), with zero unresolved prompts.

Copyright ©2026 by ASME

A paired baseline-vs-pipeline test showed a strong shift toward success (McNemar: $b = 0$, $c = 295$, $p = 1.10 \times 10^{-65}$), indicating that improvements were driven by first-shot failures that were recovered by iterative repair.

TABLE 1: Overall compiler-gated outcomes for single-shot baseline vs iterative pipeline.

Metric	Value
Prompts	604
First-shot pass	309 (51.16%)
First-shot fail	295 (48.84%)
Eventual pass (pipeline)	604 (100.00%)
Absolute gain	48.84%
Relative gain over first-shot	95.47%
Unresolved	0 (0.00%)
Recovered after first-shot failure	295
First-shot pass 95% Wilson CI	[47.18%, 55.13%]
Eventual pass 95% Wilson CI	[99.37%, 100.00%]
Mean iterations-to-success	1.733
Mean iterations 95% bootstrap CI	[1.654, 1.816]

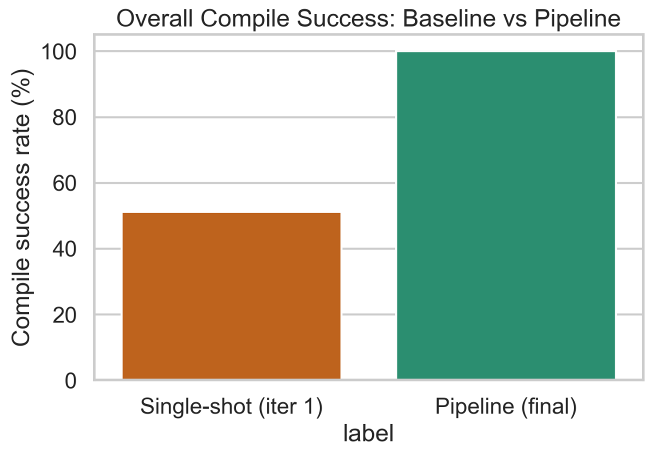

FIGURE 2: Overall compile success for single-shot baseline (iteration 1) vs final compiler-in-the-loop output.

4.2. Per-Model Syntactic Reliability

All models reached 151/151 eventual compilation under the compiler-gated loop, but first-shot pass rates differed substantially. Anthropic Sonnet 4.6 had the highest first-shot pass rate (82.78%), while OpenAI (41.72%), DeepSeek Reasoner (41.06%), and Mistral Large (39.07%) showed similar first-shot behavior and correspondingly larger iterative gains.

TABLE 2: Per-model syntactic reliability summary.

Provider	Model	N	First-shot pass	Eventual pass	Gain (pp)	Unresolved	Mean iters	Median iters	M
anthropic	claude-sonnet-4-6	151	82.78%	100.00%	17.22%	0.00%	1.252	1.000	11
deepseek_reasoner	deepseek-reasoner	151	41.06%	100.00%	58.94%	0.00%	1.974	2.000	8.0
mistral_large	mistral-large-latest	151	39.07%	100.00%	60.93%	0.00%	1.980	2.000	5.0
openai	openai	151	41.72%	100.00%	58.28%	0.00%	1.728	2.000	5.0

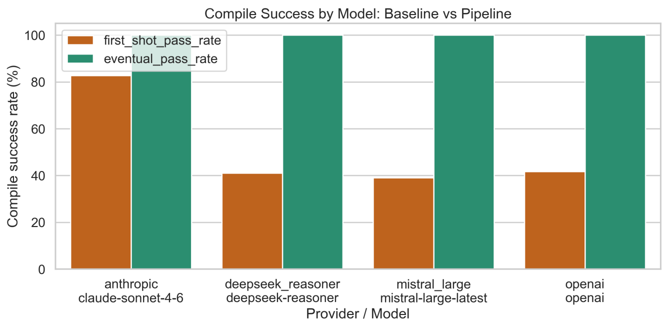

FIGURE 3: Single-shot vs final compile success broken out by provider/model.

4.3. Iteration and Recovery Dynamics

Iterations-to-success over all successful runs had mean 1.733, median 1, and maximum 11 (bootstrap 95% CI for mean: 1.654–1.816). The distribution was concentrated in early iterations: 309 cases succeeded at iteration 1, 201 at iteration 2, and 60 at iteration 3; only two cases required more than five iterations.

Among first-shot failures, recovery was complete: 295/295 (100%) failed-first-shot cases eventually compiled. For this recovery subset, mean iterations-to-success was 2.502 (median 2, max 11).

TABLE 3: Distribution of iterations required to reach first successful compile.

Group	Iteration	Count	Share
ALL/ALL	1.000	309	51.16%
ALL/ALL	2.000	201	33.28%
ALL/ALL	3.000	60	9.93%
ALL/ALL	4.000	23	3.81%
ALL/ALL	5.000	9	1.49%
ALL/ALL	8.000	1	0.17%
ALL/ALL	11.000	1	0.17%
anthropic/claude-sonnet-4-6	1.000	125	82.78%
anthropic/claude-sonnet-4-6	2.000	22	14.57%
anthropic/claude-sonnet-4-6	3.000	3	1.99%
anthropic/claude-sonnet-4-6	11.000	1	0.66%
deepseek_reasoner/deepseek-reasoner	1.000	62	41.06%
deepseek_reasoner/deepseek-reasoner	2.000	54	35.76%
deepseek_reasoner/deepseek-reasoner	3.000	19	12.58%
deepseek_reasoner/deepseek-reasoner	4.000	12	7.95%
deepseek_reasoner/deepseek-reasoner	5.000	3	1.99%
deepseek_reasoner/deepseek-reasoner	8.000	1	0.66%
mistral_large/mistral-large-latest	1.000	59	39.07%
mistral_large/mistral-large-latest	2.000	55	36.42%
mistral_large/mistral-large-latest	3.000	23	15.23%
mistral_large/mistral-large-latest	4.000	9	5.96%
mistral_large/mistral-large-latest	5.000	5	3.31%
openai/openai	1.000	63	41.72%
openai/openai	2.000	70	46.36%
openai/openai	3.000	15	9.93%
openai/openai	4.000	2	1.32%
openai/openai	5.000	1	0.66%

Copyright ©2026 by ASME

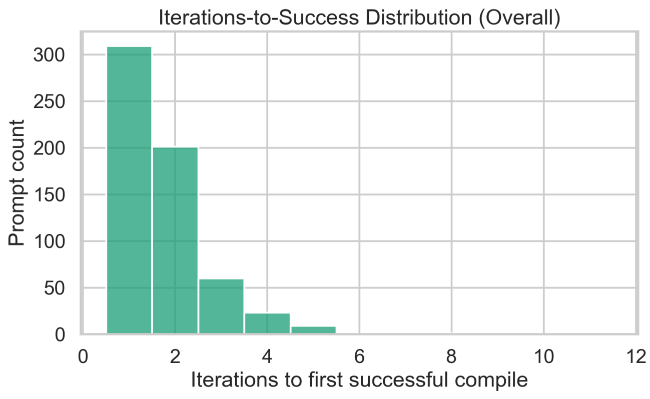

FIGURE 4: Histogram of iterations required to first successful compile.

TABLE 4: Top compiler error families on first iteration (pooled across models).

Error family	Count	Prompts affe
parsing-error	1142	234
reference-error	559	130
port-definition-owned-usages-not-composite	82	29
feature-chaining-feature-not-one	28	14
type-error	26	7
feature-reference-expression-referent-is-feature	18	8
invocation-expression-instantiated-type	17	9
feature-value-overriding	16	3
quantity-operator-expression	13	3
parameter-membership-owning-type	12	6
attribute-usage-features	7	1
cross-subsetting-crossing-feature	6	1
feature-crossing-specialization	6	2
connector-related-features	4	4
feature-owned-cross-subsetting	4	1
part-usage-part-definition	1	1
subsetting-uniqueness-conformance	1	1

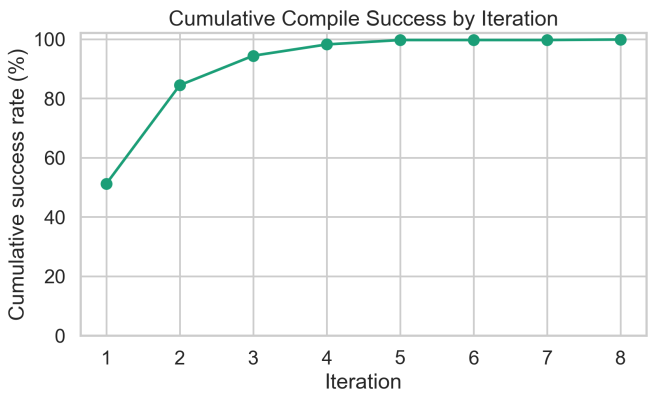

FIGURE 5: Cumulative success-by-iteration under compiler-in-the-loop refinement.

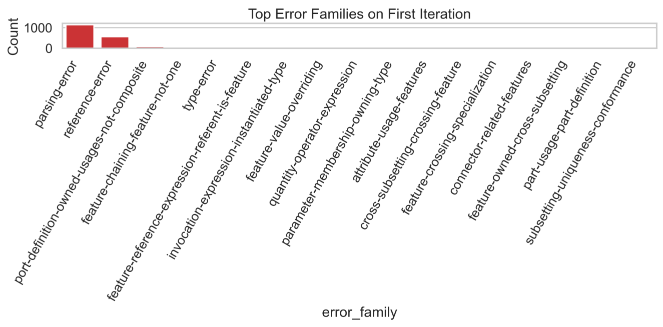

FIGURE 6: Pareto view of first-iteration compiler error families.

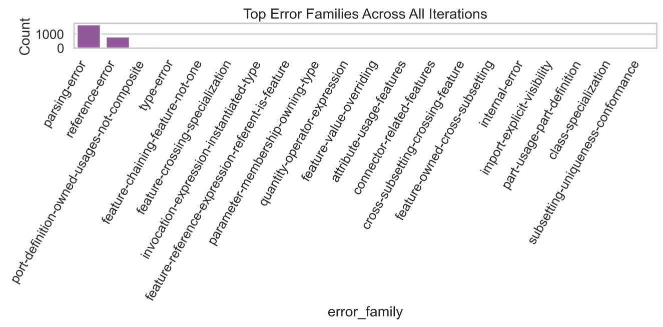

FIGURE 7: Pareto view of cumulative compiler error families across all iterations.

4.4. Compiler Error Taxonomy

At first iteration, the dominant error families were `parsing-error` (1142), `reference-error` (559), and `port-definition-owned-usages-not-composite` (82). Across all iterations, the same families remained dominant, with cumulative counts 1658, 794, and 97 respectively.

4.5. Prompt-Level Difficulty and Diagnostic Burden

Error burden was heterogeneous across prompts. The largest pooled cumulative error volumes were observed for prompt IDs 93 (133), 9 (88), 139 (81), and 144 (80), indicating a long-tail of syntactically difficult cases even under eventual convergence.

Copyright ©2026 by ASME

TABLE 5: High-burden prompts by cumulative compiler error volume.

Provider	Model	Prompt ID	First-iter errors	Total errors	Iterations run	Eventual pass
openai	openai	93	52	78	3	True
mistral_large	mistral-large-latest	9	12	67	5	True
mistral_large	mistral-large-latest	99	32	62	3	True
mistral_large	mistral-large-latest	74	18	58	4	True
openai	openai	117	11	54	5	True
deepseek_reasoner	deepseek-reasoner	93	13	51	4	True
deepseek_reasoner	deepseek-reasoner	88	39	49	3	True
mistral_large	mistral-large-latest	70	36	46	4	True
openai	openai	43	45	46	3	True
deepseek_reasoner	deepseek-reasoner	144	12	44	8	True
mistral_large	mistral-large-latest	139	29	39	3	True
deepseek_reasoner	deepseek-reasoner	89	14	38	5	True
mistral_large	mistral-large-latest	65	17	36	5	True
deepseek_reasoner	deepseek-reasoner	137	8	35	4	True
mistral_large	mistral-large-latest	148	4	[illegible]	[illegible]	[illegible]
openai	openai	111	31	[illegible]	[illegible]	[illegible]
mistral_large	mistral-large-latest	149	19	[illegible]	[illegible]	[illegible]
deepseek_reasoner	deepseek-reasoner	77	29	[illegible]	[illegible]	[illegible]
deepseek_reasoner	deepseek-reasoner	140	14	[illegible]	[illegible]	[illegible]
deepseek_reasoner	deepseek-reasoner	34	4	[illegible]	[illegible]	[illegible]
mistral_large	mistral-large-latest	51	3	[illegible]	[illegible]	[illegible]
mistral_large	mistral-large-latest	94	20	[illegible]	[illegible]	[illegible]
openai	openai	102	26	[illegible]	[illegible]	[illegible]
deepseek_reasoner	deepseek-reasoner	6	19	[illegible]	[illegible]	[illegible]
mistral_large	mistral-large-latest	121	17	24	4	True

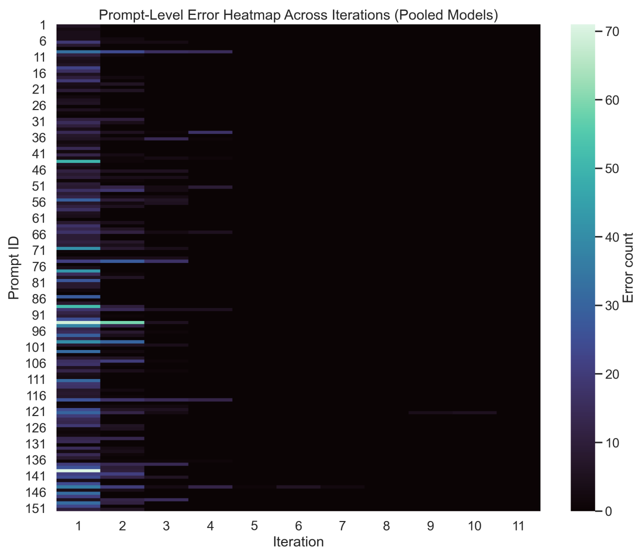

FIGURE 8: Prompt-by-iteration error heatmap across all models.

4.6. Runtime and Token Sensitivity (Secondary)

Runtime and token data, where available in artifacts, indicate substantial efficiency differences by model despite identical syntactic endpoints. Mean wall-time ranged from 12.97 s (Mistral Large) to 90.45 s (DeepSeek Reasoner), and mean total tokens ranged from 1795.24 to 5938.80 per prompt. One resumed-segment case (Anthropic prompt 121) is excluded from wall-time/token aggregates because only iterations 9–11 were persisted for that run segment.

These are secondary operational diagnostics and are not used as primary efficacy claims in this paper.

TABLE 6: Runtime, token, and estimated cost summary by model (cost is NA when unavailable).

Provider	Model	N	N runtime	N tokens
anthropic	claude-sonnet-4-6	151	150	150
deepseek_reasoner	deepseek-reasoner	151	151	151
mistral_large	mistral-large-latest	151	151	150
openai	openai	151	151	151

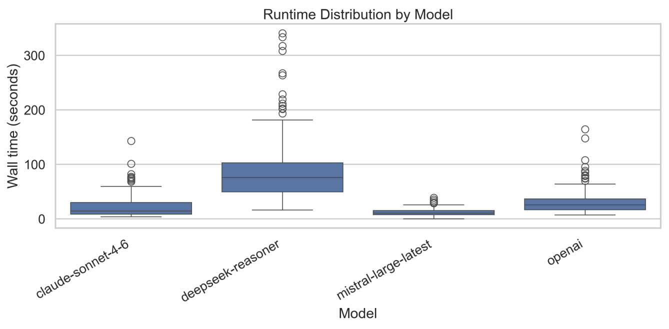

FIGURE 9: Runtime distribution by provider/model.

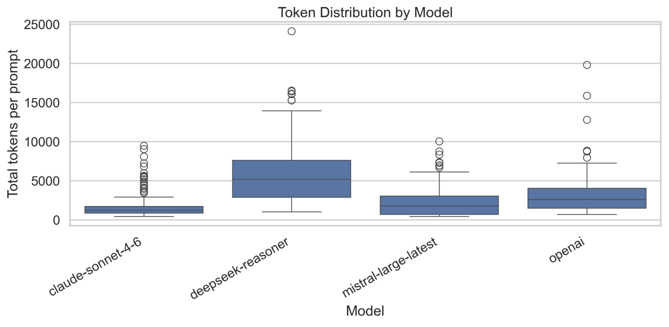

FIGURE 10: Token-use distribution by provider/model.

5. DISCUSSION

We discuss why deterministic compiler diagnostics provide an effective supervision signal for new, sparsely represented languages like SysMLv2, and the resulting trade-offs in runtime and number of iterations.

6. CONCLUSION

We present a reproducible pathway for reliable natural-language to SysMLv2 translation by embedding deterministic compiler feedback into LLM generation, thereby guaranteeing syntactically valid (compilable) output by construction.

ACKNOWLEDGMENTS

REFERENCES

[1] Group, Object Management. "OMG Systems Modeling Language (SysML) v2 Specification, Version 1.0 Beta." Technical report no. Object Management Group (OMG). 2024. URL https://www.omg.org/spec/SysML/2.0/. Official release draft, September 2024.

[2] Bouamra, A. et al. "SysTemp." arXiv:2506.21608 (2025). URL https://arxiv.org/abs/2506.21608.

[3] Cibrián, F. et al. "Agent-Based SysML v2 Synthesis with Retrieval-Augmented Generation and Grammar Validation." *Computers in Industry* (2025)DOI 10.1016/j.compind.2025.104350. URL https://doi.org/10.1016/j.compind.2025.104350.

[4] Wang, Xin, Chen, Wenhu, Chen, Xinyun and Wang, William Yang. "Compilable Neural Code Generation with Compiler Feedback." *Findings of the Association for Computational Linguistics: ACL 2022*: pp. 138–150. 2022. URL https://aclanthology.org/2022.findings-acl.2/.

[5] Grubišić, A. et al. "Compiler Feedback for Large Language Models." arXiv:2403.14714 (2024). URL https://arxiv.org/abs/2403.14714.

[6] Jin, Dongming, Jin, Zhi, Li, Linyu, Fang, Zheng, Li, Jia, Chen, Xiaohong and Luo, Yixing. "A System Model Generation Benchmark from Natural Language Requirements." *arXiv preprint arXiv:2508.03215* (2025).

[7] OpenAI. "GPT-5.2-Codex Model Documentation." https://developers.openai.com/api/docs/models/gpt-5.2-codex (2026). Model used in this campaign: `gpt-5.2-codex`. Accessed February 20, 2026.

[8] Anthropic. "Introducing Claude Sonnet 4.6." https://www.anthropic.com/news/claude-sonnet-4-6 (2026). API usage specifies `claude-sonnet-4-6`. Accessed February 20, 2026.

[9] DeepSeek. "Reasoning Model (`deepseek-reasoner`)." https://api-docs.deepseek.com/guides/reasoning_model (2026). Accessed February 20, 2026.

[10] Mistral AI. "Function Calling (Model Example: `mistral-large-latest`)." https://docs.mistral.ai/capabilities/function_calling/ (2026). Accessed February 20, 2026.

[11] Sensmetry. "SysIDE: The Open-Source IDE and Compiler for SysML v2." https://sensmetry.com/syside/ (2024). Accessed November 2025.

Copyright ©2026 by ASME