EVOSYSML: TRAINING A SPECIALIZED MODEL FOR SYSML V2 CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Automatically generating SysML v2 code from natural-language requirements can reduce the cost of model-based systems engineering, but progress is hindered by scarce requirement–code data and the limited SysML knowledge of general-purpose language models. We present EVOSYSML, a framework for training and deploying a specialized SysML generator, which consists of four stages. (1) An API teacher constructs cold-start data through breadth and depth evolution. (2) Local role models continue both processes, reconstruct aligned requirements, and filter candidates into a validated 50K training corpus. (3) Supervised fine-tuning and syntax-directed direct preference optimization train the final generator. (4) At deployment, formal validation selects an output from multiple sampled candidates. We also introduce EvoSysML-204, a manually verified bilingual benchmark containing 204 aligned requirements per language across three difficulty levels. Against 12 strong baselines, EVOSYSML achieves the highest average bilingual syntax pass rate using single-pass inference. Furthermore, compared to strong API models, our method can sample ten reasoning paths under an equivalent time budget, increasing syntax pass rates to 39.22% in English and 37.25% in Chinese. This outperforms the best-performing baselines in each language (Kimi-K2.7-Code and Claude-Sonnet-5) by 18.14 and 19.11 percentage points, respectively. Among syntax-valid outputs, EVOSYSML achieves conditional semantic averages of 86.90% and 77.34%, ranking first in English and tying for first in Chinese. Ablation studies confirm that data evolution, two-stage post-training, and validation-guided candidate selection all contribute to the final performance.

1 INTRODUCTION

Modern systems integrate increasingly complex software, hardware, physical processes, and human interactions. Model-Based Systems Engineering (MBSE) (Ramos et al., 2011) manages this complexity by representing requirements, structures, behaviors, interfaces, constraints, and verification relations in a consistent system model. As a core language for MBSE, the Systems Modeling Language (SysML) (Bajaj et al., 2022) supports traceable system specification throughout the engineering lifecycle. Currently, SysML v2 further introduces a standardized textual notation and formal semantics for machine-processable modeling, automated analysis, and tool integration (Hause et al., 2006; Bajaj et al., 2022). Nevertheless, translating natural-language requirements into syntactically valid and semantically complete SysML code remains labor-intensive and requires substantial modeling expertise (Dehn et al., 2025; Wang et al., 2025). Automating this process could therefore reduce modeling costs and make formal system models accessible earlier in system development.

Recent advances in code large language models (code LLMs) provide a promising foundation for requirement-to-code generation. Models such as CodeT5 (Wang et al., 2021), StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024) have demonstrated strong capabilities in code generation, completion, repair, and reasoning across mainstream programming languages. Their performance has been further improved through instruction tuning (Wang et al., 2023), synthetic task evolution (Luo et al., 2024), execution feedback (Wei et al., 2024), and model-based data filtering (Ahmad et al., 2025; Zhang et al., 2025). However, these advances largely rely on abundant repositories of compilable mainstream code and well-established execution environments. As a result, these methods cannot be directly applied to low-resource formal modeling languages such as SysML.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

SysML code generation remains constrained by two related limitations. **First, native SysML v2 requirement–code resources are scarce.** Current research has mainly focused on preliminary explorations of SysML v2 (DeHart, 2024), fine-tuning on 914 requirement–PlantUML pairs (An et al., 2026), and multi-agent generation and repair (Liu et al., 2026). However, these studies remain limited by small-scale data, non-native PlantUML representations, or reliance on inference-time repair. A large corpus of native SysML v2 requirement–code pairs with formal and semantic validation is still lacking. **Second, general-purpose LLMs exhibit insufficient adaptability to SysML.** Preliminary investigations reveal that current LLMs struggle to adhere to SysML-specific syntax and semantics, often producing plausible-looking but syntactically invalid or semantically incomplete code (detailed examples are provided in Section 2). These limitations motivate both scalable construction of validated training data and explicit domain adaptation for SysML code generation.

To address these limitations, we propose EvoSysML, a dedicated approach for building specialized models adapted to SysML code generation. This method is grounded in two key ideas: (a) **LLM-driven data synthesis via prompt engineering**: We propose using prompt engineering to systematically explore syntactic and semantic spaces, enabling LLMs to automatically synthesize code data that covers a broader range of topic domains and more complex structures. (b) **Targeted specialization via two-stage training and diverse sampling**: By constructing high-quality requirement-code pairs, we enhance the model's adaptability to SysML through a progressive two-stage training process, coupled with diversified sampling during deployment to ensure robust code generation.

To realize these ideas, EvoSysML is structured into four stages. Stage 1 uses DeepSeek-V4-Pro as an API teacher to construct cold-start data through breadth and depth evolution. Breadth evolution introduces new domains and requirement scenarios, whereas depth evolution increases program complexity. Stage 2 trains local role models to continue both processes, reconstruct aligned requirements, and filter candidates into a validated 50K training set. Stage 3 trains the final generator through supervised fine-tuning and syntax-directed direct preference optimization. Stage 4 deploys the model locally and uses formal validation to select an output from multiple sampled candidates.

To evaluate EvoSysML, we construct EvoSysML-204, a manually verified bilingual benchmark containing 204 aligned requirement–code pairs per language across three difficulty levels (see details in Section 4.1). On this benchmark, we compare our method against 12 strong general-purpose, code-specialized, and locally deployable baselines. Experimental results show that EvoSysML already achieves the highest average bilingual syntax pass rate using single-pass inference (19.61% in English and a baseline-leading 18.63% in Chinese). Furthermore, under a time budget equivalent to strong API models, our method samples ten reasoning paths, increasing syntax pass rates to 39.22% in English and 37.25% in Chinese. This outperforms the strongest baselines per language—Kimi-K2.7-Code (Team et al., 2025) (English) and Claude-Sonnet-5 (Jiang et al., 2025) (Chinese)—by 18.14 and 19.11 percentage points, respectively. Among syntax-valid outputs, EvoSysML achieves conditional semantic averages of 86.90% and 77.34%, ranking first in English and tying for first in Chinese. Ablation studies further confirm that data evolution, two-stage post-training, and validation-guided candidate selection all contribute to the final performance.

The main contributions of this work are summarized as follows:

Dataset We construct a high-quality 50K training dataset for native SysML v2 code generation. We also introduce EvoSysML-204, a manually verified benchmark containing 204 English and 204 aligned Chinese requirements across three difficulty levels.

Method We propose EvoSysML, a four-stage framework that automatically constructs validated training data and trains a specialized SysML generator with substantially improved performance.

Study We conduct a bilingual evaluation against 12 strong baselines. EvoSysML achieves state-of-the-art syntactic and conditional semantic performance among the evaluated systems, while controlled ablations verify the contribution of each framework stage.

2 Motivation

SysML code generation currently faces two primary limitations. First, there is a severe scarcity of large-scale, high-quality training data that aligns natural-language requirements with validated, na-

Table 1: Analysis of representative requirement–code and SysML resources. *Paired* indicates aligned data pairs; *Native v2* denotes native SysML v2 textual models; *Syntax* and *Semantic* refer to formal code and requirement-level validation, respectively. SysML-PU and OMG-v2 represent the SysML v1 PlantUML corpus and official SysML v2 resources.

Resource	Scale	Paired	Native v2	Syntax	Semantic
PURE (Ferrari et al., 2017)	79 docs; 34K sentences	–	–	–	–
PlantUCD (Alaswad et al., 2026)	9.5K UML pairs	✓	–	–	–
SysML-PU (An et al., 2026)	914 pairs; 5 types	✓	–	✓	–
OMG-v2 (Object Management Group, 2025)	Dozens of examples	–	✓	–	–
Ours	50K pairs	✓	✓	✓	✓

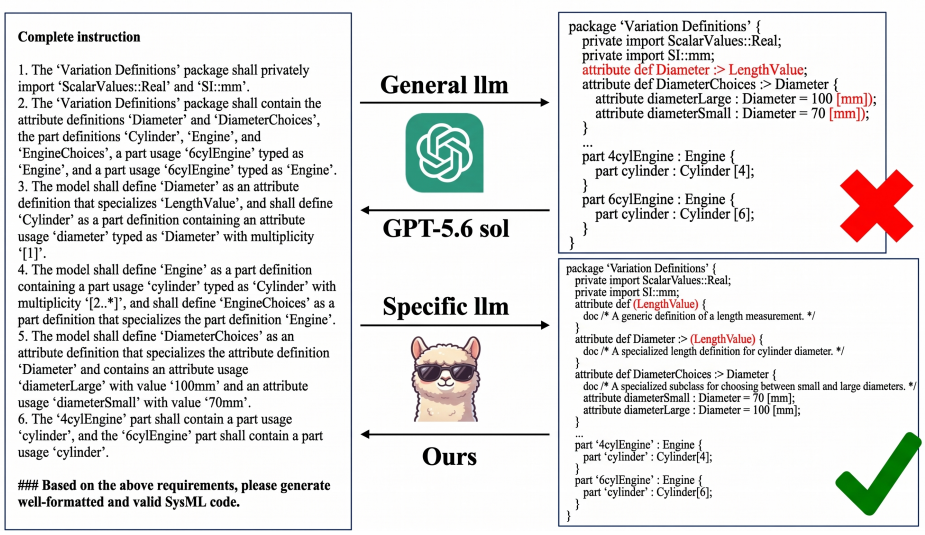

Figure 1: Recorded outputs for the same English instruction. GPT-5.6-Sol generates incomplete structural definitions (e.g., missing base types) and invalid specialization syntax, whereas the SysML-adapted model accurately covers all requested part and attribute hierarchies with kernel-accepted syntax. Kernel outcomes report syntax validation.

tive SysML v2 code. Second, general-purpose LLMs exhibit insufficient adaptability to the domain-specific syntax and formal semantics of SysML. Below, we elaborate on these two points in detail.

Motivation 1: Insufficient Scale, Quality, and Coverage of SysML Training Data. We systematically evaluate related requirement–code and SysML resources. As Table 1 shows, existing datasets lack either aligned code, native SysML v2 representations, or formal validation. Specifically, PURE lacks aligned code; PlantUCD targets UML instead of SysML; and the SysML-PU corpus relies on SysML v1 PlantUML rather than native v2 code. Meanwhile, the official OMG SysML v2 release provides dozens of examples but lacks aligned training pairs. Consequently, a large-scale, validated corpus of native SysML v2 requirement–code pairs remains absent.

Motivation 2: Insufficient Adaptability of General-Purpose LLMs to SysML Code. General-purpose LLMs can produce plausible-looking SysML while violating its formal grammar or omitting necessary element definitions. As shown in Figure 1, GPT-5.6-Sol fails kernel validation because it incorrectly handles structural specifications, such as omitting the base `LengthValue` definition before specialization and struggling with correct syntax for part hierarchies. In contrast, our SysML-adapted model successfully defines all required base elements and correctly expresses the package, attribute, and part relations using kernel-accepted syntax. This gap motivates domain-specific training on validated requirement–code pairs to reduce formally invalid outputs in downstream modeling and verification.

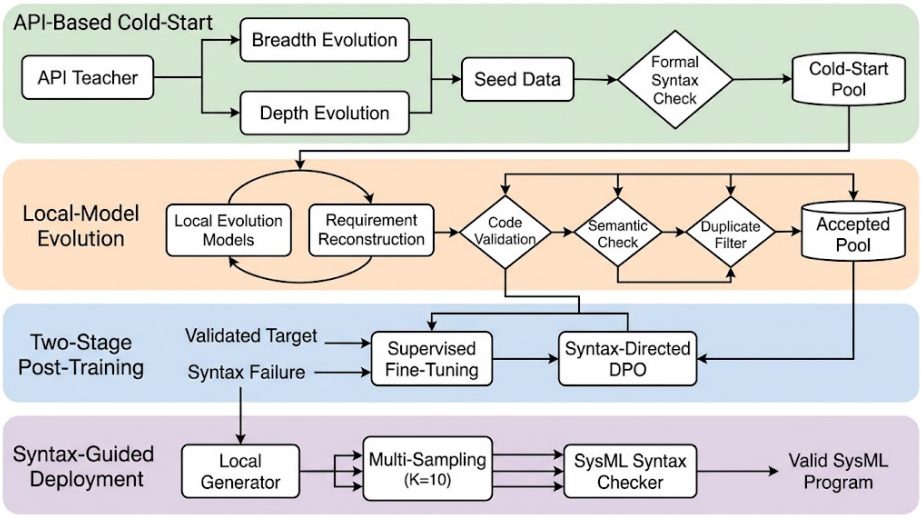

Figure 2: The overall framework of our proposed method.

3 METHOD

High-quality SysML requirement–code pairs are scarce, limiting the training of specialized SysML generators. We therefore propose EVOSYSML, a four-stage training and deployment framework. Stage 1 uses DeepSeek-V4-Pro to construct a cold-start pool through breadth evolution, which expands application scenarios, and depth evolution, which increases program complexity. Stage 2 trains local role models to continue data evolution, reconstruct aligned requirements, and filter the resulting pairs. Stage 3 trains the final generator through supervised fine-tuning followed by syntax-directed direct preference optimization using actual model failures. Stage 4 samples multiple outputs for each requirement and returns the first candidate accepted by the SysML syntax checker.

3.1 API-BASED COLD-START CODE SYNTHESIS

This stage uses DeepSeek-V4-Pro to construct a structurally diverse SysML code pool from a seed pool $\mathcal{P}_0$ of 10 manually authored requirement–code pairs. We define three functional API roles: breadth-evolution and depth-evolution for code synthesis, and requirement-authoring for reconstructing natural-language descriptions of valid programs.

For any seed pair $(r, c) \in \mathcal{P}_0$, where r is the requirement and c the SysML code, the breadth route first transforms r and generates novel code:

$$\widetilde{r}^{\mathrm{B}} = \pi_{\mathrm{B}}^{\mathrm{API}}(r; q_{\mathrm{B}}), \qquad \widetilde{c}^{\mathrm{B}} = G_{\mathrm{API}}(\widetilde{r}^{\mathrm{B}}),$$

where the API acts as both the breadth-evolution role $\pi_{\mathrm{B}}^{\mathrm{API}}$ (using prompt q_{B}) and the code generator G_{API}. The transformed requirement $\widetilde{r}^{\mathrm{B}}$ preserves the core intent while altering the domain (e.g., via cross-domain transfer), yielding the new program $\widetilde{c}^{\mathrm{B}}$.

Conversely, the depth route directly rewrites the existing program c to increase structural complexity:

$$\widetilde{c}^{\mathrm{D}} = \pi_{\mathrm{D}}^{\mathrm{API}}(c; q_{\mathrm{D}}),$$

where the depth-evolution role $\pi_{\mathrm{D}}^{\mathrm{API}}$ applies prompt q_{D} to output the enhanced program $\widetilde{c}^{\mathrm{D}}$. This adds advanced constructs—such as constraint blocks, deeper hierarchies, or state-machine behaviours—without changing the parent model's main intent.

Successfully synthesized outputs from both routes form the cold-start code pool:

$$\mathcal{C}_{\mathrm{API}} = \left\{\widetilde{c}^{\mathrm{B}}\right\} \cup \left\{\widetilde{c}^{\mathrm{D}}\right\}.$$

Every candidate in $\mathcal{C}_{\mathrm{API}}$ is verified by a SysML parser. Only syntax-valid programs are retained for subsequent local-model adaptation; failed generations are stored exclusively as error records. For clarity, we provide a reference workflow example in Appendix A.6.

3.2 Local-Model-Accelerated Data Evolution and Requirement Reconstruction

This stage iterates on the API-generated cold-start pool to construct a large-scale, high-quality dataset. To reduce costs and scale evolution, we replace the API with local small models (e.g., Qwen3.5-9B). Initially trained on the cold-start trajectories to assume the three functional roles, these local models are cyclically fine-tuned as the data pool expands to continuously improve their generation efficiency.

At iteration t, a parent pair (r, c) from the current pool $\mathcal{P}_t$ is evolved. The breadth route expands scenario coverage: the local breadth-evolution model $\pi_{\mathrm{B}}^{\mathrm{loc}}$ (using prompt q_{B}) transforms requirement r into $\widetilde{r}_t^{\mathrm{B}}$, and the local generator G_{ϕ_t} (with weights ϕ_t) synthesizes new code $\widetilde{c}_t^{\mathrm{B}}$:

$$\widetilde{r}_t^{\mathrm{B}} = \pi_{\mathrm{B}}^{\mathrm{loc}}(r; q_{\mathrm{B}}), \qquad \widetilde{c}_t^{\mathrm{B}} = G_{\phi_t}(\widetilde{r}_t^{\mathrm{B}}).$$

Alternatively, the depth route adds advanced SysML constructs by directly modifying the code:

$$\widetilde{c}_t^{\mathrm{D}} = \pi_{\mathrm{D}}^{\mathrm{loc}}(c; q_{\mathrm{D}}),$$

where the local depth-evolution model $\pi_{\mathrm{D}}^{\mathrm{loc}}$ uses prompt q_{D} to produce structurally enriched code $\widetilde{c}_t^{\mathrm{D}}$.

For each valid generated program (denoted generically as $\widetilde{c}_t$), the local requirement-authoring model $\pi_{\mathrm{R}}^{\mathrm{loc}}$ reconstructs a highly aligned natural-language requirement $\widetilde{r}_t$ via prompt q_{R}:

$$\widetilde{r}_t = \pi_{\mathrm{R}}^{\mathrm{loc}}(\widetilde{c}_t; q_{\mathrm{R}}).$$

This ensures the requirement is strictly grounded in the elements and behaviors actually implemented in $\widetilde{c}_t$.

A candidate pair $(\widetilde{r}_t, \widetilde{c}_t)$ joins the accepted set $\mathcal{A}_{t+1}$ only if it passes three strict filters:

$$\mathcal{A}_{t+1} = \{(\widetilde{r}_t, \widetilde{c}_t) \mid V_{\mathrm{code}}(\widetilde{c}_t) = 1,\ V_{\mathrm{sem}}(\widetilde{r}_t, \widetilde{c}_t) = 1,\ V_{\mathrm{dup}}(\widetilde{c}_t) = 0\},$$
$$\mathcal{P}_{t+1} = \mathcal{P}_t \cup \mathcal{A}_{t+1}.$$

Here, V_{code} verifies formal SysML syntax, V_{sem} ensures requirement–code consistency, and V_{dup} removes duplicates. The updated pool $\mathcal{P}_{t+1}$ accumulates these pairs until reaching a target budget at iteration T. The final high-quality pool $\mathcal{P}_T$ is then used in Stage 3 to train the ultimate target generator.

3.3 Two-Stage Post-Training of the SysML Generator

To train a specialized model for SysML code generation, we use the Stage 2 pool $\mathcal{P}_T$ to train the target generator via supervised fine-tuning (SFT) followed by syntax-directed correction.

Supervised fine-tuning. Let $\mathcal{D}_{\mathrm{SFT}} = \mathcal{P}_T$. Starting from a base language model, we optimize its parameters θ via the negative log-likelihood loss $\mathcal{L}_{\mathrm{SFT}}$:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\sum_{(r,c)\in\mathcal{D}_{\mathrm{SFT}}} \log p_\theta(c \mid r),$$

where $p_\theta(c \mid r)$ is the probability of generating the correct SysML code c given requirement r. This stage teaches the model fundamental SysML structural conventions and requirement-to-code mappings.

Syntax-directed preference optimization. Since the SFT model may still produce syntactically invalid code, we construct a preference dataset $\mathcal{D}_{\mathrm{DPO}}$ to explicitly correct these residual errors:

$$\mathcal{D}_{\mathrm{DPO}} = \left\{(r, c^+, c^-) \;\middle|\; (r, c^+) \in \mathcal{D}_{\mathrm{SFT}},\ V_{\mathrm{code}}(c^-) = 0\right\}.$$

Here, c^+ is the validated program for requirement r, and c^- is an SFT-generated output failing the syntax checker ($V_{\mathrm{code}}(c^-) = 0$). Using the SFT checkpoint as the reference policy p_{ref}, we minimize the DPO loss $\mathcal{L}_{\mathrm{DPO}}$:

$$\mathcal{L}_{\mathrm{DPO}}(\theta) = -\mathbb{E}_{(r,c^+,c^-)\sim\mathcal{D}_{\mathrm{DPO}}} \log\sigma\left(\beta\left[\log\frac{p_\theta(c^+ \mid r)}{p_{\mathrm{ref}}(c^+ \mid r)} - \log\frac{p_\theta(c^- \mid r)}{p_{\mathrm{ref}}(c^- \mid r)}\right]\right),$$

where σ is the sigmoid function and β controls deviation from p_{ref}. This formulation directly exploits task-specific syntax failures as negative preference signals, improving structural reliability without requiring human annotation.

3.4 SYNTAX-GUIDED MULTI-SAMPLING DEPLOYMENT

For reliable deployment, we employ repeated sampling to generate $K = 10$ independent candidates $c^{(j)}$ for each requirement r at temperature T:

$$c^{(j)} \sim p_{\theta^\star}(\cdot \mid r; T), \qquad j = 1, \ldots, K,$$

where $p_{\theta^\star}$ is the final trained generator and j is the index. We deploy the first candidate index $j^\star$ that passes the syntax checker:

$$j^\star = \min \left\{ j \in \{1, \ldots, K\} \mid V_{\text{code}}(c^{(j)}) = 1 \right\}.$$

Over an evaluation set of N requirements indexed by i, cumulative validity under this repeated sampling is measured by:

$$\text{Syntax@}\, K = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1} \left[\exists j \le K : V_{\text{code}}(c_i^{(j)}) = 1 \right],$$

where $c_i^{(j)}$ is the j-th candidate for requirement i, and indicator $\mathbf{1}[\cdot]$ equals 1 if at least one valid program exists. This inference-time selection operates without requiring additional training data.

4 EXPERIMENTAL

4.1 EXPERIMENTAL SETUP

Datasets. Our dataset comprises training and test sets, detailed below.

Training data (EvoSysML-50K). We construct 50,677 validated requirement–code pairs with the proposed breadth–depth evolution pipeline. The final generator uses a Breadth:Depth ratio of 1:3, determined via a validation set split from the training data; the effectiveness of this ratio is subsequently demonstrated in Section 4.4. Every retained pair passes the syntax, semantic, and duplicate filters defined in the method.

Test data (EvoSysML-204). We curate 204 strictly de-contaminated SysML programs (ensuring zero AST or semantic overlap with the training seeds) from the official SysML v2 pilot repository (Systems-Modeling, 2026) and construct aligned English and Chinese requirements. Every pair is manually checked for semantic alignment. A fixed structural-complexity rubric assigns 69 examples to Low, 68 to Medium, and 67 to High difficulty; these labels are computed before model evaluation and shared by both languages.

Compared Models. We compare our model with 12 baselines in three groups: **(1) General-purpose API models**: Claude-Sonnet-5 (Jiang et al., 2025), GPT-5.6-Sol (Achiam et al., 2023), DeepSeek-V4-Pro (Xu et al., 2026), and GLM-5.2 (Zeng et al., 2025); **(2) Code-specialized models**: Kimi-K2.7-Code (Team et al., 2025), Qwen3-5.9B-Coder (Team, 2026), and Seed-Coder-8B-Instruct (Seed et al., 2025); and **(3) Locally deployable models**: Qwen3-8B (Team, 2026), Qwen3-5.9B (Team, 2026), Qwen3-8.27B (Team, 2026), GLM-Z1-9B-0414 (Zeng et al., 2025), and Gemma-4-E4B-it (Team et al., 2026).

Implementation Details. All trainable components (the three auxiliary role adapters and final generator) use Qwen3.5-9B (Team, 2026). SFT and syntax-directed DPO are implemented via LLaMA-Factory (Zheng et al., 2024) and LoRA ($r = 8$, $\alpha = 16$). Both are trained for three epochs using bf16, per-device batch size 1, 15,000-token cutoff, cosine scheduling, 0.1 warmup ratio, and seed 42. SFT uses a learning rate of 10^{-4} and 16 gradient-accumulation steps. DPO continues from the SFT checkpoint with a 5×10^{-6} learning rate, 8 accumulation steps, $\beta = 0.1$, and sigmoid loss. Section 4.4 ablates the preference-data budget. In primary comparisons, models generate a single response per requirement (temperature 0, top-$p = 1$, max 8,192 new tokens). Local models are served via vLLM (Kwon et al., 2023) (32,768-token context). We also evaluate our full deployment pipeline using a multi-sampling setting (temperature 0.6, top-$p = 0.95$). All experiments run on NVIDIA RTX PRO 6000 GPUs.

Evaluation Metrics. We evaluate each generated SysML program against its input requirement using formal syntax validity and semantic fidelity.

Table 2: Syntax pass rate(SPR) (%) on EvoSysML-204.

		English				Chinese			
Group	**Model/setting**	**Low**	**Medium**	**High**	**Overall**	**Low**	**Medium**	**High**	**Overall**
General-purpose API	Claude-Sonnet-5	39.13	14.71	2.99	19.12	36.23	16.18	1.49	18.14
	GPT-5.6-Sol	33.33	10.29	4.48	16.18	34.78	13.24	2.99	17.16
	DeepSeek-V4-Pro	13.04	2.94	1.49	5.88	14.49	4.41	0.00	6.37
	GLM-5.2	18.84	2.94	0.00	7.35	15.94	2.94	0.00	6.37
Code-specialized	Kimi-K2.7-Code	40.58	17.65	4.48	21.08	33.33	11.76	4.48	16.67
	Qwen3-5.9B-Coder	1.45	0.00	0.00	0.49	0.00	0.00	0.00	0.00
	Seed-Coder-8B-Instruct	0.00	0.00	0.00	0.00	1.45	0.00	0.00	0.49
Locally deployable	Qwen3-8B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Qwen3-5.9B	1.45	0.00	0.00	0.49	1.45	0.00	0.00	0.49
	Qwen3-8.27B	2.90	0.00	0.00	0.98	2.90	1.47	0.00	1.47
	GLM-Z1-9B-0414	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Gemma-4-E4B-it	0.00	1.47	0.00	0.49	0.00	0.00	0.00	0.00
Ours	Single-pass inference	39.13	14.71	4.48	19.61	36.23	16.18	2.99	18.63
	Repeated sampling (10)	**62.32**	**36.76**	**17.91**	**39.22**	**57.97**	**35.29**	**17.91**	**37.25**

(a) Syntax metric. We evaluate the formal validity of the generated SysML programs using the Syntax Pass Rate (SPR). Let $V_{\text{syn}}(c_i) = 1$ if the final program c_i generated for requirement i successfully passes the SysML validation kernel, and 0 otherwise. For N test requirements, SPR is defined as:

$$\text{SPR} = \frac{1}{N} \sum_{i=1}^{N} V_{\text{syn}}(c_i).$$

Note that our proposed method leverages efficient local inference to perform multiple fast samplings, returning the first syntactically valid candidate. This metric reflects the end-to-end reliability of the generation pipeline.

(b) Semantic metrics. We measure semantic fidelity using requirement coverage (Req.) and entity recall (Ent.), and use their mean as the overall semantic score (SemAvg). Let $F(r_i)$ denote the normalized facts in requirement r_i, $F(c_i)$ the facts recovered from generated program c_i, and $E(r_i)$ and $E(c_i)$ their respective entity sets. We use $M(E(r_i), E(c_i))$ for normalization- and alias-aware entity matches. The three scores are computed as

$$\text{Req}_i = \frac{|F(r_i) \cap F(c_i)|}{|F(r_i)|}, \quad \text{Ent}_i = \frac{|M(E(r_i), E(c_i))|}{|E(r_i)|}, \quad \text{SemAvg}_i = \frac{\text{Req}_i + \text{Ent}_i}{2}.$$

Dataset-level semantic scores average these quantities over syntax-valid generations and are reported together with SPR.

4.2 Syntax Performance Comparison Across Models

To evaluate the formal reliability of SysML generation, we compare our model against general-purpose API, code-specialized, and locally deployable baselines on EvoSysML-204. Specifically, we evaluate our method using the repeated sampling strategy (**generating 10 candidate samples**) detailed in Section 3.4, while also reporting its performance under single-pass inference for direct comparison. Table 2 presents the difficulty-stratified and pooled syntax pass rates across English and Chinese. **Overall conclusion: our model achieves the highest average bilingual syntax pass rate using single-pass inference, and multi-sampling further maximizes performance under a time budget equivalent to large API models. (a)** With single-pass inference, our model leads in Chinese (18.63%) and ranks closely second in English (19.61%) behind Kimi-K2.7-Code (21.08%), securing the highest overall bilingual average against these massive APIs. **(b)** Other code-specialized and local models achieve at most 1.47% overall, indicating that general code pretraining alone cannot ensure SysML grammar adherence. This stark performance gap underscores the necessity of targeted post-training, as standard zero-shot prompting fails to consistently navigate SysML's rigid

Table 3: Conditional semantic fidelity (%) on syntax-valid outputs.

		English			Chinese		
Group	**Model**	**Req.**	**Ent.**	**SemAvg**	**Req.**	**Ent.**	**SemAvg**
General	Claude-Sonnet-5	97.98	75.43	86.71	99.52	55.16	**77.34**
	GPT-5.6-Sol	98.66	75.10	86.88	98.26	53.37	75.82
	DeepSeek-V4-Pro	**99.44**	72.83	86.14	**99.55**	51.79	75.67
	GLM-5.2	94.44	74.68	84.56	96.43	52.63	74.53
Code	Kimi-K2.7-Code	99.12	74.32	86.72	97.91	54.09	76.00
Ours	Repeated sampling (10)	97.25	**76.55**	**86.90**	96.62	**58.05**	**77.34**

Table 4: Ablations of preference tuning and breadth–depth composition. Each metric cell reports ZH / EN / Avg (%).

Factor	**Setting**	**SPR ZH / EN / Avg**	**Conditional SemAvg ZH / EN / Avg**
Preference tuning	SFT	16.18 / 18.63 / 17.40	76.83 / **85.90** / 81.37
	DPO-100	17.65 / 18.63 / 18.14	**77.57** / 85.43 / **81.50**
	DPO-200	**18.63** / **19.61** / **19.12**	76.53 / 84.87 / 80.70
	DPO-400	17.65 / 17.65 / 17.65	77.04 / 85.10 / 81.07
B:D composition	Depth-only	10.29 / 14.22 / 12.25	78.25 / 85.40 / 81.82
	Breadth:Depth 1:3	11.27 / **14.71** / **12.99**	78.85 / 85.98 / 82.42
	Breadth:Depth 1:1	**11.76** / 12.75 / 12.25	76.87 / 85.38 / 81.13
	Breadth:Depth 3:1	10.29 / 12.75 / 11.52	76.96 / **88.03** / 82.50
	Breadth-only	11.27 / 10.78 / 11.03	**81.47** / 85.85 / **83.66**

structural constraints. **(c)** Generating ten validation-guided samples locally takes the same latency as a single query to large API models (see Appendix B). Under this matched time budget, our model reaches 39.22% in English and 37.25% in Chinese, vastly outperforming all single-pass baselines and demonstrating highly efficient candidate exploration. Fixed role-prompt contracts and training-scale comparisons are detailed in Appendix A.1 and Appendix C.

4.3 Semantic Performance of Generated SysML Code

To verify the semantic performance of the SysML code generated by our model, we evaluated all syntax-valid outputs produced under our multi-sampling setting against the top five performing baselines. Since strict syntactic validity must not come at the expense of user design intent, this analysis specifically measures whether the successfully parsed code accurately reflects the original textual constraints and instantiates the necessary structural elements. Table 3 compares requirement coverage (Req.), entity recall (Ent.), and their average (SemAvg) conditionally over these successful generations. **Overall conclusion: our model maintains top-tier semantic fidelity even while producing a significantly larger volume of valid programs. (a)** Among its syntax-valid outputs, Ours obtains the highest English SemAvg (86.90%). **(b)** In Chinese, Ours reaches 77.34%, tying with Claude-Sonnet-5 for the highest SemAvg. **(c)** A consistent gap between Req. and Ent. across all models highlights that recovering surface facts does not guarantee complete structural entity generation. Crucially, because these metrics are calculated exclusively on valid outputs, they must be interpreted alongside syntax pass rates. For instance, our model achieves these leading semantic scores over a much larger and more diverse subset of successfully parsed programs, whereas baselines are evaluated on a narrower subset of easier tasks they managed to parse.

4.4 Ablation Studies

To systematically investigate the individual contributions of our key modeling choices and inference strategies to formal compliance and semantic quality, we evaluate the impact of preference tuning, breadth–depth composition, and candidate construction in Table 4. This analysis aims to

clarify whether grammatical constraints can be effectively reinforced without compromising generation fidelity across languages. Further details on role prompts (Appendix A.1), training scales (Appendix C), and generation strategies (Appendix D, Appendix E) are deferred to the appendices. **Overall conclusion: design choices produce complementary but non-monotonic trade-offs between syntactic validity and semantic fidelity. (a) Preference tuning:** Compared to standard SFT (17.40% SPR), preference alignment steadily enhances formal syntax adherence; DPO-200 yields the highest average SPR (19.12%), whereas DPO-100 maximizes SemAvg (81.50%), showing that expanding the preference budget does not monotonically improve both metrics. **(b) Evolution direction:** The depth-heavy 1:3 mixture maximizes average syntax (12.99%), while breadth-only training peaks in SemAvg (83.66%). We adopt the 1:3 ratio since formal validity is the primary deployment prerequisite. **(c) Candidate construction:** Repeated sampling (@10) reaches 39.22% in English and 37.25% in Chinese, clearly outperforming rewrite-based sampling (31.86% and 32.84%).

5 Related Work

5.1 Evolution of SysML and LLM-Assisted Modeling

SysML is a core language for MBSE, representing system requirements, structure, behavior, constraints, and verification in a unified model. SysML v2 further provides standardized textual notation and formal semantics for model exchange, automated analysis, and tool integration (Hause et al., 2006; Bajaj et al., 2022). However, manually translating natural-language requirements into valid and semantically complete SysML models is costly and requires substantial expertise (Dehn et al., 2025; Wang et al., 2025). LLMs offer a promising alternative, and existing studies have explored requirement extraction, prompt-based generation, lightweight fine-tuning, and validation-guided repair (Ruan et al., 2023; DeHart, 2024; An et al., 2026; Liu et al., 2026). These studies establish the feasibility of LLM-assisted SysML modeling, but remain limited to small-scale exploration, restricted diagram types, or inference-time refinement. **To the best of our knowledge, our work proposes the first automated method for generating high-quality SysML code and trains a dedicated model specifically for SysML code generation tasks.**

5.2 Code LLMs

Code LLMs translate natural-language intent into executable programs, with representative models (e.g., CodeT5, StarCoder, Code Llama, DeepSeek-Coder, and Qwen2.5-Coder) learning from massive repositories of mainstream languages (Wang et al., 2021; Li et al., 2023; Rozière et al., 2023; Guo et al., 2024; Hui et al., 2024). Their specialization increasingly relies on synthetic data and model-driven curation, such as instruction generation, task evolution, and execution feedback (Wang et al., 2023; Luo et al., 2024; Wei et al., 2024; Ahmad et al., 2025; Zhang et al., 2025). For general-purpose languages like Python or Java, these techniques thrive because they can leverage robust execution environments and rich open-source unit tests to automatically verify code correctness. However, these methods depend on abundant pretraining corpora, offering limited guidance for low-resource formal languages where aligned data are scarce and generated programs must satisfy strict grammatical and semantic constraints. **Our method addresses this gap via breadth–depth data evolution to expand SysML topics and structural complexity, coupled with strict validation to construct reliable supervision for training a specialized SysML generator.**

6 Conclusion

We presented EvoSysML, a framework for training and deploying a specialized SysML v2 generator via breadth–depth data evolution, two-stage post-training, and validation-guided sampling. Evaluated against 12 baselines on EvoSysML-204, our model achieves the highest average bilingual syntax pass rate using single-pass inference. Furthermore, under a time budget equivalent to large API models, ten-candidate sampling maximizes syntax performance to 39.22% (EN) and 37.25% (ZH) while maintaining top-tier conditional semantic fidelity (86.90% EN, 77.34% ZH). Ablations confirm the critical value of validated data evolution and syntax-aware adaptation for reliable low-resource formal code generation.

AI USE STATEMENT

In this work, we used generative AI tools solely for text editing and language polishing to improve the readability and grammatical flow of the manuscript. We have not used generative AI tools for experimental execution, data contamination, or the formulation of our core research ideas. We have meticulously reviewed all AI-assisted language revisions to ensure absolute accuracy and fidelity to our original scientific intent. We take full responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To ensure the full reproducibility of our experimental results and the proposed framework, we have provided the complete, anonymized source code in the supplementary materials. This includes all data evolution scripts, training and validation pipelines, and evaluation kernels required to replicate the results presented in this paper.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *CoRR*, abs/2504.04030, 2025. doi: 10.48550/ARXIV.2504.04030. URL https://doi.org/10.48550/arXiv.2504.04030.

Feisal Alaswad, Poovammal E, and Batoul Aljaddouh. Plantucd: A dataset of software requirements and corresponding plantuml-based class diagrams. 2026.

Baoran An, Tao Lei, and Guangtai Tian. An llm-based framework for the automatic generation of sysml models. *Sensors*, 26(16):5133, 2026.

Manas Bajaj, Sanford Friedenthal, and Ed Seidewitz. Systems modeling language (sysml v2) support for digital engineering. *Insight*, 25(1):19–24, 2022.

John K DeHart. Leveraging large language models for direct interaction with sysml v2. In *INCOSE International Symposium*, volume 34, pp. 2168–2185. Wiley Online Library, 2024.

Simon Dehn, Simon Schnürer, Georg Jacobs, and Gregor Höpfner. Generating sysml v2 models from natural language requirements using large language models. In *2025 IEEE International Symposium on Systems Engineering (ISSE)*, pp. 1–7. IEEE, 2025.

Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. PURE: A dataset of public requirements documents. In Ana Moreira, João Araújo, Jane Hayes, and Barbara Paech (eds.), *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pp. 502–505. IEEE Computer Society, 2017. doi: 10.1109/RE.2017.29. URL https://doi.org/10.1109/RE.2017.29.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Matthew Hause et al. The sysml modelling language. In *Fifteenth European systems engineering conference*, volume 9, pp. 1–12, 2006.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186, 2024. doi: 10.48550/ARXIV.2409.12186. URL https://doi.org/10.48550/arXiv.2409.12186.

Qile Jiang, Zhiwei Gao, and George Em Karniadakis. Deepseek vs. chatgpt vs. claude: A comparative study for scientific computing and scientific machine learning tasks. *Theoretical and Applied Mechanics Letters*, 15(3):100583, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023, 2023. URL https://openreview.net/forum?id=KoFOg41haE.

Yuhao Liu, Junjie Hou, Haolong Zhang, and Zhiang Lu. Mas4sysml: A multi-agent framework for sysml v2 model generation from natural language. *Journal of Visualized Experiments (JoVE)*, (231):e70395, 2026.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

Object Management Group. Systems Modeling Language (SysML) Version 2 Specification. Specification, Object Management Group (OMG), 2025. URL https://www.omg.org/spec/SysML.

Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):101–111, 2011.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Kun Ruan, Xiaohong Chen, and Zhi Jin. Requirements modeling aided by chatgpt: An experience in embedded systems. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pp. 170–177. IEEE, 2023.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

Systems-Modeling. SysML v2 Pilot Implementation. https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation, 2026. GitHub repository, accessed 19 September 2026.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Qwen Team. Qwen3.5-omni technical report. *CoRR*, abs/2604.15804, 2026. doi: 10.48550/ARXIV.2604.15804. URL https://doi.org/10.48550/arXiv.2604.15804.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 13484–13508. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.ACL-LONG.754. URL https://doi.org/10.18653/v1/2023.acl-long.754.

Yuan Wang, Ning Ge, Jiangxi Liu, Zhilong Cao, Zheping Chen, and Chunming Hu. Generating sysml behavior models via large language models: an empirical study. In *Proceedings of the 16th International Conference on Internetware*, pp. 366–377, 2025.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 8696–8708. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.EMNLP-MAIN.685. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/wei24h.html.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. Seed-coder: Let the code model curate data for itself. *CoRR*, abs/2506.03524, 2025. doi: 10.48550/ARXIV.2506.03524. URL https://doi.org/10.48550/arXiv.2506.03524.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A PROMPTS

A.1 ROLE PROMPT TEMPLATES

We use fixed prompt contracts for the three data-construction roles and the final requirement-to-code generator. Each trajectory records the role, strategy identifier, parent sample, verified facts, model output, and validation result. Role trajectories remain separate; only validated requirement–code pairs enter the final generator pool.

Let $\{r, c, F\}$ denote the parent requirement, parent SysML code, and verified fact list. The placeholders used by the prompts are written as literal text; they are not interpreted as LaTeX commands.

Shared preservation contract. All role prompts begin with the following instruction:

> You are a senior systems engineer and SysML v2 expert. Follow the selected strategy exactly. Preserve every required parent fact, including identifiers, entity kinds, attributes, values, units, multiplicities, relations, constraints, interfaces, behaviours, and traceability links. Do not silently delete or rename a parent fact. Mark newly introduced facts explicitly. Do not invent unsupported numerical values, physical properties, relations, or behaviours. Return only the requested JSON object, with no Markdown fences or commentary.

Table 5: Role-level prompt contracts used during data construction.

Role	Input	Output contract
Breadth exploration	Parent requirement r (or an authoring-role reconstruction)	Create one complete requirement under the selected breadth strategy. Preserve all required parent facts and identify newly introduced facts. Return a JSON object with the fields `role`, `strategyId`, `parentId`, `requirement`, `preservedFactIds`, and `addedFactIds`.
Depth exploration	Parent SysML code c and verified fact list F	Apply exactly one structural operation. Return one complete SysML model, not a patch, together with `role`, `strategyId`, `parentId`, `changeSummary`, `preservedFactIds`, and `addedFactIds`.
Requirement authoring	Validated candidate code c and verified facts F	Write one auditable requirement for each assigned fact. Return an ordered JSON list of objects with the fields `factId` and `text`. Every fact identifier occurs exactly once, and unsupported facts are forbidden.
Final generator	Natural-language requirement r	Return only complete SysML v2 code. Preserve all entities, attributes, values, constraints, relations, interfaces, behaviours, and traceability statements. Do not return JSON, Markdown, or an explanation.

A.2 BREADTH STRATEGY POOL

One strategy is sampled from $\mathcal{S}_{\mathrm{B}} = \{\mathrm{B1}, \ldots, \mathrm{B6}\}$. Breadth changes the requirement space while preserving the parent facts needed for a valid modeling task.

The breadth prompt is instantiated with the following literal fields: `parentRequirement`, `strategyId`, `strategyInstruction`, and `verifiedFactList`. The model returns exactly one JSON object and no code block.

A.3 DEPTH STRATEGY POOL

One strategy is sampled from $\mathcal{S}_{\mathrm{D}} = \{\mathrm{D1}, \ldots, \mathrm{D6}\}$. Depth acts directly on SysML code and increases structural, behavioural, or traceability complexity while preserving the parent model.

The depth prompt receives the literal fields `parentCode`, `verifiedFactList`, `strategyId`, and `strategyInstruction`. It returns one complete SysML model in the `code` field and a concise `changeSummary`.

Table 6: Breadth exploration strategies.

ID	Strategy	Instruction
B1	Domain transfer	Move the task to another engineering domain while preserving structural and logical complexity. Replace domain-specific terms with a coherent target-domain vocabulary.
B2	System-composition expansion	Introduce or reorganize subsystems, components, or stakeholders. Preserve parent functions and relations.
B3	Functional extension	Add a compatible system capability and connect it to existing entities or interfaces without invalidating the parent task.
B4	Decomposition and refinement	Refine a coarse system or component into interacting subcomponents while exposing containment, typing, and interface relations.
B5	Alternative expression	Reorganize the requirement and change its sentence structure while preserving every fact, value, constraint, relation, and item count.
B6	Scenario or environment expansion	Add a coherent operating scenario, environment, lifecycle phase, or system boundary. Do not invent numerical limits that are absent from the parent facts.

Table 7: Depth exploration strategies.

ID	Strategy	Instruction
D1	Constraint and parametric enrichment	Add one meaningful constraint, binding, calculation, or quantitative relation referencing existing or explicitly introduced elements.
D2	Hierarchy and inheritance deepening	Introduce a coherent specialization hierarchy with specializations or redefinitions. Preserve parent members and make inherited features explicit.
D3	Behaviour and state-machine enrichment	Add states, transitions, events, actions, or behavioural conditions and connect them to existing elements.
D4	Requirement traceability	Add requirement definitions or usages and valid `satisfy`, `verify`, or `derive` relations with resolved endpoints.
D5	Ports, interfaces, connections, and flows	Add compatible ports or interfaces and connect endpoints while preserving directions, types, and multiplicities.
D6	Allocation and interaction refinement	Add allocation, dependency, communication, or interaction structure without unresolved endpoints or changes to existing constraints.

A.4 Requirement Authoring and Final Generation

After formal code validation, the requirement-authoring role converts verified facts into auditable natural-language requirements. The final generator then converts the resulting requirement into SysML code. Their contracts are kept separate as follows.

Requirement-authoring contract.

- Write exactly one concise, testable requirement for each assigned fact.
- Preserve identifiers, entity kinds, attributes, values, units, multiplicities, relation endpoints, directions, constraints, and behaviour conditions.
- Cover every assigned fact exactly once. Do not merge unrelated facts or omit details.
- Do not repair invalid code, invent unsupported values, or introduce unsupported system intent.

Final-generator contract.

- Input: one natural-language requirement and the fixed bilingual generation suffix.
- Preserve every entity, attribute, value, constraint, relation, interface, behaviour, and traceability statement.
- Return only complete SysML v2 code, without JSON, Markdown fences, analysis, or explanation.

A.5 Trajectory and Validation Contract

Each trajectory records the role, strategy identifier, prompt version, parent identifier and hash, model identifier, seed, output and output hash, and a validation object. The validation object contains parser status, syntax status, semantic coverage, entity recall, triple recall, duplicate status, and a rejection reason when applicable. A candidate enters the final EvoSysML-50K pool only after formal validation, semantic consistency checks, and duplicate filtering. Role trajectories remain auxiliary data and are not directly merged into the final generator SFT set.

A.6 Worked Example of the API Cold-Start Mappings

The following synthetic example follows one seed pair (r, c) through the exact mappings in Section 3.1. It is an explanatory example, not an additional experiment. The notation is instantiated as follows:

Symbol	Concrete meaning in this example
r	The natural-language requirement for a water-pump system.
c	The parent SysML model that implements r.
q_{B}	The instruction to transfer the pump task to spacecraft thermal control while preserving the flow-transfer intent.
$\pi_{\mathrm{B}}^{\mathrm{API}}$	The API breadth role that rewrites r into $\widetilde{r}^{\mathrm{B}}$.
G_{API}	The API code generator that maps $\widetilde{r}^{\mathrm{B}}$ to $\widetilde{c}^{\mathrm{B}}$.
q_{D}	The instruction to add interfaces, pressure, and a physical constraint to c.
$\pi_{\mathrm{D}}^{\mathrm{API}}$	The API depth role that directly rewrites c into $\widetilde{c}^{\mathrm{D}}$.

1. Seed pair (r, c). The parent requirement states: "The system shall transfer water from a reservoir to a storage tank using a motor-driven pump and shall expose the pump flow rate." The corresponding parent code contains only the core parts:

Code c:
```
package WaterPump {
part def Pump;
part def Motor;
part pump:  Pump;
part motor:  Motor;
attribute flowRate:  Real;
}
```

2. Breadth route: change the requirement, then generate code. The breadth prompt q_{B} asks for a domain transfer to spacecraft thermal control while preserving the original flow-transfer intent. Thus, $\pi_{\mathrm{B}}^{\mathrm{API}}$ changes the nouns and system context but keeps the required function:

Transformed requirement $\widetilde{r}^{\mathrm{B}}$: The spacecraft thermal-control loop shall transfer coolant from a reservoir to a radiator using a motor-driven pump and shall expose coolant flow rate.

Generated code $\widetilde{c}^{\mathrm{B}} = G_{\mathrm{API}}(\widetilde{r}^{\mathrm{B}})$:

```
package ThermalControl {
part def Reservoir; part def Radiator;
part def CoolantPump; part def PumpMotor;
attribute coolantFlowRate:  Real;
}
```

The code is newly generated from $\widetilde{r}^{\mathrm{B}}$; it is not a line-by-line rewrite of c. This is the concrete meaning of $r \xrightarrow{\pi_{\mathrm{B}}^{\mathrm{API}}} \widetilde{r}^{\mathrm{B}} \xrightarrow{G_{\mathrm{API}}} \widetilde{c}^{\mathrm{B}}$.

3. Depth route: rewrite the parent code directly. The depth prompt q_{D} keeps the water-pump domain and asks for one structural enrichment. The depth role therefore acts on c itself:

Generated code $\widetilde{c}^{\mathrm{D}} = \pi_{\mathrm{D}}^{\mathrm{API}}(c; q_{\mathrm{D}})$:

```
part def Pump {
port inlet; port outlet;
attribute pressure:  Real;
constraint flowConstraint { flowRate = pressure / resistance; }
}
```

Here the parent pump, motor, and flow-rate intent are retained, while ports, pressure, and a quantitative constraint are added. Both generated programs are then checked by the parser; only candidates with $V_{\mathrm{code}} = 1$ enter $\mathcal{C}_{\mathrm{API}}$, and failed candidates remain error records.

4. Continuation through the remaining stages. The same accepted child can be followed through the rest of the framework: the local requirement-authoring role reads the validated $\widetilde{c}_t$ and writes a grounded requirement $\widetilde{r}_t = \pi_{\mathrm{R}}^{\mathrm{loc}}(\widetilde{c}_t; q_{\mathrm{R}})$; the pair is added to $\mathcal{P}_{t+1}$ only when code validity, semantic consistency, and duplicate checks all pass. The resulting pair is then one training example (r, c) for SFT. If the SFT model emits an invalid variant c^- for the same requirement, the validated child is c^+ and the pair (r, c^+, c^-) supplies one syntax-directed DPO example. At deployment, the final generator samples $c^{(1)}, \ldots, c^{(K)}$ for one requirement and the first candidate satisfying $V_{\mathrm{code}} = 1$ is returned. Thus, the single illustrative model follows the complete path *seed pair → evolution → validation → grounded pair → SFT/DPO → validated deployment output*.

B Inference Throughput and Cost-Equivalent Analysis

To address the fairness of comparing our repeated sampling setting (@10) against baseline single-pass API queries (@1), we evaluate both paradigms under a strictly matched practical deployment budget.

Table 8: Estimated inference budget and deployment comparison per requirement. API latency uses conservative end-to-end statistics. The local row denotes batched generation of ten candidates.

Model	Serving	Samples (K)	Throughput (req/s)	Latency (s)	Marginal Cost
Claude-Sonnet-5	Cloud API	1	~ 0.05	18.63	API fee
GPT-5.6-Sol	Cloud API	1	~ 0.06	16.20	API fee
Kimi-K2.7-Code	Cloud API	1	~ 0.01	90.00	API fee
Ours (9B LoRA)	Local vLLM	10	$\sim$ **0.06**	**15.50**	**Zero**

Table 8 presents the estimated inference budget and deployment comparison per requirement, tracking throughput, end-to-end latency, and marginal cost. **Overall conclusion: leveraging optimized local batching enables 10-candidate multi-sampling within the same time and cost constraints**

as a single cloud API call. (a) Latency parity: Generating $K = 10$ batched candidates concurrently via local vLLM takes an estimated 15.5 seconds per requirement. This is even faster than the end-to-end latency of a single cloud API call (16.20s for GPT-5.6-Sol and 18.63s for Claude-Sonnet-5). **(b) Throughput efficiency:** Even when producing ten times the candidate volume, our local pipeline maintains a requirement-level throughput ($\sim$0.06 req/s) completely equivalent to single-pass APIs, thanks to a high local token-generation rate ($\sim$964 output tok/s). **(c) Cost-equivalent fairness:** Requesting 10 independent samples from closed-source APIs strictly multiplies the financial token cost, making $K = 1$ the only realistic commercial baseline. Conversely, local generation incurs zero marginal API fees.

Therefore, comparing our local model at @10 against APIs at @1 provides a fundamentally fair evaluation of practical, cost-equivalent system pipelines. Unadapted local baselines are omitted at @10 because their single-pass grammar capability is near zero; giving them a higher budget yields no meaningful performance gain.

C Effect of Training Data Scale

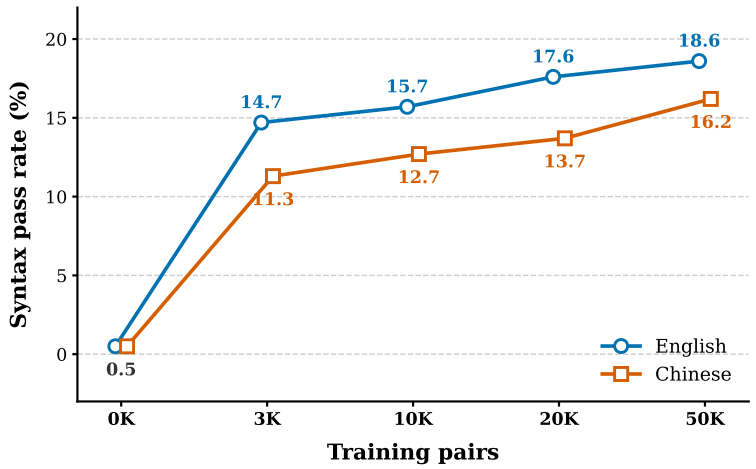

Figure 3: Single-sample syntax pass rate on EvoSysML-204 at different training scales. Filled markers share the matched evaluation contract; hollow 20K markers are retained as protocol-mismatched diagnostics and are not connected to the main trend.

Figure 3 isolates the effect of the number of self-evolved training pairs. Under the matched single-sample protocol, the base model (0K) obtains 0.49% in both languages. With 3K pairs, syntax pass rate rises to 14.71% in English and 11.27% in Chinese; 10K pairs yield 15.69% and 12.75%; and the full 50K set reaches 18.63% and 16.18%. Thus, each matched increase in training scale improves bilingual formal validity. The 20K measurements use a different prompt contract and are shown only as diagnostic points.

D Effect of Candidate Construction

Single sampling generates one candidate from the original requirement. Repeated sampling generates ten candidates from that unchanged requirement, whereas rewrite-based sampling uses the original requirement once and nine cached rewrites under the direct rewrite policy. All three rows in Table 9 use Ours and EvoSysML-204. Rewrite-based sampling reaches 31.86% in English and 32.84% in Chinese. Repeated sampling reaches 39.22% and 37.25%, respectively. The repeated and rewrite-based strategies share the model and test split but use different prompt and decoding contracts, so their comparison measures the stored inference strategies rather than an isolated rewrite effect.

Table 9: Syntax pass rate (%) of Ours under three candidate strategies.

Group	Strategy	English	Chinese	Bilingual Avg.
Ours	Single sample	19.61	18.63	19.12
	Repeated sampling (10)	**39.22**	**37.25**	**38.24**
	Rewrite-based sampling (10)	31.86	32.84	32.35

E CUMULATIVE CANDIDATE BUDGET FROM 1 TO 10

For each requirement, Syntax@k counts success when at least one of the first k attempts passes the SysML kernel. Repeated sampling uses the unchanged requirement, whereas rewrite-based sampling uses the original requirement for the first attempt and cached rewrites for attempts 2–10. Table 10 reports the cumulative curves for both candidate strategies.

Table 10: Cumulative syntax pass rate (%) of Ours as the candidate budget increases for repeated and rewrite-based sampling.

Group	k	Repeated EN	Repeated ZH	Rewrite EN	Rewrite ZH
Ours	1	16.18	18.63	18.63	16.67
	2	23.04	23.04	23.04	19.61
	3	26.47	24.51	25.49	24.02
	4	28.92	28.92	28.43	25.49
	5	30.88	29.90	30.39	27.45
	6	32.84	31.86	31.37	28.92
	7	35.29	34.31	31.37	29.41
	8	36.27	35.78	31.37	30.88
	9	38.24	36.27	31.86	32.35
	10	**39.22**	**37.25**	**31.86**	**32.84**

Both strategies improve as the candidate budget increases. At $k = 10$, repeated sampling reaches 39.22% English and 37.25% Chinese, while rewrite-based sampling reaches 31.86% and 32.84%. These curves quantify additional inference budget and are not training-time improvements.