From Intent to Executable Code: Lifecycle-Aware Intermediate Supervision for LLM Code Generation

Anonymous ACL submission

Abstract

One-step code generation maps an intent directly to an implementation, but provides little supervision over the software-development process that determines whether requirements, behavior, and control flow are preserved. We propose lifecycle-aware intermediate supervision, which trains language models on aligned software-engineering artifact chains consisting of user intent, Software Requirements Specification (SRS), Detailed Design Specification (DDS), SCXML behavioral models, pseudocode, and implementation. We construct 457 complete artifact chains from OPNET, XState, and Simulink, expand 366 training chains into 1,830 stage-wise instruction instances, and evaluate on 91 held-out chains with both stage-specific overlap metrics and test-function-based pass@K, where K=5, for final implementations. Across four open backbones, lifecycle-aware fine-tuning improves implementation pass@K for every model; Qwen3-8B reaches 0.355 pass@K with full-data supervision and is competitive with the strongest prompted baseline, while DeepSeek-Coder-6.7B obtains the largest absolute gain (+0.250). Data-scaling and ablation analyses further show that substantial gains appear with limited supervision and that execution-proximal artifacts, especially SCXML and pseudocode, contribute most to final functional coverage. An anonymous artifact is available at https://anonymous.4open.science/r/4G8r97NR.

1 Introduction

Large language models (LLMs) have substantially advanced automatic code generation, with systems such as Codex, AlphaCode, Code Llama, and recent open code models showing strong results on benchmarks such as HumanEval, APPS, and MBPP (Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021; Li et al., 2022; Roziere et al., 2023; Nijkamp et al., 2022; Li et al., 2023b). Yet most evaluations still frame generation as a direct mapping from a natural-language prompt to code. This one-step formulation leaves the model without explicit supervision over the development process that connects intent to implementation. It is effective for compact programming problems, but provides limited structure for tasks that require requirements clarification, behavioral modeling, and consistent control-flow implementation. As a result, a model may produce syntactically plausible code while silently losing intent, omitting edge cases, or implementing incoherent state transitions.

Recent work has made code generation more structured through chain-of-thought, planning, structured prompting, generated tests, execution feedback, and agentic workflows (Wei et al., 2022; Jiang et al., 2024b; Li et al., 2023a; Chen et al., 2023; Ni et al., 2024; He et al., 2024; Lin et al., 2025). These methods show that process structure is useful, but most of it is introduced at inference time through prompts, agents, or refinement loops rather than learned as reusable software-engineering artifacts. Less is known about whether such artifacts can serve as supervised intermediate representations that smaller or open-source backbones can learn and reuse.

We study this question through *lifecycle-aware intermediate supervision*. Rather than training only on user-intent/code pairs, we construct aligned chains of software-engineering artifacts: user intent, Software Requirements Specification (SRS), Detailed Design Specification (DDS), SCXML behavioral model, pseudocode, and implementation. The resulting task asks a model to learn transformations from informal intent to executable code. Unlike inference-time planning or multi-agent process simulation, these artifacts are explicit training targets, can be evaluated at each stage, and can be removed to estimate their contribution to final correctness.

Our empirical setting focuses on state-machine-oriented and control-flow-heavy programs, where

behavioral artifacts are natural. We collect 457 complete artifact chains from OPNET, XState, and Simulink, expand the 366 training chains into 1,830 stage-wise supervised examples, and evaluate on 91 held-out chains. Because surface-overlap metrics alone are insufficient for code, we treat CodeBLEU (Ren et al., 2020) as auxiliary and foreground test-function-based functional coverage. Specifically, GPT-5 is used to construct and filter test-function lists for each held-out instance, which are then used to compute test-function-based pass@K for generated implementations.

Across general and code-pretrained backbones, lifecycle-aware supervision improves final-stage pass@K for all evaluated models, with Qwen3-8B achieving the strongest full-data result among the fine-tuned models. Data-scale experiments show substantial gains even with 20% of the training data, while Qwen3-8B implementation performance peaks at a moderate supervision budget. Stage-removal ablations further show that removing any artifact weakens final code quality, with behavioral modeling and pseudocode producing the largest drops.

Our contributions are:

- We introduce lifecycle-aware intermediate supervision, which uses aligned software-engineering artifacts as supervised targets between user intent and executable code.
- We construct a state-machine-oriented artifact-chain dataset with 457 complete examples and 1,830 stage-wise training instances.
- We provide test-function-based evaluation and analysis showing improved executable correctness, data efficiency, and the importance of behavioral and algorithmic artifacts.

2 Related Work

2.1 LLM Code Generation and Evaluation

LLM-based code generation is commonly formulated as generating executable programs from natural-language specifications. Codex/HumanEval, MBPP, APPS, AlphaCode, and Code Llama established direct natural-language-to-code generation as a central evaluation setting (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Roziere et al., 2023); open code models such as CodeGen, InCoder, StarCoder, StarCoder2, WizardCoder, DeepSeek-Coder, and Qwen-family models further advanced pretraining and instruction tuning for code (Nijkamp et al., 2022; Fried et al., 2023; Li et al., 2023b; Lozhkov et al., 2024; Luo et al., 2024; Guo et al., 2024; Yang et al., 2025). Surveys of NL2Code and code LLMs summarize the broader progress in model scaling, code pretraining, and instruction tuning (Zan et al., 2023; Jiang et al., 2024a; Zheng et al., 2023).

Evaluation has also shifted from surface similarity toward functional correctness. CodeBLEU adds code-specific syntax and data-flow signals (Ren et al., 2020), while HumanEval-style evaluation uses execution-based pass rates (Chen et al., 2021). More recent benchmarks stress multilingual generation, data-science library use, class-level programming, complex function calls, contamination control, and stronger tests (Cassano et al., 2022; Lai et al., 2023; Du et al., 2023; Zhuo et al., 2024; Jain et al., 2024; Liu et al., 2023). We follow this direction by reporting overlap and CodeBLEU as auxiliary indicators while emphasizing test-function-based pass@K.

2.2 Intermediate Reasoning and Planning for Code

Many recent methods improve code generation by adding intermediate reasoning or refinement. Chain-of-thought prompting elicits natural-language reasoning (Wei et al., 2022); Self-planning decomposes generation into planning and implementation (Jiang et al., 2024b); SCoT structures reasoning with program constructs such as sequence, branch, and loop (Li et al., 2023a); and planning-guided decoding or plan search uses planners, natural-language plans, or public tests during decoding (Zhang et al., 2023; Wang et al., 2024). Test-time feedback and execution-aware methods refine candidate solutions with generated tests, self-feedback, verbal reinforcement, or execution-trace supervision (Chen et al., 2023; Madaan et al., 2023; Shinn et al., 2023; Ni et al., 2024).

These approaches show that intermediate structure matters, but they primarily introduce it at prompting or decoding time. Our work instead learns heterogeneous software-engineering artifacts as supervised targets, including requirements, detailed design, SCXML, pseudocode, and implementation.

2.3 Software-Engineering-Aware LLM Workflows

Another line of work brings LLMs closer to software engineering workflows. ChatDev and MetaGPT coordinate role-specialized agents for development activities (Qian et al., 2023; Hong et al., 2023), while SWE-bench and SWE-agent study repository-level issue resolution and agent-computer interfaces (Jimenez et al., 2023; Yang et al., 2024). Agentless further shows that carefully designed non-agentic repair pipelines can compete with more complex agent systems (Xia et al., 2024). Most closely related, SOEN-101 introduces FlowGen, a multi-agent framework that emulates process models such as Waterfall, TDD, and Scrum (Lin et al., 2025). CoCoST is also related in spirit, combining online search, test generation, and correctness testing for complex code generation (He et al., 2024).

Our work is complementary to these workflow-oriented methods. Rather than designing an inference-time agent organization or refinement loop, we train open backbones on aligned artifact chains and evaluate each stage. This lets us ask a different question: whether lifecycle artifacts provide learnable intermediate supervision for executable code generation.

3 Lifecycle-Aware Intermediate Supervision

3.1 Task Formulation

We formulate lifecycle-aware code generation as supervised learning over a sequence of software-engineering artifacts. For each example, let a_0 denote the user intent and let $a_1, \ldots, a_5$ denote the Software Requirements Specification (SRS), Detailed Design Specification (DDS), SCXML behavioral model, pseudocode, and implementation, respectively. Standard one-step code generation learns a direct mapping $a_0 \rightarrow a_5$. In contrast, our setting learns a set of adjacent transformations

$$a_{s-1} \rightarrow a_s, \quad s \in \{1, \ldots, 5\}, \tag{1}$$

so that each intermediate artifact becomes both a supervision target and the input to the next generation stage.

Each transformation is represented as an instruction-following instance (I_s, x, y), where I_s specifies the stage, $x = a_{s-1}$ is the input artifact, and $y = a_s$ is the target artifact. This formulation supports two modes: stage-specific generation, where a user provides an intermediate artifact and asks for the next one, and end-to-end generation, where the model recursively generates a_1 through a_5 from an initial intent.

3.2 Artifact Chain

The artifact chain is designed to make the development process explicit while remaining compact enough for supervised fine-tuning.

Intent. The input intent is an informal natural-language description with a list of core functions. It captures what the user wants but may omit design decisions, edge cases, and behavioral constraints.

Software Requirements Specification (SRS). The SRS expands an informal intent into testable functional and non-functional requirements. It specifies *what* the system should do, including scope, user roles, core functions, interfaces, constraints, and open issues. This stage turns an under-specified prompt into a reviewable specification.

Detailed Design Specification (DDS). The DDS translates the SRS into an implementation-oriented design. It specifies *how* the system is organized, including architecture, modules, responsibilities, interfaces, classes, data structures, and key interactions. In our dataset, this specification is represented as a detailed design document, and it provides the structural bridge between requirements and behavior.

SCXML. The behavioral model represents control flow as State Chart XML (SCXML), a W3C standard for state-machine notation (Barnett et al., 2015). SCXML is useful for our setting because many examples involve event-driven transitions, guards, and stateful control logic. Compared with free-form text, it provides a machine-readable behavioral artifact that can be inspected independently of code syntax.

Pseudocode and Implementation. Pseudocode converts the declarative SCXML behavior into an imperative, language-neutral procedure with conditionals, loops, event handlers, and state updates. The final stage maps pseudocode to an executable implementation. This last transformation isolates language-specific syntax from earlier design and behavioral reasoning; in our experiments, the implementation target is Python.

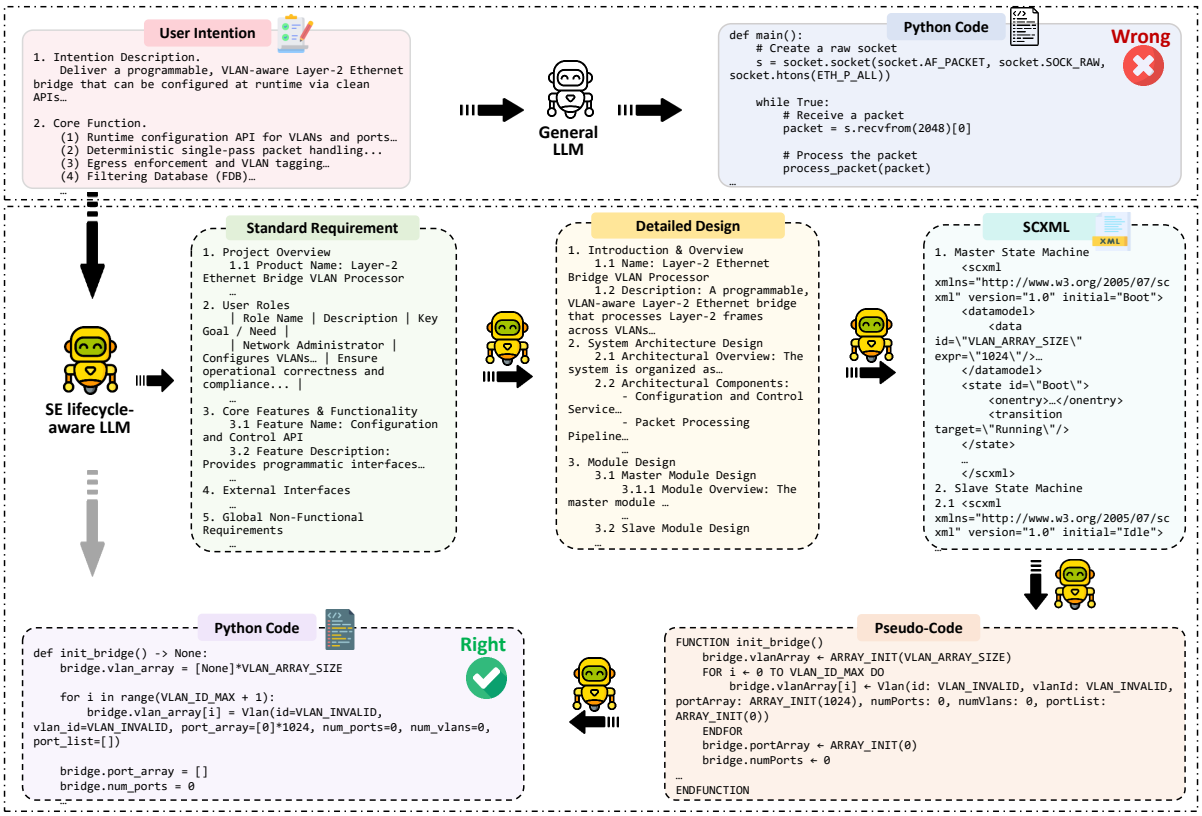

Figure 1: Overview of lifecycle-aware intermediate supervision. The upper path illustrates direct one-step generation from intent to code, while the lower path decomposes generation into supervised software-engineering artifacts: SRS, DDS, SCXML, pseudocode, and implementation.

3.3 Training Objective

Given a training set $\mathcal{D}$ of artifact chains, we expand each chain into five supervised instances. The model is trained to maximize the conditional likelihood of each target artifact:

$$\mathcal{L}(\theta) = -\sum_{(I_s, x, y) \in \mathcal{D}'} \log p_\theta(y \mid I_s, x), \qquad (2)$$

where $\mathcal{D}'$ is the stage-wise expanded dataset. The same backbone is used for all stages, so the model must learn both natural-language transformations, such as intent-to-SRS, and structured transformations, such as DDS-to-SCXML and pseudocode-to-implementation. In our implementation, fine-tuning is parameter-efficient using LoRA (Hu et al., 2022), but the formulation is independent of the adaptation method.

This objective differs from training only on final implementations. The model receives direct gradient signal for intermediate artifacts, and errors can be measured before they propagate to code. It also differs from prompt-only planning: the intermediate artifacts are not transient hidden reasoning traces, but supervised targets with stage-level references.

3.4 Multi-step Inference

At inference time, the model can be used in a sequential pipeline. Given a user intent a_0, it first generates an SRS $\hat{a}_1$, then conditions on $\hat{a}_1$ to generate $\hat{a}_2$, and continues until the final implementation $\hat{a}_5$ is produced:

$$\hat{a}_s = \arg\max_a p_\theta(a \mid I_s, \hat{a}_{s-1}). \qquad (3)$$

This procedure exposes intermediate artifacts that can be logged, evaluated, or revised by a human before continuing. It also enables partial workflows: for example, an engineer can provide a DDS and request an SCXML model, or provide pseudocode and request an implementation. In the experiments, we evaluate both stage-level artifact quality and final implementation quality to test whether improvements in intermediate supervision translate into executable correctness.

4 Dataset and Evaluation

4.1 Data Construction and Quality Control

We construct a state-machine-oriented dataset from three sources: OPNET/Riverbed Modeler,[1] XState,[2] and Simulink.[3] The corpus contains 457 complete artifact chains: 363 from OPNET, 47 from XState, and 47 from Simulink. The resulting corpus combines official examples, public project hubs, and collected state-machine-oriented source files from these ecosystems. These sources were selected because they contain event-driven or control-flow-heavy programs where behavioral modeling is meaningful. The original source identifiers include 410 C-family files and 47 Python files; in the supervised artifact chains, the final implementation field is represented as Python code to provide a consistent target language for evaluation.

The artifact chains are constructed in reverse from implementation to intent. Starting from code, GPT-5 is used to derive pseudocode, SCXML state-machine design, Detailed Design Specification (DDS), Software Requirements Specification (SRS), and user intent. This reverse construction is paired with fixed prompt templates for the forward transformations used during training. All GPT-5-assisted artifact data were manually quality-controlled by three computer-science researchers with doctoral-level training before being used for training or evaluation. Reviewers checked chain completeness, adjacent-stage semantic consistency, structured-artifact validity, and traceability from intent through implementation. Chains were discarded when they contained missing stages, malformed structured artifacts, ambiguous behavior, inconsistent requirements, pseudocode-code mismatch, or implementation details that could not be traced back to upstream artifacts. Disagreements were resolved through discussion, and only data accepted after this review process were retained. Thus, GPT-5 was used as a reconstruction assistant, while the final artifact chains were manually quality-controlled. Each retained example contains six aligned fields: intent, SRS, DDS, SCXML, pseudocode, and implementation.

[1] https://support.riverbed.com/content/support/software/opnet-model/modeler.html

[2] https://stately.ai/docs/examples; https://github.com/statelyai/xstate

[3] https://ww2.mathworks.cn/help/simulink/examples.html; https://github.com/mathworks/MATLAB-Simulink-Challenge-Project-Hub

4.2 Train/Test Split and Stage Expansion

We split the 457 complete chains into 366 training chains and 91 held-out test chains. To study data efficiency, we also create reduced training subsets containing 80%, 60%, 40%, and 20% of the training chains, corresponding to 293, 220, 146, and 73 chains, while keeping the same 91 test chains for all settings.

For supervised fine-tuning, each training chain is expanded into five adjacent transformation instances: intent-to-SRS, SRS-to-DDS, DDS-to-SCXML, SCXML-to-pseudocode, and pseudocode-to-implementation. Thus, the full training split yields $366 \times 5 = 1{,}830$ instruction-following instances. The held-out split remains at the chain level to avoid leakage across stages: all artifacts from a test chain are excluded from training.

4.3 Test-Function Evaluation

To evaluate final implementations beyond surface similarity, we construct test-function lists for each held-out instance. GPT-5 receives the user intent and reference implementation and generates candidate functional test scenarios. Each scenario contains four fields: an ID, a natural-language description, an expected input condition, and an expected output behavior. GPT-5 is then used in a filtering step to remove scenarios that expose implementation details or describe design internals rather than externally observable functionality. The same three-reviewer quality-control protocol was applied to the test-function lists. Reviewers removed scenarios that were redundant, underspecified, implementation-specific, inconsistent with the user intent, or not checkable from the generated implementation. The final filtered set contains 91 manually reviewed test-function lists, with 3 to 69 scenarios per instance and an average of 24.04 scenarios.

Given a generated implementation and the filtered test-function list, GPT-5 produces a coverage report indicating whether the implementation covers each functional scenario. We summarize this report as a test-function-based pass@K score, where K=5 follows the evaluation setting in the released result files. This metric is intended to approximate functional coverage of the generated implementation; it is not a substitute for executable unit tests, and we discuss this limitation explicitly.

We evaluate intermediate artifacts and final im-

plementations with stage-specific metrics. For SRS, DDS, and pseudocode, we report exact match (EM), BLEU (Papineni et al., 2002), and ROUGE-2 (Lin, 2004). For SCXML, we report EM, tree edit distance (TED) based on tree-edit operations (Zhang and Shasha, 1989), and TF-IDF similarity (Salton and Buckley, 1988), reflecting both structural and lexical agreement. For implementations, we report EM, BLEU, CodeBLEU (Ren et al., 2020), and test-function-based pass@K. We treat pass@K as the primary implementation-level metric because it directly targets intended behavior, while overlap metrics are retained for comparability with prior work.

Representative prompts, test-function evaluation prompts, data artifact details, and quality-control criteria are provided in Appendices Appendix B.–Appendix D..

5 Experiments

We organize the experiments around three parts: the comparison setting, the main evidence for lifecycle-aware supervision, and two analyses of data scale and artifact contribution. Complete stage-level tables are provided in Appendix A..

5.1 Experimental Setup

We evaluate four open backbones: Qwen3-1.7B, Qwen3-8B, DeepSeek-Coder-1.3B-Instruct, and DeepSeek-Coder-6.7B-Instruct, following recent open code-model families represented by Qwen3 and DeepSeek-Coder (Yang et al., 2025; Guo et al., 2024). For each backbone, we compare the original model with the model fine-tuned on lifecycle-expanded instances. Unless otherwise stated, "FT" denotes fine-tuning on the full 1,830-instance stage-expanded training set.

We also compare with five instruction/API baselines that are not fine-tuned on our data: DeepSeek-Chat, DeepSeek-Reasoner, GPT-3.5-Turbo, GPT-4o-mini, and LLaMA3-8B-Instruct. These baselines are prompted with the same evaluation inputs and are included to contextualize whether lifecycle supervision lets smaller open models approach stronger general-purpose systems. We treat test-function-based pass@K as the primary implementation metric and CodeBLEU as an auxiliary code-similarity metric.

5.2 Main Results

Lifecycle-aware fine-tuning improves final implementation quality for every open backbone. The

Model	CodeBLEU	pass@K	Δ
Qwen3-1.7B	0.255	0.193	+0.114
Qwen3-8B	0.294	0.355	+0.157
DS-Coder-1.3B	0.241	0.208	+0.139
DS-Coder-6.7B	0.285	0.335	+0.250
DeepSeek-Chat	0.326	0.327	–
DeepSeek-Reasoner	0.336	0.346	–
GPT-3.5-Turbo	0.283	0.306	–
GPT-4o-mini	0.192	0.104	–
LLaMA3-8B	0.237	0.107	–

Table 1: Implementation-stage results. Open backbones report FT scores and pass@K gains over base models; prompt baselines are not fine-tuned. K=5.

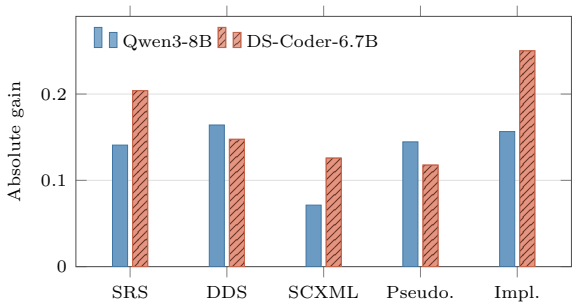

Figure 2: Absolute gains at each stage for two representative backbones.

largest absolute pass@K gain is observed for DeepSeek-Coder-6.7B (+0.250), while Qwen3-8B reaches the strongest fine-tuned pass@K score (0.355), comparable to DeepSeek-Reasoner (0.346). CodeBLEU and pass@K do not produce identical rankings, which reinforces our choice to treat test-function-based pass@K as the primary implementation metric and CodeBLEU as auxiliary.

The gains are not limited to final code. Figure 2 summarizes representative stage-level improvements for Qwen3-8B and DeepSeek-Coder-6.7B using each stage's primary metric: EM for SRS, DDS, and pseudocode, EM for SCXML, and pass@K for implementation. Both models improve throughout the artifact chain. The improvement is especially large at the implementation stage for DeepSeek-Coder-6.7B (+0.250 pass@K), while Qwen3-8B shows strong gains in both design-like stages and final implementation. Full metrics for all models, stages, and overlap scores are given in Appendix A..

5.3 Scaling and Ablation

Lifecycle supervision is data-efficient. As shown in Figure 3, Qwen3-8B improves sharply even with

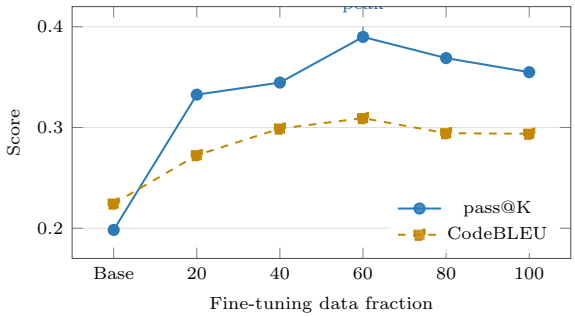

Figure 3: Qwen3-8B data scaling on the implementation stage.

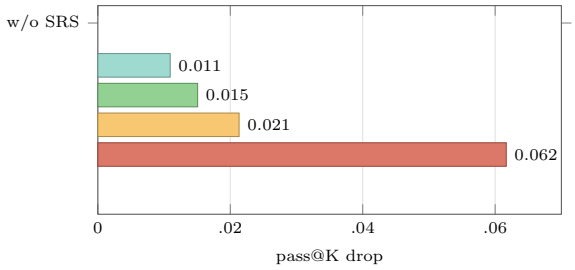

Figure 4: Implementation pass@K drop when removing one artifact type.

20% of the training chains, increasing implementation pass@K from 0.1983 to 0.3326. Final implementation quality peaks at a moderate supervision budget: pass@K reaches 0.3899 and CodeBLEU reaches 0.3093 at 60% of the training data. Increasing the fraction to 80% or 100% does not further improve functional correctness, suggesting that data quality and stage balance matter in addition to quantity.

Finally, we remove one lifecycle artifact type at a time for DeepSeek-Coder-6.7B and evaluate the final implementation stage. Figure 4 shows the pass@K drop for each removal setting, using the matched tuned baseline for that ablation. Removing the SCXML behavioral model causes the largest drop (0.062), followed by pseudocode (0.021). Removing DDS and SRS also weakens performance, but more modestly. This pattern indicates that upstream requirements and design preserve semantic grounding, while SCXML and pseudocode provide the behavioral and algorithmic constraints closest to executable correctness. The complete four-metric ablation figure is provided in Appendix A..

6 Analysis and Discussion

Why later artifacts matter. The ablation results suggest that the most execution-proximal artifacts provide the strongest constraints on final code quality. SRS and DDS establish semantic grounding and structural intent, but their influence is mediated by later artifacts: once an underspecified behavior or incorrect transition reaches the implementation stage, the model has little direct supervision for recovering it. SCXML and pseudocode are closer to executable behavior. SCXML makes states, events, guards, and transitions explicit, which is especially important for the state-machine-oriented programs in our benchmark. Pseudocode then converts this behavioral structure into an imperative procedure with control flow, updates, and branch conditions. Removing either stage weakens test-function-based pass@K more sharply than removing earlier text-heavy artifacts, indicating that lifecycle supervision is not merely adding more tokens during training; it is adding intermediate representations that constrain the form of executable reasoning.

General LLMs versus code-pretrained LLMs. The results also clarify how lifecycle supervision interacts with model pretraining. Code-pretrained backbones remain strong on implementation-oriented similarity metrics, consistent with their exposure to program syntax and code-like distributions. However, the best fine-tuned general backbone, Qwen3-8B, obtains the highest implementation pass@K among the fine-tuned models. This suggests that explicit lifecycle artifacts can partially compensate for weaker code-centric pretraining by making the reasoning path observable and learnable. In other words, code pretraining helps models produce code-shaped outputs, while lifecycle supervision helps models preserve requirements, behavior, and algorithmic intent across stages. The two sources of signal are therefore complementary rather than interchangeable.

Failure cases. The remaining failures are likely concentrated where the artifact chain amplifies, rather than repairs, earlier mistakes. If the SRS omits an edge case, later stages may produce internally consistent but incomplete behavior. If SCXML contains missing transitions, incorrect guards, or overly coarse states, pseudocode can faithfully translate the wrong behavior into plausible implementation logic. Conversely, if pseudocode is too abstract, the final model may fill in

details with syntactically valid but semantically unsupported code. These cases explain why pass@K and CodeBLEU do not always rank systems identically: a generated implementation can look close to the reference while failing a functional scenario, or use different surface structure while covering the intended behavior. Future versions of the framework should therefore expose intermediate artifacts to targeted verification, for example by checking SCXML transition completeness or validating pseudocode against test-function scenarios before code generation.

Limitations

Although the GPT-5-assisted artifact chains and test-function lists are manually quality-controlled, our test-function-based pass@K metric still relies on LLM-based coverage judgments, so it should be interpreted as an approximate functional-coverage measure rather than executable unit-test pass@K over fully specified environments. Our experiments compare lifecycle-expanded fine-tuning against base models and prompted baselines, while an explicitly matched final-code-only SFT control remains future work. The evaluation also focuses on state-machine-oriented and control-flow-heavy programs, so the same artifact chain may be less useful for domains where SCXML-style behavioral modeling is unnatural. Finally, we evaluate one training split and a limited set of open backbones, leaving multi-seed confidence intervals and larger model families for future work.

Ethics Statement

The dataset is derived from software artifacts rather than human-subject data. Generated implementations may still contain bugs, insecure behavior, or licensing-sensitive fragments if deployed without review. Any generated code should receive human inspection and security review before deployment. We therefore present the method as a supervised research framework for code-generation evaluation, not as a guarantee of production-ready software.

References

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

Jim Barnett, Rahul Akolkar, Robert J. Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Michael Helbing, Rafah Hosn, T. V. Raman, Klaus Reifenrath, Noam Rosenthal, and Johan Roxendal. 2015. State chart XML (SCXML): State machine notation for control abstraction. W3C Recommendation.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Xinyi He, Jiaru Zou, Yun Lin, Mengyu Zhou, Shi Han, Zejian Yuan, and Dongmei Zhang. 2024. CoCoST: Automatic complex code generation with online searching and correctness testing. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu,

and Jürgen Schmidhuber. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):182:1–182:30.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. *arXiv preprint arXiv:2309.06180*.

Yizhi Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, pages 18319–18345.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023b. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81.

Feng Lin, Dong Jae Kim, and Tse-Hsun Chen. 2025. SOEN-101: Code generation by emulating software process models using large language model agents. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. StarCoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering code large language models with Evol-Instruct. In *International Conference on Learning Representations*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NExT: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. In *Proceedings of the 2020 Conference on Empirical*

Methods in Natural Language Processing: Findings, pages 1399–1407.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning in natural language improves LLM search for code generation. *arXiv preprint arXiv:2409.03733*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464.

Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. 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)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Appendix A. Full Experimental Results

This appendix reports the complete stage-level metrics corresponding to the summarized results in Section 5.

Stage	Metric	Qwen1.7		Qwen8		DSC1.3		DSC6.7	
		base	ft	base	ft	base	ft	base	ft
SRS	EM	0.4639	0.5936	0.4952	**0.6360**	0.4267	0.5949	0.4228	0.6268
	BLEU	0.0852	0.1713	0.0915	0.2220	0.0559	0.1948	0.0657	**0.2281**
	ROUGE2	0.4827	0.3494	0.4808	0.3909	0.3739	0.3951	0.4218	0.3996
DDS	EM	0.2930	0.4397	0.3253	**0.4895**	0.3159	0.4363	0.3346	0.4822
	BLEU	0.0153	0.0666	0.0256	0.1022	0.0310	0.0804	0.0322	0.1045
	ROUGE2	0.3267	0.2650	0.2984	0.2510	0.2273	0.2753	0.2499	0.2425
SCXML	EM	0.1646	0.2148	0.1854	0.2567	0.1089	0.2371	0.1439	**0.2697**
	TED	0.0973	0.1235	0.1170	0.1202	0.0995	0.1182	0.0932	0.1231
	TF-IDF	0.2748	0.3166	0.3049	0.3399	0.2750	0.3360	0.2560	0.3491
Pseudocode	EM	0.0801	0.1841	0.1364	**0.2810**	0.1060	0.2240	0.1395	0.2572
	BLEU	0.0007	0.0068	0.0015	0.0190	0.0011	0.0096	0.0013	0.0149
	ROUGE2	0.0266	0.0741	0.0390	0.1064	0.0352	0.1023	0.0364	0.1275
Implementation	EM	0.0678	0.1673	0.0845	0.2232	0.0801	0.1514	0.0703	0.1911
	BLEU	0.0004	0.0084	0.0010	0.0172	0.0009	0.0054	0.0010	0.0129
	CodeBLEU	0.2144	0.2547	0.2244	**0.2938**	0.2210	0.2411	0.2042	0.2852
	pass@K	0.0790	0.1934	0.1983	**0.3548**	0.0691	0.2082	0.0849	0.3351

Table A.1: Full five-stage results before and after lifecycle-aware fine-tuning. Best values for the key comparison rows are bolded. DSC denotes DeepSeek-Coder.

Stage	Metric	DS Chat	DS Reasoner	GPT 3.5	GPT 4o-mini	LLaMA3 8B
SRS	EM	**0.56969**	0.56335	0.53334	0.46955	0.48891
	BLEU	0.14178	0.13246	0.15386	0.10093	0.09374
	ROUGE2	0.42806	0.42484	0.43878	0.44417	0.45529
DDS	EM	0.37438	0.37297	**0.38450**	0.31526	0.33800
	BLEU	0.04442	0.04351	0.05075	0.02534	0.03061
	ROUGE2	0.24553	0.24804	0.26153	0.28155	0.25204
SCXML	EM	0.21132	0.21176	0.21407	0.17658	0.14940
	TED	0.10156	0.10233	0.10532	0.10564	0.09253
	TF-IDF	0.33420	0.33442	0.30105	0.26607	0.28178
Pseudocode	EM	0.17305	0.18294	0.13920	0.14366	0.14648
	BLEU	0.00359	0.00336	0.00219	0.00134	0.00280
	ROUGE2	0.03991	0.03785	0.03059	0.02199	0.02599
Implementation	EM	0.14488	0.14496	0.12802	0.08261	0.09686
	BLEU	0.00674	0.00535	0.00267	0.00074	0.00238
	CodeBLEU	0.32595	**0.33625**	0.28306	0.19239	0.23681
	pass@K	0.32725	**0.34627**	0.30555	0.10403	0.10672

Table A.2: Full five-stage results for prompt baselines without lifecycle fine-tuning. Best values for the key comparison rows are bolded. DS denotes DeepSeek.

Stage	Metric	Base	Fine-tuning Data Fraction				
			20%	40%	60%	80%	100%
SRS	EM	0.4952	0.6296	0.6357	0.6378	**0.6410**	0.6361
	BLEU	0.0916	0.2264	0.2322	0.2312	0.2363	0.2221
	ROUGE2	0.4808	0.4002	0.4021	0.4101	0.4147	0.3910
DDS	EM	0.3254	0.4737	0.4877	0.4838	0.4968	0.4895
	BLEU	0.0256	0.1012	0.1090	0.1058	0.1124	0.1022
	ROUGE2	0.2985	0.2407	0.2472	0.2473	0.2554	0.2511
SCXML	EM	0.1855	0.2580	0.2632	0.2840	0.2745	0.2568
	TED	0.1170	0.1177	0.1330	0.1220	0.1269	0.1202
	TF-IDF	0.3049	0.3289	0.3421	0.3327	**0.3690**	0.3399
Pseudocode	EM	0.1364	0.2627	0.2709	0.2744	0.2658	**0.2810**
	BLEU	0.0016	0.0140	0.0163	0.0195	0.0169	0.0191
	ROUGE2	0.0391	0.0760	0.0985	0.1032	0.1143	0.1065
Implementation	EM	0.0846	0.1882	0.2097	0.2131	0.2046	0.2232
	BLEU	0.0010	0.0086	0.0146	0.0159	0.0134	0.0173
	CodeBLEU	0.2244	0.2725	0.2989	**0.3093**	0.2943	0.2938
	pass@K	0.1983	0.3326	0.3446	**0.3899**	0.3689	0.3549

Table A.3: Full five-stage data-scaling results for Qwen3-8B. Best values for the key comparison rows are bolded.

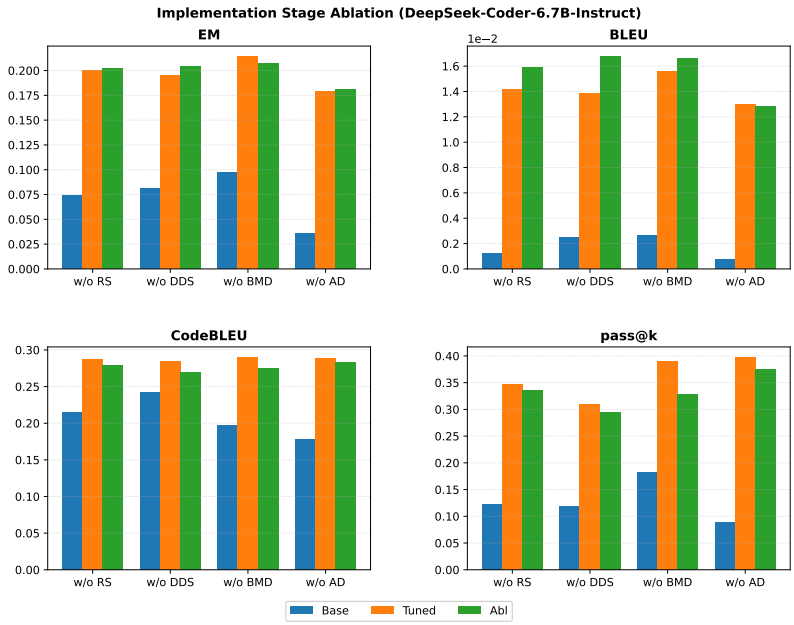

Figure A.1: Full implementation-stage ablation on DeepSeek-Coder-6.7B-Instruct. Original labels map to our terminology as RS=SRS, DDS=DDS, BMD=SCXML behavioral model, and AD=Pseudocode. The original plotted pass@k uses K=5 in our test-function-based evaluation.

Appendix B. Prompt Templates

This appendix summarizes representative prompt skeletons used to construct and evaluate lifecycle artifacts. The full prompt file contains the same stage structure with longer formatting instructions. We omit instance-specific inputs and use placeholders for artifact contents.

Forward artifact generation. The forward prompts follow the adjacent-stage transformations used for supervised fine-tuning:

```
Role: experienced product manager /
system architect / software engineer.
Task: transform the input artifact into
the next lifecycle artifact.
Input: <previous artifact content>.
Output: follow the target document
template exactly; be concise; avoid
irrelevant content.
```

For intent-to-SRS, the target template contains project overview, user roles, core features and functionality, external interfaces, global non-functional requirements, and open issues. For SRS-to-DDS, the target template contains introduction and overview, system architecture design, and module design. For DDS-to-SCXML, the prompt asks for a master state machine and optional slave state machines in SCXML. For SCXML-to-pseudocode, the prompt requires structured pseudocode with explicit conditionals, loops, functions, returns, comments, and data structures. For pseudocode-to-implementation, the prompt asks for Python code that strictly follows the provided pseudocode.

Reverse artifact reconstruction. The GPT-5-assisted reconstruction prompts use the reverse direction:

```
code → pseudocode → SCXML → DDS →
SRS → intent.
```

Each reverse prompt mirrors the corresponding forward output template. For example, code-to-pseudocode asks an experienced software engineer to summarize Python code into the pseudocode conventions, while SRS-to-intent asks an experienced product manager to produce a concise intention description and core-function list without exposing downstream SCXML details.

Appendix C. Test-Function Evaluation Prompts

Test-function list generation. Given a user intention and Python implementation, GPT-5 is prompted as an experienced software testing engineer to extract functional test points:

```
Input: user intention; Python code.
Workflow: analyze intent, analyze code
logic, combine both sources.
Output: JSON array with fields ID,
Description, Except Input, Except
Output.
ID format: TF-XX.
Description format: When <condition>,
the system could/couldn't <action>.
```

Test-function filtering. The filtering prompt receives one JSON object containing `code_file`, `user_intention`, `code`, and `test_list`. It asks GPT-5 to remove test-function scenarios that describe internal implementation details or system-design internals rather than externally observable behavior, then renumber the retained items in the original JSON format.

Coverage judgment. For implementation evaluation, GPT-5 receives generated Python code and the filtered test-function list:

```
Input: Python code; filtered
test-function list.
Task: decide whether the code covers
each test function or a similar
function.
Output: JSON object with test_report.
Each report item contains ID and Cover,
where Cover is YES or NO.
```

The test-function-based pass@K score is computed from these coverage reports using K=5, consistent with the released result files. This is an approximate functional-coverage metric rather than executable unit-test pass@K.

Appendix D. Data Quality Control and Artifact Details

Data artifacts. The local artifact files contain four JSON datasets: `all data` with 457 complete chains, `training data` with 1,830 stage-expanded instruction instances, `test data` with 91 held-out chains, and `filtered test-function lists` with 91 reviewed lists. The complete-chain fields are user intent, SRS, DDS, SCXML, pseudocode, implementation, and source file identifier. The stage-expanded training fields are instruction, input, and output.

Dataset statistics. The corpus contains 457 complete artifact chains: 363 from OPNET/Riverbed Modeler, 47 from XState, and 47 from Simulink. The 366 training chains are expanded into 1,830

adjacent-stage instruction instances, and the 91 held-out chains are kept at the chain level. The filtered test-function lists contain 3 to 69 scenarios per instance, with a mean of 24.04 and median of 22.

Artifact terms and intended use. The source ecosystems have different access and redistribution terms. The released artifact is intended for research and evaluation use, retains source identifiers where applicable, and should be used in accordance with the original source terms. Generated implementations should not be treated as production-ready software without human inspection and security review.

Quality control. All GPT-5-assisted artifact chains and test-function lists were manually quality-controlled by three computer-science researchers with doctoral-level training. For artifact chains, reviewers checked chain completeness, adjacent-stage semantic consistency, structured-artifact validity, and traceability from intent through implementation. Reviewers also checked for personally identifying or offensive content; the retained artifacts are software-centric and do not contain human-subject records. Chains were discarded for missing stages, malformed structured artifacts, ambiguous behavior, inconsistent requirements, pseudocode-code mismatch, or implementation details not traceable to upstream artifacts. For test-function lists, reviewers removed redundant, underspecified, implementation-specific, intent-inconsistent, or uncheckable scenarios. Disagreements were resolved through discussion, and only data accepted after this process were retained.

Training and inference setup. Fine-tuning was conducted with LLaMA-Factory (Zheng et al., 2024) using its default supervised fine-tuning configuration for the selected backbones, without manual hyperparameter tuning. Inference for generated candidates was served with vLLM (Kwon et al., 2023). Experiments were run on NVIDIA A100 GPUs, and fine-tuning took approximately four hours per backbone. The current artifact files specify the data construction process, model families, evaluation metrics, result tables, prompt templates, and evaluation prompts, but do not include an exported run configuration table with exact LoRA rank, learning rate, batch size, number of epochs, sequence length, or aggregate GPU-hour logs. We therefore report the tool-level setup here and leave the exact run-configuration table for the released artifact package.