REARCH-CODER: REQUIREMENT ALIGNMENT AND ARCHITECTURE DESIGN FOR RELIABLE REPOSITORY-LEVEL CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Open-source language models offer low cost and flexible deployment, making them attractive for practical coding agents. Yet they still struggle to generate executable repositories reliably from requirement documents. Existing approaches use harness engineering to organize repository generation, with closed-source frontier models serving as the underlying base models. However, applying these approaches to open-source models exposes two limitations: (1) Models may misinterpret requirements and implement functionality that departs from user needs. (2) Their limited architecture design capabilities can lead to flawed repository designs that prevent modules from working together. To address these challenges, we propose ReArch-Coder, a method for requirement alignment and repository architecture design optimization in open-source models. First, ReArch-Coder aligns the target model's understanding of the requirement document through diagnostic questions and answers and factual hints. Then, it formulates architecture design as a trainable agentic decision-making process and optimizes it using GRPO with rewards based on requirement coverage and dependency modularity. Experimental results on representative repository-level code generation benchmarks demonstrate the effectiveness of our method. For example, ReArch-Coder outperforms the strongest coding-agent baseline on RepoGenesis by 21.8 percentage points in Passed and 19.6 percentage points in Resolved. The code is anonymously available at `https://zenodo.org/records/22974177`.

1 INTRODUCTION

Large language models have made substantial progress in generating functions and individual files from natural-language descriptions Rozi'ere et al. (2023); Hui et al. (2024). However, real-world software development requires more than producing isolated code fragments. Coding agents must generate a coherent software system from high-level requirements, ensuring that its architecture and implementation jointly realize the intended functionality. Therefore, repository-level code generation extends code synthesis toward end-to-end software construction, where an agent builds a complete and executable repository from a requirement document and an empty workspace Peng et al. (2026); Ding et al. (2026); Luo et al. (2026). This capability represents an important step toward turning user requirements directly into deployable software and realizing more autonomous software development.

Existing research has developed various inference-time workflows and structured contexts to improve repository generation with closed-source frontier models. Multi-agent frameworks such as ChatDev Qian et al. (2024) and MetaGPT Hong et al. (2024) organize software development through specialized roles and collaboration, while Paper2Code Seo et al. (2026), HiRAS Hong et al. (2026), and AutoReproduce Zhao et al. (2026) decompose repository construction into stages of planning, implementation, and validation. Other methods provide explicit intermediate representations to guide generation. CodeS Zan et al. (2024) progressively refines natural-language requirements into repository structures, file sketches, and function implementations, while RPG Luo et al. (2026) represents capabilities, files, interfaces, and data flows in a unified planning graph. SpecFirst Chen et al. (2026) elicits a structured behavioral specification before code synthesis, and CodeSpec Wang

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.

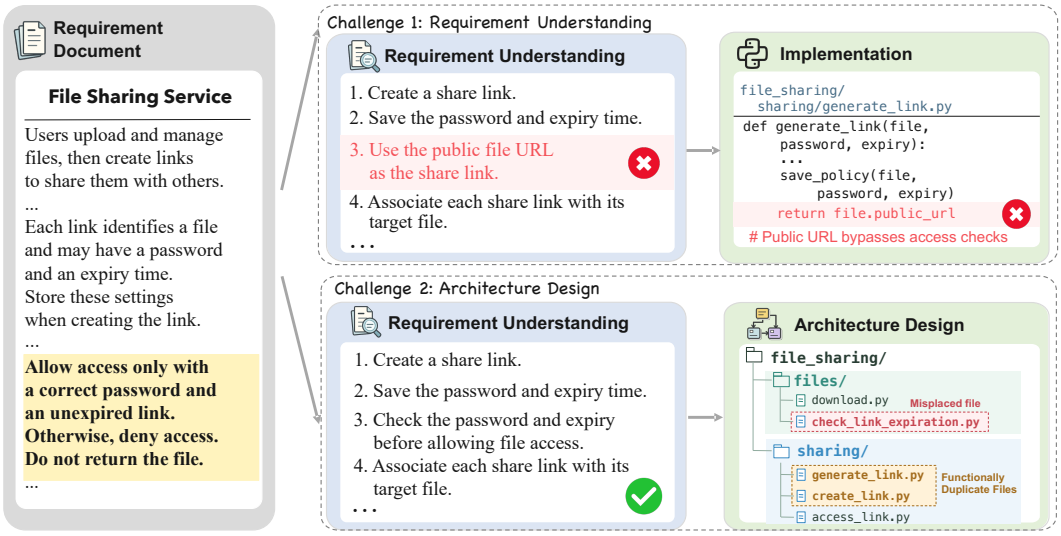


Figure 1: Two challenges faced by open-source models in repository-level code generation include (1) requirement misunderstandings and (2) architecture design flaws.

et al. (2026) constructs executable architecture and behavior specifications to maintain consistency throughout repository-level development.

Open-source language models offer low cost and flexible deployment, making them an important option for practical coding agents, particularly when code must remain within private environments. However, applying existing harnesses to these models still presents two limitations, as illustrated in Figure 1. (1) Models may misinterpret requirement documents, causing the implemented functionality to deviate from user requirements. For example, in a file-sharing service, a model incorrectly interprets the requirement to allow file access only after password and expiry validation as merely storing these settings and returning the public file URL. (2) Limited architecture design ability can produce repository architectures with poorly assigned responsibilities even when requirements are correctly understood, complicating subsequent implementation and integration. For example, a model incorrectly places link expiry checks, which belong to share link management, in the file management directory and designs two duplicate files for the same link creation functionality.

To address these challenges, we propose ReArch-Coder, a method for requirement alignment and repository architecture design optimization in open-source models. First, ReArch-Coder uses diagnostic questions and answers to identify requirement misunderstandings, generates factual hints, and inserts verified hints into the original document to produce an aligned requirement document. Then, ReArch-Coder formulates repository architecture design as an agentic decision-making process in which the model constructs an initial architecture from the requirements and revises it using diagnostic feedback. ReArch-Coder computes rewards from the final architecture's requirement coverage and dependency modularity and uses GRPO to optimize decisions throughout the design trajectory.

We evaluate ReArch-Coder on RepoGenesis Peng et al. (2026) and NL2Repo-Bench Ding et al. (2026), two benchmarks for generating executable repositories from natural-language requirements. With Qwen3-Coder-30B under Codex, ReArch-Coder improves functional test pass rates over the strongest evaluated baselines by 21.8 and 13.9 percentage points, respectively. Ablation studies further confirm that both requirement alignment and GRPO-based architecture design contribute to the improvement. These results demonstrate that correcting requirement misunderstandings and optimizing repository architecture design can improve the functional correctness of generated repositories.

Our contributions are summarized as follows:

- We introduce requirement alignment as a diagnostic process that identifies errors in a target model's understanding and corrects them using factual reminders verified against the original requirements.

- We formulate repository architecture design as an agentic decision-making process and use programmatically computable rewards with GRPO to directly optimize the model's architecture design capability.
- We evaluate ReArch-Coder on RepoGenesis and NL2Repo-Bench, demonstrating consistent improvements in functional test pass rates and validating the contribution of both components through ablation studies.

2 Related Work

2.1 Code LLMs and Coding Agents

Open code models such as StarCoder Li et al. (2023), Code Llama Rozi'ere et al. (2023), DeepSeek-Coder Guo et al. (2024), StarCoder2 Lozhkov et al. (2024), and Qwen2.5-Coder Hui et al. (2024) improve code synthesis through code-centric pretraining, long-context modeling, instruction tuning, and execution feedback. Repository tasks nevertheless require an agent loop around the model. CodeAgent Zhang et al. (2024) integrates tools for repository-level tasks. SWE-Agent Yang et al. (2024) designs an agent–computer interface for modifying existing projects, and OpenHands Wang et al. (2025) provides a general platform for software-development agents. Our setting differs from repository repair because no implementation context exists initially, so architecture and interfaces must be derived from the requirement document.

2.2 Repository-Level Code Generation

Existing methods support repository generation through development workflows and intermediate representations. ChatDev Qian et al. (2024) and MetaGPT Hong et al. (2024) organize multi-agent collaboration, while Paper2Code Seo et al. (2026), HiRAS Hong et al. (2026), and AutoReproduce Zhao et al. (2026) use staged or hierarchical planning for paper reproduction. CodeS Zan et al. (2024) progressively refines requirements into code, and RPG Luo et al. (2026) represents repository structure through a planning graph. SpecFirst Chen et al. (2026) and CodeSpec Wang et al. (2026) guide generation with explicit specifications. However, these methods remain constrained by the base model's requirement understanding and architecture design abilities, whereas ReArch-Coder corrects requirement misunderstandings with verified hints and optimizes architecture design through reinforcement learning.

3 Method

In this section, we propose ReArch-Coder, a repository-level code generation method for open-source models. As illustrated in Figure 2, ReArch-Coder consists of two modules. Requirement Alignment corrects the target model's misunderstandings of the requirement document. Repository Architecture Design Optimization optimizes the model's architecture design ability through GRPO.

3.1 Requirement Alignment

Requirement alignment aims to identify and correct the target model's misunderstandings before repository architecture design. It consists of four components, ❶ constructing an evidence-grounded question checklist, ❷ diagnosing requirement misunderstandings from the target model's answers, ❸ generating factual hints for incorrect answers, and ❹ verifying hints through masked recovery and inserting each verified hint after the corresponding requirement passage.

3.1.1 Evidence-Grounded Question Checklist Construction

To make the target model's requirement understanding explicitly testable, we define a set of requirement dimensions based on prior research in requirements engineering (Glinz, 2000; Greenspan et al., 1994) and the IEEE recommended practice for software requirements specifications (IEEE, 1998). These dimensions guide the construction of diagnostic questions. The full list and definitions are provided in the Appendix A

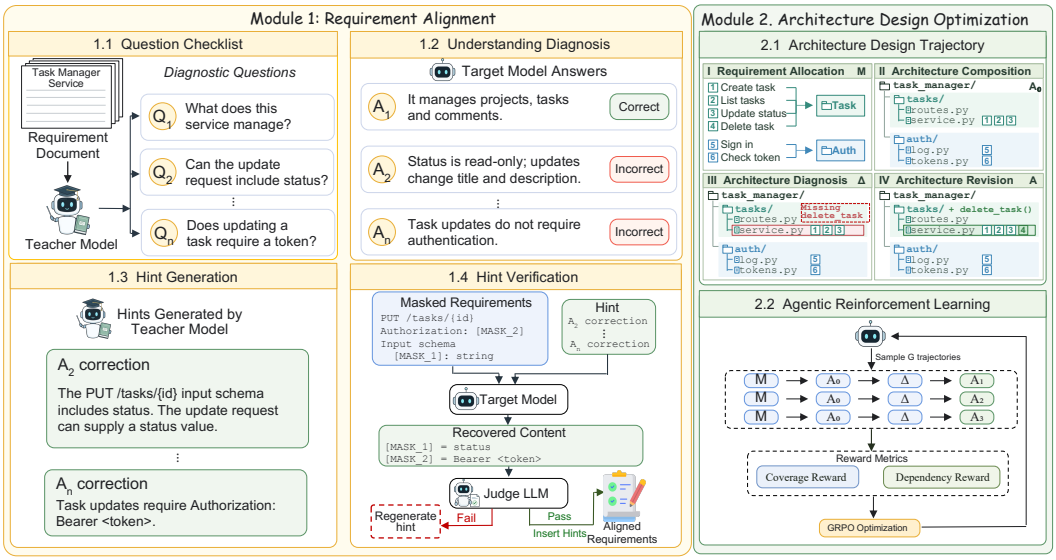


Figure 2: Overview of ReArch-Coder, which consists of two modules for requirement alignment and architecture design optimization.

Given a requirement document, the teacher model analyzes the text along these dimensions to construct a question checklist with reference answers, retaining the relevant document excerpt for each question. We include a small set of manually annotated question-answer examples in the prompt to demonstrate how to formulate independently answerable questions about individual requirements and derive answers from the original text.

3.1.2 Requirement Understanding Diagnosis

The target model answers the questions in the checklist using the original requirement document. The teacher model compares these answers with the reference answers to determine whether the target model correctly understands the corresponding requirements. Correctly answered questions do not proceed to hint generation. For incorrectly answered questions, we record the question, the reference answer, and the target model's answer for factual hint generation.

3.1.3 Factual Hint Generation

The teacher model uses the questions, reference answers, and incorrect target model answers recorded in the previous stage to generate factual hints grounded in the corresponding document excerpts. For misinterpreted content, the hints clarify the meaning specified in the original requirements. For omitted content, the hints supply requirement facts missing from the answers. Each hint contains only information supported by the original requirements and is expressed as a concise, self-contained statement.

3.1.4 Hint Verification

For each factual hint, ReArch-Coder masks the semantic units expressing the requirement facts addressed by the hint in the corresponding document excerpt. The target model then recovers the masked content using the remaining local context and the hint.

We use an additional teacher model as a judge to compare the recovered content with the original text. A hint passes verification when the recovered content accurately expresses the corresponding requirement without introducing additional information. Verified hints are inserted after the corresponding requirements in the original document. Hints that fail verification are regenerated based on discrepancies between the recovered content and the original text until they pass verification or reach the maximum number of attempts.

3.2 Repository Architecture Design Optimization

3.2.1 Architecture Design Trajectory

Given an aligned requirement document, ReArch-Coder decomposes repository architecture design into four consecutive steps. Each step produces a structured artifact that guides subsequent design decisions. The steps are as follows:

Step I. Requirement Allocation. The model assigns each requirement to a functional module and specifies its responsibility, producing a requirement-to-module mapping M.

Step II. Architecture Composition. Based on M, the model assigns module responsibilities to directories and files, specifying file responsibilities, imports, and function or class signatures. It also establishes requirement-to-file assignments, producing an initial repository architecture $A^{(0)}$.

Step III. Architecture Diagnosis. The model compares the aligned requirements with $A^{(0)}$ to examine the completeness of requirement coverage and the appropriateness of architecture organization. Identified issues are summarized in a diagnostic report Δ.

Step IV. Architecture Revision. The model revises $A^{(0)}$ according to Δ, adding designs for uncovered requirements and adjusting inappropriate architecture organization to produce the final repository architecture A.

3.2.2 Reinforcement Learning for Architecture Design

ReArch-Coder uses the final repository architecture of each design trajectory to evaluate the agent's decisions and optimize its architecture design policy. The training process consists of architecture reward design and GRPO optimization.

Architecture Reward Design. The architecture reward combines requirement coverage and dependency organization, evaluating requirement-to-file assignments and the alignment between file dependencies and directory grouping.

The requirement coverage reward measures the proportion of requirements assigned to at least one file in the final architecture. Let $\mathcal{R}$ denote the requirement set, F the files in the final architecture A, and Γ the requirement-to-file assignments, where $(r, f) \in \Gamma$ indicates that requirement r is assigned to file f. The reward is defined as follows:

$$R_{\text{cov}}(A) = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \mathbb{I}\big[\exists f \in F,\ (r, f) \in \Gamma\big]. \tag{1}$$

Following prior work (Newman & Girvan, 2004), the dependency organization reward uses modularity to measure how concentrated file dependencies are within directories, serving as a structural proxy for high cohesion and low coupling. For each directory c, L_c denotes the number of dependencies within the directory, d_c denotes the sum of the degrees of its files, and m denotes the total number of dependencies between files in the repository. Calculation details are provided in the Appendix D. When $m = 0$, the reward is set to 0. Otherwise, it is computed as follows:

$$R_{\text{dep}}(A) = \sum_{c} \left[\frac{L_c}{m} - \left(\frac{d_c}{2m} \right)^2 \right]. \tag{2}$$

The final reward combines the two components computed from the revised architecture, with weights λ_{cov} and λ_{dep} controlling their respective contributions,

$$R(A) = \lambda_{\text{cov}} R_{\text{cov}}(A) + \lambda_{\text{dep}} R_{\text{dep}}(A). \tag{3}$$

GRPO Optimization. For each aligned requirement document, ReArch-Coder samples G architecture design trajectories $\{\tau_i\}_{i=1}^{G}$ from the old policy. Each trajectory produces a final architecture A_i with reward $R_i = R(A_i)$. Its group-relative advantage is computed as $\widehat{g}_i = (R_i - \mu_R)/(\sigma_R + \epsilon)$, where μ_R and σ_R are the mean and standard deviation of the group rewards. ReArch-Coder opti-

mizes the design policy using the standard GRPO objective Shao et al. (2024):

$$\mathcal{J}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\mathbb{E}_{t\sim\tau_i}\left[\min\left(\rho_{i,t}\widehat{g}_i,\ \mathrm{clip}(\rho_{i,t}, 1-\varepsilon, 1+\varepsilon)\widehat{g}_i\right) - \beta D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}})\right], \tag{4}$$

where $\rho_{i,t} = \pi_\theta(a_{i,t} \mid s_{i,t})/\pi_{\theta_{\mathrm{old}}}(a_{i,t} \mid s_{i,t})$. The trajectory-level advantage is applied to all model-generated actions in the corresponding architecture design trajectory.

4 Experiments

4.1 Experimental Setup

Datasets. We use NL2Repo-Bench Ding et al. (2026) and RepoGenesis Peng et al. (2026), which contain 104 Python library generation tasks and 106 Python and Java microservice generation tasks, respectively. We split each dataset into disjoint training and test sets at the project level. We use 52 projects from NL2Repo-Bench and 53 from RepoGenesis to train the model for repository architecture design, reserving the remaining 52 and 53 projects for evaluation. The number of architecture design trajectories and the sampling settings are provided in Appendix E.

Evaluation Metrics. Following prior works Ding et al. (2026); Peng et al. (2026), we report **%Passed** and **%Resolved**. **%Passed** is the average percentage of test cases passed by each generated repository, while **%Resolved** is the percentage of evaluated repositories that pass all test cases. Resolved is computed over the 52 test projects in NL2Repo-Bench and the 53 test projects in RepoGenesis.

Models and Baselines. We evaluate Qwen3-Coder-30B-Instruct, Qwen3-30B-Instruct, and CodeLlama-Instruct-34B using four agent frameworks, SWE-Agent Yang et al. (2024), OpenHands Wang et al. (2025), Qwen-Agent [1], and Codex [2]. Within each agent framework, the baseline uses the original model to generate repositories from the requirement documents. ReArch-Coder first aligns the requirements, then uses the trained model to design the repository architecture and generate code. We additionally evaluate RPG Luo et al. (2026) and Paper2Code Seo et al. (2026) using the same backbone models. Specifically, RPG guides repository generation through a repository planning graph, while Paper2Code organizes generation into planning, analysis, and implementation stages.

Implementation Details. For requirement alignment, we use GPT-5.5 to generate diagnostic questions, reference answers, and factual hints, and Claude-Opus-4.6 for requirement diagnosis and hint verification. We generate at most 20 questions per requirement document and allow up to 3 verification attempts per hint. For architecture design optimization, we set the reward weights to $\lambda_{\mathrm{cov}} = 0.7$ and $\lambda_{\mathrm{dep}} = 0.3$. Training is performed on 8 NVIDIA H100 (80GB) GPUs using verl Sheng et al. (2024). Each task runs in an isolated Docker container with a time limit of 1 hour. Additional implementation details are provided in Appendix E.

4.2 Main Results

Table 1 presents the evaluation results of ReArch-Coder and the baselines on the two benchmarks. We observe that ReArch-Coder consistently improves performance across all agent and base model combinations. For example, with Qwen3-Coder-30B under the Codex framework, ReArch-Coder increases the Passed rate on NL2Repo-Bench from 13.5 % to 27.4% and on RepoGenesis from 23.4% to 45.2%. The Resolved rate also increases from 5.8% to 11.0% and from 23.2% to 42.8% on the two benchmarks, respectively. Qwen3-30B and CodeLlama-34B also achieve performance improvements, showing that the benefits of requirement alignment and architecture design generalize across different base models.

In addition, we evaluate RPG and Paper2Code on open-source models. We observe that our approach outperforms these agents in both Passed and Resolved rates. The results also reveal differences across agent frameworks. For example, Codex achieves the highest performance among the

[1] https://github.com/QwenLM/Qwen-Agent

[2] https://github.com/openai/codex

Table 1: Performance of different coding agents using base models and their corresponding ReArch-Coder-trained versions on NL2Repo-Bench and RepoGenesis. Both metrics are percentages, reported as mean ± sample standard deviation. Δ denotes gains in percentage points over the corresponding base models, computed from the displayed means.

Agent	Model	Method	NL2Repo-Bench				RepoGenesis			
			Passed (%) ↑	Δ	Resolved (%) ↑	Δ	Passed (%) ↑	Δ	Resolved (%) ↑	Δ
SWE-Agent	Qwen3-Coder-30B	Base	4.1 ± 1.2	-	0.6 ± 1.1	-	6.2 ± 1.8	-	2.5 ± 2.3	-
		ReArch-Coder	**5.3** ± 1.4	+1.2	**1.9** ± 1.0	+1.3	**8.3** ± 2.1	+2.1	**5.1** ± 2.8	+2.6
	Qwen3-30B	Base	3.2 ± 0.9	-	0.6 ± 1.1	-	4.8 ± 1.3	-	1.9 ± 1.0	-
		ReArch-Coder	**4.6** ± 1.3	+1.4	**1.3** ± 1.1	+0.7	**6.5** ± 1.6	+1.7	**3.8** ± 1.2	+1.9
	CodeLlama-34B	Base	2.1 ± 0.7	-	0.0 ± 0.0	-	3.5 ± 1.1	-	1.3 ± 1.1	-
		ReArch-Coder	**3.0** ± 1.0	+0.9	**0.6** ± 1.1	+0.6	**5.1** ± 1.3	+1.6	**2.5** ± 0.9	+1.2
OpenHands	Qwen3-Coder-30B	Base	10.5 ± 2.5	-	3.3 ± 2.9	-	18.5 ± 3.8	-	15.7 ± 4.0	-
		ReArch-Coder	**23.8** ± 3.1	+13.3	**9.0** ± 4.0	+5.7	**33.2** ± 4.1	+14.7	**27.0** ± 4.7	+11.3
	Qwen3-30B	Base	8.3 ± 2.2	-	2.5 ± 1.3	-	16.2 ± 3.1	-	12.6 ± 4.0	-
		ReArch-Coder	**20.1** ± 2.8	+11.8	**5.8** ± 3.3	+3.3	**31.4** ± 3.5	+15.2	**22.6** ± 4.9	+10.0
	CodeLlama-34B	Base	6.4 ± 2.1	-	1.9 ± 0.9	-	13.4 ± 2.6	-	10.8 ± 2.8	-
		ReArch-Coder	**17.2** ± 2.6	+10.8	**5.2** ± 4.0	+3.3	**27.8** ± 3.1	+14.4	**20.2** ± 4.0	+9.4
Qwen-Agent	Qwen3-Coder-30B	Base	11.8 ± 2.3	-	4.4 ± 2.9	-	19.8 ± 3.5	-	17.5 ± 4.7	-
		ReArch-Coder	**24.5** ± 2.9	+12.7	**9.6** ± 3.3	+5.2	**34.9** ± 3.9	+15.1	**31.5** ± 5.8	+14.0
	Qwen3-30B	Base	9.3 ± 2.1	-	3.3 ± 2.0	-	17.1 ± 3.0	-	14.5 ± 2.3	-
		ReArch-Coder	**21.2** ± 1.7	+11.9	**7.7** ± 3.8	+4.4	**32.6** ± 3.4	+15.5	**27.0** ± 4.0	+12.5
	CodeLlama-34B	Base	7.1 ± 1.9	-	2.5 ± 0.9	-	14.5 ± 2.8	-	11.9 ± 4.0	-
		ReArch-Coder	**18.4** ± 2.4	+11.3	**7.1** ± 4.0	+4.6	**29.2** ± 3.2	+14.7	**24.0** ± 4.7	+12.1
Codex	Qwen3-Coder-30B	Base	13.5 ± 2.6	-	5.8 ± 3.3	-	23.4 ± 4.0	-	23.2 ± 0.9	-
		ReArch-Coder	**27.4** ± 3.4	+13.9	**11.0** ± 4.0	+5.2	**45.2** ± 4.2	+21.8	**42.8** ± 6.0	+19.6
	Qwen3-30B	Base	11.2 ± 2.1	-	4.4 ± 2.3	-	20.8 ± 3.6	-	18.3 ± 4.0	-
		ReArch-Coder	**25.1** ± 3.0	+13.9	**10.2** ± 4.8	+5.8	**41.5** ± 3.9	+20.7	**35.3** ± 4.7	+17.0
	CodeLlama-34B	Base	9.1 ± 1.8	-	3.3 ± 2.9	-	17.2 ± 3.1	-	15.1 ± 3.8	-
		ReArch-Coder	**21.8** ± 2.5	+12.7	**8.3** ± 4.0	+5.0	**36.6** ± 3.5	+19.4	**30.8** ± 5.1	+15.7
RPG	Qwen3-Coder-30B	Base	12.6 ± 2.8	-	4.4 ± 3.1	-	22.5 ± 3.8	-	17.9 ± 4.2	-
	Qwen3-30B	Base	10.8 ± 2.4	-	3.6 ± 2.5	-	17.1 ± 3.3	-	14.5 ± 3.7	-
	CodeLlama-34B	Base	7.3 ± 2.0	-	2.8 ± 2.2	-	15.4 ± 3.0	-	10.8 ± 3.5	-
Paper2Code	Qwen3-Coder-30B	Base	8.8 ± 2.2	-	2.5 ± 1.6	-	15.2 ± 3.2	-	12.0 ± 3.4	-
	Qwen3-30B	Base	6.9 ± 1.7	-	1.6 ± 1.1	-	12.6 ± 2.8	-	9.4 ± 3.1	-
	CodeLlama-34B	Base	5.1 ± 3.1	-	0.9 ± 1.1	-	10.1 ± 2.3	-	7.2 ± 3.6	-

evaluated agent frameworks, while SWE-Agent obtains low Passed rates on both benchmarks. This suggests that more capable agent frameworks can make better use of ReArch-Coder.

5 Analysis

5.1 Ablation Studies

Table 2 presents the experimental results for different ablation variants. First, the stage-level ablations show that both requirement alignment and architecture design are essential for ReArch-Coder. Removing requirement alignment decreases Passed by 5.2 and 9.6 percentage points on NL2Repo-Bench and RepoGenesis, respectively. Removing architecture design leads to a larger decline, reducing Passed by 8.3 and 13.9 percentage points on the respective benchmarks. Moreover, the component-level ablations validate the detailed design choices of ReArch-Coder. Excluding hint verification lowers performance on both benchmarks, which demonstrates the necessity of filtering unreliable hints.

Replacing reinforcement learning with supervised fine-tuning also leads to a notable performance decline. Additionally, removing either the requirement coverage reward or the dependency modularity reward decreases both Passed(%) and Resolved(%), with the omission of the requirement coverage reward causing a more pronounced degradation.

Table 2: Ablation results for ReArch-Coder with Qwen3-Coder-30B using Codex. Req. and Arch. denote requirement and architecture, respectively.

Variant	NL2Repo-Bench		RepoGenesis	
	Passed (%)	Resolved (%)	Passed (%)	Resolved (%)
ReArch-Coder	**27.4** ±3.4	**11.0** ±4.0	**45.2** ±4.2	**42.8** ±6.0
Stage ablations (w/o)				
Req. alignment	22.2 ±2.1	8.3 ±1.5	35.6 ±3.1	30.2 ±3.4
Arch. design	19.1 ±1.8	7.1 ±1.2	31.3 ±2.5	26.8 ±2.8
Component ablations (w/o)				
Hint verification	25.1 ±2.8	9.6 ±1.9	40.8 ±3.5	36.4 ±4.0
Coverage reward	23.8 ±2.5	9.0 ±2.1	38.9 ±3.2	34.3 ±3.8
Modularity reward	25.6 ±3.0	10.2 ±2.3	42.5 ±3.8	38.7 ±4.3
Architecture training alternative				
SFT training	22.6 ±2.3	7.7 ±1.5	37.2 ±2.8	32.3 ±3.0
PPO optimization	25.0 ±2.6	9.2 ±1.8	41.6 ±3.6	37.5 ±3.8

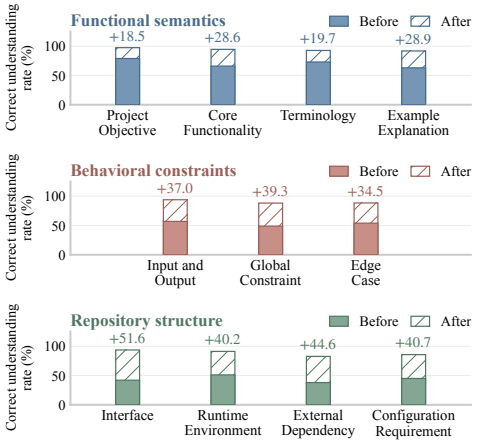


Figure 3: Correct understanding rates before and after requirement alignment.

5.2 Analysis of Requirement Dimensions in Requirement Alignment

Requirement alignment improves the model's understanding of different types of requirements. To examine how different types of requirements are addressed, we group requirements into functional semantics, behavioral constraints, and repository structure, and measure the proportion of correctly understood requirements in each category before and after alignment. As shown in Figure 3, our method corrects most requirement misunderstandings across these three categories. Furthermore, before requirement alignment, requirements related to repository structure are more frequently misunderstood than those related to functional semantics. After requirement alignment, the model's understanding of these structural requirements improves substantially. For example, requirement alignment increases the correct understanding rates for interface and runtime-environment requirements by 51.6 and 40.2 percentage points, respectively.

5.3 Analysis of Performance across Task Difficulty

Table 3: Passed-test rate (%) by task difficulty. Δ denotes gains over the corresponding base models.

Benchmark	Agent	Model	Low			Moderate			High		
			Base	ReArch-Coder	Δ	Base	ReArch-Coder	Δ	Base	ReArch-Coder	Δ
NL2Repo-Bench	OpenHands	Qwen3-Coder	15.4	**34.2**	+18.8	9.8	**22.1**	+12.3	4.8	**11.9**	+7.1
		Qwen3	12.2	**29.4**	+17.2	7.6	**18.2**	+10.6	3.9	**10.1**	+6.2
	Codex	Qwen3-Coder	19.8	**39.4**	+19.6	12.6	**25.4**	+12.8	6.2	**13.8**	+7.6
		Qwen3	16.4	**36.2**	+19.8	10.4	**23.2**	+12.8	5.2	**12.8**	+7.6
RepoGenesis	OpenHands	Qwen3-Coder	27.2	**48.1**	+20.9	17.2	**31.1**	+13.9	8.4	**15.8**	+7.4
		Qwen3	23.8	**45.4**	+21.6	15.1	**29.5**	+14.4	7.5	**14.9**	+7.4
	Codex	Qwen3-Coder	34.2	**65.2**	+31.0	21.8	**42.5**	+20.7	10.9	**21.5**	+10.6
		Qwen3	30.2	**60.0**	+29.8	19.4	**38.9**	+19.5	9.8	**19.8**	+10.0

ReArch-Coder yields consistent gains across all difficulty levels, including the hardest tasks. We group tasks into three difficulty levels according to the number of atomic requirements in each document. Low contains tasks with 1–5 requirements, Moderate contains tasks with 6–10 requirements, and High contains tasks with at least 11 requirements. As shown in Table 3, ReArch-Coder consistently improves every evaluated agent and base model configuration on both benchmarks. For example, on RepoGenesis, ReArch-Coder improves the Codex-based configuration by 31.0 percentage points on low-difficulty tasks and still achieves gains of more than 7.0 percentage points on high-difficulty tasks. Although overall performance decreases as the number of requirements grows, ReArch-Coder maintains improvements over the corresponding baselines in the High group across both benchmarks. These results demonstrate the effectiveness of our method across requirement documents with varying levels of difficulty.

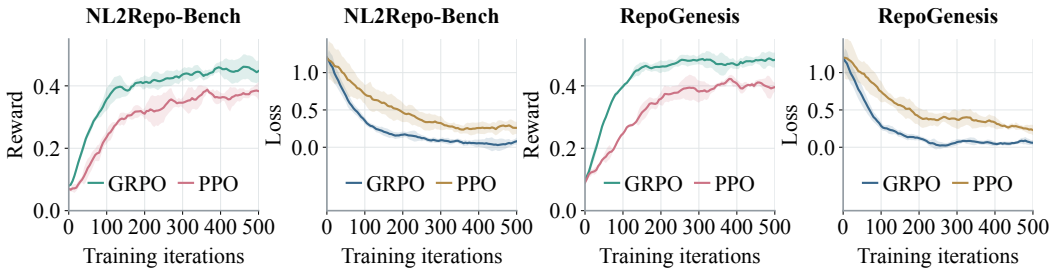


Figure 4: GRPO and PPO training reward and loss over training iterations.

5.4 GRPO versus PPO: Performance and Training Loss

GRPO shows better convergence behavior than PPO on both benchmarks. Figure 4 compares architecture reward and training loss over 500 training iterations on both benchmarks. GRPO's reward increases more rapidly during early training, while PPO improves more gradually and remains below GRPO at the end of training. The loss curves decrease for both methods, with GRPO showing a steeper initial decline and lower final values on both benchmarks.

5.5 Analysis of Token Cost for Correct Solutions

ReArch-Coder uses fewer tokens per resolved task. Figure 5(a) compares token costs with the base method normalized to 1.0. For failed tasks, we report the average token expenditure over those tasks. For passed tests and fully resolved tasks, we divide the total token expenditure across all evaluated tasks by the corresponding number of passed tests and resolved tasks. ReArch-Coder spends more tokens per failed task, but reduces the cost per passed test by 41% on NL2Repo-Bench and 54% on RepoGenesis, and the cost per resolved task by 52% and 54%, respectively. Figure 5(b) reports the average token expenditure per task at each stage, calculated by dividing the total tokens consumed at that stage by the number of evaluated tasks, including both successful and failed tasks.

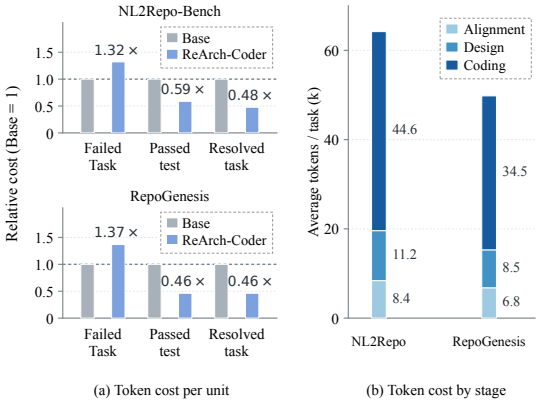


Figure 5: Token cost of the base model and ReArch-Coder by task outcome.

Code generation is the largest contributor on both benchmarks, consuming 44.6K tokens per task on NL2Repo-Bench and 34.5K on RepoGenesis. Requirement alignment consumes 8.4K and 6.8K tokens per task, respectively, while architecture design consumes 11.2K and 8.5K.

6 Conclusion

In this paper, we present ReArch-Coder, a repository generation framework that combines requirement alignment and architecture design optimization. These modules help open-source models follow requirement documents and generate functionally correct repositories. Experiments on NL2Repo-Bench and RepoGenesis show that ReArch-Coder improves the functional correctness of generated repositories compared with existing approaches.

In the future, we plan to explore more effective agentic reinforcement learning methods for long-horizon repository generation and extend our approach to more diverse software development settings. We also aim to investigate the security of generated repositories, including vulnerability prevention, secure dependency management, and compliance with security requirements.

AI USE STATEMENT

In this work, we used generative AI tools to assist with research ideation and discussions of potential approaches. Additionally, we used generative AI tools to support literature retrieval and the discovery of related work, as well as drafting, language polishing, and improving clarity.

We have reviewed all AI-assisted work. We assessed AI-generated research suggestions for soundness and originality through human evaluation and literature checks. We verified the original sources and relevance of retrieved information and references. We manually reviewed AI-assisted writing and revisions to ensure that they accurately conveyed our research. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Yihao Chen, Shi Chang, Khaled Chawa, Feng Lin, Boyuan Chen, Shaowei Wang, and Ahmed E. Hassan. Specfirst: Behavioral specification elicitation as a first-class step in agent-based program synthesis from scratch. *arXiv preprint arXiv:2607.27167*, 2026. URL https://arxiv.org/abs/2607.27167.

Jingzhe Ding et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2026.

Martin Glinz. Problems and deficiencies of UML as a requirements specification language. In *Proceedings of the Tenth International Workshop on Software Specification and Design*, pp. 11–22. IEEE, 2000.

Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: RML revisited. In *Proceedings of the 16th International Conference on Software Engineering*, pp. 135–147. IEEE, 1994.

Daya Guo et al. Deepseek-coder: When the large language model meets programming. *arXiv preprint arXiv:2401.14196*, 2024.

Hanhua Hong et al. Hiras: A hierarchical multi-agent framework for paper-to-code generation and execution. *arXiv preprint arXiv:2604.17745*, 2026.

Sirui Hong et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Binyuan Hui et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

IEEE. IEEE recommended practice for software requirements specifications. IEEE Standard 830-1998, Institute of Electrical and Electronics Engineers, 1998.

Raymond Li et al. Starcoder: May the source be with you! *Transactions on Machine Learning Research*, 2023. URL https://arxiv.org/abs/2305.06161.

Anton Lozhkov et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Jane Luo et al. Rpg: A repository planning graph for unified and scalable codebase generation. In *International Conference on Learning Representations*, 2026.

M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004. doi: 10.1103/PhysRevE.69.026113.

Zhiyuan Peng et al. Repogenesis: Benchmarking end-to-end microservice generation from readme to repository. In *Proceedings of ACL*, 2026.

Chen Qian et al. Chatdev: Communicative agents for software development. In *Proceedings of ACL*, pp. 15174–15186, 2024.

Baptiste Rozi‘ere et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Minju Seo et al. Paper2code: Automating code generation from scientific papers in machine learning. In *International Conference on Learning Representations*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3951–3973, 2024. doi: 10.18653/v1/2024.acl-long.225. URL https://aclanthology.org/2024.acl-long.225.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv:2409.19256*, 2024. URL https://arxiv.org/abs/2409.19256.

Peiding Wang, Li Zhang, Fang Liu, Taichuan Li, and Yinghao Zhu. Codespec: Dual executable specifications for agentic long-horizon feature development. *arXiv preprint arXiv:2607.26777*, 2026. URL https://arxiv.org/abs/2607.26777.

Xingyao Wang et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2025.

John Yang et al. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Daoguang Zan, Ailun Yu, Wei Liu, et al. Codes: Natural language to code repository via multi-layer sketch. *arXiv preprint arXiv:2403.16443*, 2024. URL https://arxiv.org/abs/2403.16443.

Kechi Zhang et al. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of ACL*, pp. 13643–13658, 2024.

Xuanle Zhao et al. Autoreproduce: Automatic ai experiment reproduction with paper lineage. In *Proceedings of ACL*, 2026.

A REQUIREMENT DIMENSIONS

Table 4 organizes document analysis into functional semantics, behavioral constraints, and repository structure. Each dimension provides eight guiding questions for examining requirements across the document. Related dimensions may be considered together when a requirement spans multiple sections. These questions guide checklist construction, with at most 20 diagnostic questions instantiated for each document. Only questions supported by the document are retained, together with reference answers and the relevant evidence excerpts.

Table 4: Guiding questions for requirement document analysis, grouped by dimension. Bold text identifies the focus of each question.

No.	Requirement Analysis Questions
	Project Objectives and Scope — *Functional Semantics*
Q1	**Identify the intended purpose and functional boundaries.** What objectives must the project achieve, and what limits does the document place on its scope?
Q2	**Identify the intended users and their needs.** Which users or calling systems does the project serve, and what tasks must it enable them to complete?
Q3	**Determine the core capabilities required by the project.** Which functions directly support its objectives, and how fully must each function be implemented?
Q4	**Distinguish mandatory requirements from optional capabilities.** Which requirements must always be satisfied, and which apply only under specified conditions or as extensions?
Q5	**Establish responsibility boundaries with external systems.** Which tasks belong to this project, and which are assigned to other systems or to its users?
Q6	**Understand the intended workflow as a whole.** What major stages must users complete from initiating an operation to obtaining the expected result?
Q7	**Identify the conditions and limits of intended use.** What restrictions does the document specify on operating conditions, supported inputs, or supported use cases?
Q8	**Relate project objectives to acceptance requirements.** What observable behaviors or outcomes must the software exhibit for the project to be considered complete?
	Core Functionality — *Functional Semantics*
Q9	**Identify each function's behavior and completion conditions.** What operations must the function perform, and what result must it achieve?
Q10	**Determine the trigger and prerequisites for execution.** What event or action initiates the function, and which conditions must hold before it runs?
Q11	**Trace the main steps within a function.** Which steps are necessary to complete the function, and does the document prescribe their execution order?
Q12	**Analyze dependencies between functions.** Which existing results does a function require, and which subsequent operations consume the results it produces?
Q13	**Determine the state changes caused by execution.** What data must be created, modified, or preserved, and how do these changes affect subsequent behavior?
Q14	**Distinguish behavior under different conditions.** How must execution and results change when the inputs, configuration, or current state differ?
Q15	**Understand the requirements for repeated execution.** When an operation is repeated, must it produce a new result, update existing content, or preserve the current state?
Q16	**Combine descriptions of a function across sections.** What complete behavior is specified jointly by its functional description, constraints, and supplementary conditions?

Continued on the next page

No.	Requirement Analysis Questions
	Terminology and Concepts — *Functional Semantics*
Q17	**Establish the meaning of key terms in the document.** What entity or behavior does a term denote, and which conditions are essential to its definition?
Q18	**Distinguish related concepts with different roles.** What does each term mean, and how do their applicable scope or behavioral requirements differ?
Q19	**Connect different names for the same concept.** Does the document describe an entity using multiple names, and how should those descriptions be associated?
Q20	**Resolve the referent of a term in context.** Does the same name have different meanings across contexts, and which conditions determine its interpretation?
Q21	**Examine the attributes of major entities.** Which attributes does the document assign to an entity, what do they mean, and which behaviors do they affect?
Q22	**Understand relationships between entities.** Which entities have ownership, association, or dependency relationships, and what conditions constrain those relationships?
Q23	**Clarify state names and transitions.** Which states does the document define, what does each state represent, and which transitions are permitted?
Q24	**Identify qualifications on a concept's meaning.** Does its interpretation depend on a role, stage, or mode, and how do these qualifications affect related requirements?
	Example Interpretation — *Functional Semantics*
Q25	**Identify the functionality and rules illustrated by an example.** What operation does it demonstrate, and which requirement connects its input to its result?
Q26	**Understand the context in which an example applies.** How do its initial state, inputs, and configuration jointly determine the demonstrated behavior?
Q27	**Relate an example's result to the written requirements.** Why should the stated result occur, and which passages support this explanation?
Q28	**Compare conditions and behavior across examples.** Which conditions differ, and why do these differences lead to different execution paths or outcomes?
Q29	**Trace state changes across a sequence of example operations.** What changes does each step produce, and how do they affect subsequent steps?
Q30	**Extract output conventions illustrated by examples.** Which explicit requirements govern the content, ordering, and organization of the demonstrated output?
Q31	**Separate example-specific settings from general requirements.** Which details belong to the particular example, and which behaviors apply more broadly according to the document?
Q32	**Interpret examples of boundary conditions and failures.** Why does the example require special handling, and what are the triggering conditions and expected outcomes?
	Input Requirements — *Behavioral Constraints*
Q33	**Determine the sources and meaning of input data.** How does the function receive its inputs, and what information does each kind of input represent?
Q34	**Identify input structure and representation requirements.** Which types, formats, and organizational structures are required, and what content must be included?
Q35	**Establish valid input ranges and their applicability.** Which values are permitted, and do these restrictions change with the operation or execution mode?
Q36	**Distinguish required inputs from optional information.** What must be provided, and which default values or behaviors apply when other information is omitted?

Continued on the next page

No.	Requirement Analysis Questions
Q37	**Analyze constraints involving multiple inputs.** Which parameters must occur together, and which combinations are constrained by dependencies, exclusions, or consistency rules?
Q38	**Understand the effects of ordering and duplication.** How must behavior change when inputs are reordered or the same content is provided repeatedly?
Q39	**Determine how competing input sources are resolved.** When the same information is supplied through multiple channels, which source takes precedence and how are differences handled?
Q40	**Combine input requirements across document sections.** What conditions are jointly imposed by interface descriptions, functional requirements, and usage examples?
	Output Requirements — *Behavioral Constraints*
Q41	**Determine the required results and their meaning.** What must an operation return or generate, and what information does each part convey?
Q42	**Identify output structure and format conventions.** How must results be organized, and what field names, data types, and nesting relationships are specified?
Q43	**Understand precise output representation requirements.** What restrictions govern ordering, precision, and units, and under which conditions do they apply?
Q44	**Distinguish outputs under different execution conditions.** Which parts of the result change with the input or state, and which must remain consistent?
Q45	**Determine how the absence of a result is represented.** When no matching content or returnable data exists, what output and structure are required?
Q46	**Relate returned results to actual state changes.** How should the output reflect saved or modified data, and what correspondence must hold between them?
Q47	**Understand result organization for compound operations.** How should individual outcomes be combined, and must successful and failed parts be reported separately?
Q48	**Determine when and how results are delivered.** At which stage must output be provided, and must files or other persistent artifacts also be produced?
	Global Constraints — *Behavioral Constraints*
Q49	**Identify rules shared by multiple functions.** What global constraints does the document specify, and which entities and execution stages do they cover?
Q50	**Determine conditions that must survive state changes.** Which data relationships or system properties must remain valid, and how do they constrain operations?
Q51	**Understand the relationship between roles and permissions.** What may different users access, and which authorization conditions must hold before an operation proceeds?
Q52	**Analyze consistency requirements for shared data.** When different functions access or modify the same entity, which common rules must they observe?
Q53	**Identify constraints on repeated and concurrent operations.** Which outcomes and state properties must remain valid when operations overlap or execute multiple times?
Q54	**Establish resource and performance limits.** What verifiable requirements does the document impose on execution time, resource consumption, or the scale of processing?
Q55	**Trace common rules throughout data handling.** What requirements apply across functions when data is stored, accessed, updated, or deleted?
Q56	**Understand how global and local rules interact.** When a function has additional requirements, does the document specify any exceptions or precedence relationships?

Continued on the next page

No.	Requirement Analysis Questions
	Boundary Conditions and Error Handling — *Behavioral Constraints*
Q57	**Determine behavior at specified boundaries.** When an input or system state reaches a documented limit, should execution proceed normally or require special handling?
Q58	**Identify handling rules for invalid or missing input.** Which cases must be rejected, and which permit defaults or continued processing?
Q59	**Understand behavior when prerequisites are unmet.** If a required entity, resource, or state is absent, how must the operation respond and proceed?
Q60	**Determine the response to external dependency failures.** When a service or call fails, must execution stop, retry, or follow another documented procedure?
Q61	**Analyze handling after a timeout or interruption.** What happens to completed steps and generated data, and must any associated resources be released?
Q62	**Understand result retention after partial success.** When a step in a compound operation fails, which results should remain and which changes must be undone?
Q63	**Establish conditions for retry and recovery.** Which operations may be repeated, and from what state and under which restrictions must recovery begin?
Q64	**Relate error reporting to the resulting system state.** How must failures be reported, and which validity conditions must hold after error handling completes?
	Interface Contracts — *Repository Structure*
Q65	**Identify entry points for accessing project capabilities.** Which public interfaces must be provided, and what function does each interface expose?
Q66	**Determine interface names and access conventions.** Which names, paths, or invocation forms must callers use, and which conventions must remain consistent?
Q67	**Understand interface parameters and passing rules.** What does each parameter represent, how should it be supplied, and how are omitted parameters handled?
Q68	**Relate interface results to execution status.** What result must a caller receive, and what information indicates whether the operation has completed?
Q69	**Trace invocation prerequisites and effects.** Which conditions must hold before a call, and which observable changes must follow successful execution?
Q70	**Analyze cooperation between interfaces.** In what order must interfaces be called to complete a task, and how are intermediate results passed between them?
Q71	**Identify documented interface compatibility requirements.** Which names, signatures, and exports must be preserved to support the specified patterns of use?
Q72	**Establish boundaries between public and internal capabilities.** Which functionality must be externally accessible, and which is restricted to use within the project?
	Runtime Environment — *Repository Structure*
Q73	**Determine language and runtime requirements.** Which programming language and runtime must the project use, and what version ranges does the document specify?
Q74	**Identify framework and platform requirements.** Which frameworks or execution platforms are prescribed, and how do they constrain the way the project runs?
Q75	**Examine support across operating systems.** Which systems must the project support, and do any require additional preparation or impose distinct restrictions?
Q76	**Identify required system tools and underlying capabilities.** Which operations depend on components supplied by the environment, and what conditions must those components satisfy?
Q77	**Understand filesystem and working-directory assumptions.** Where must resources be accessed, and what rules govern path resolution and read or write permissions?

Continued on the next page

No.	Requirement Analysis Questions
Q78	**Determine the role of network availability.** Which capabilities require connectivity, and what connection methods and access conditions does the document specify?
Q79	**Distinguish environment requirements across execution modes.** Which modes are supported, and what environment preparation and resources does each mode require?
Q80	**Establish the limits of environment compatibility.** Which environments fall within the documented support scope, and under what conditions is execution explicitly restricted?
	External Dependencies — *Repository Structure*
Q81	**Identify the external components required by the project.** Which packages, services, or data resources participate in its functionality, and what role does each play?
Q82	**Distinguish mandatory dependencies from conditional ones.** Which components are prerequisites for overall operation, and which are needed only for particular functions or modes?
Q83	**Determine dependency version and compatibility requirements.** Which versions are permitted, and which functions or execution modes are affected by these restrictions?
Q84	**Understand conditions for connecting external components.** What connection, authentication, or preparation steps are required, and where is the necessary information supplied?
Q85	**Examine conventions for external data and resources.** Where should resources be obtained, and what format requirements govern their use and processing?
Q86	**Analyze preparation and startup dependencies.** Which components must be initialized first, and how does their ordering affect project execution?
Q87	**Establish the scope for replacing external components.** Does the document permit alternatives, and which interface and behavioral contracts must replacements preserve?
Q88	**Determine functional boundaries when dependencies are unavailable.** Which capabilities are affected, and does the document specify other functions that should remain available?
	Configuration and Startup — *Repository Structure*
Q89	**Identify configuration options and their effects.** Which settings does the project provide, and what functionality or execution behavior does each control?
Q90	**Trace configuration sources and precedence.** Through which channels are settings supplied, and how is the effective value determined when several sources specify it?
Q91	**Distinguish required settings from defaults.** Which information must be explicitly provided, and what behavior applies when other settings are omitted?
Q92	**Analyze relationships between configuration options.** What additional conditions apply when a setting is enabled, and which combinations are restricted?
Q93	**Establish the required installation and startup procedure.** Which steps must be completed before execution, and what prerequisite relationships exist between them?
Q94	**Identify data and resource preparation requirements.** Which directories, resources, or initialization operations must be prepared before the project starts?
Q95	**Relate startup entry points to execution modes.** How must the project be launched, and which arguments or settings are required for each supported mode?
Q96	**Determine configuration timing and failure behavior.** When do settings take effect, and how must the project respond when they change, are missing, or are invalid?

B PROMPTS AND OUTPUT SCHEMAS

B.1 EVIDENCE-GROUNDED QUESTION CHECKLIST CONSTRUCTION

Table 5 presents the prompt template for constructing diagnostic questions, reference answers, and supporting evidence. The input placeholders are replaced with the requirement dimensions in Appendix A, manually annotated examples, and the requirement document.

Table 5: Prompt template for evidence-grounded question checklist construction.

Question Checklist Construction

You are an expert in software requirements analysis. Using the provided requirement document and analysis dimensions, construct a diagnostic question checklist to assess a target model's understanding of the requirements. Provide a reference answer and supporting document evidence for each question.

Inputs

Requirement analysis dimensions and guiding questions
{`requirement_dimensions`}
Manually annotated question-answer examples
{`annotated_examples`}
Requirements document
{`requirements_document`}

Task Requirements

1. Read the complete requirement document and use the analysis dimensions to identify specific requirements that must be understood accurately. The guiding questions indicate what to examine. Each diagnostic question must address concrete entities, behaviors, or constraints in the current document.
2. Formulate each question around one independently assessable requirement. A question may address both the triggering conditions and expected behavior of that requirement, but must not combine unrelated requirements.
3. Prioritize requirements whose misinterpretation could affect functional implementation, interface usage, or module responsibility allocation. Pay particular attention to conditions, restrictions, exceptions, and related descriptions distributed across multiple sections.
4. Clearly identify the functionality, interface, or entity addressed by each question and include the necessary context. Each question must be answerable independently using the requirement document, without relying on other questions or their answers.
5. Answer each question directly and preserve all relevant conditions, scope restrictions, and constraints. Use only information explicitly stated in the document or directly entailed by the relevant passages. Do not add unspecified functionality, default behavior, or implementation choices.
6. Provide document excerpts that support the complete reference answer. Copy each excerpt verbatim. When an answer requires information from multiple locations, include the corresponding excerpts separately.
7. Assign each question to the most relevant requirement analysis dimension, using a dimension name from the provided list. A question may combine information from multiple sections. Equal numbers of questions across dimensions are not required.
8. Follow the annotated examples for question granularity, answer specificity, and evidence organization. Requirement facts in an example apply only to that example and must not be used as evidence for the current document.
9. Generate at most 20 questions and avoid asking about the same requirement repeatedly. Omit questions that lack sufficient evidence or cannot be given a definite reference answer. Do not add questions merely to reach the limit.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. The `questions` array must contain between 0 and 20 records. Assign consecutive question identifiers starting from `Q1`. Each record must contain the following fields.

`id` Question identifier
`dimension` Most relevant requirement analysis dimension
`question` Specific diagnostic question
`reference_answer` Reference answer grounded in the requirement document
`evidence` Array of verbatim excerpts supporting the answer

```
{
  "questions": [
    {
      "id": "Q1",
      "dimension": "<dimension name>",
      "question": "<diagnostic question>",
      "reference_answer": "<reference answer>",
      "evidence": ["<verbatim document excerpt>"]
    }
  ]
}
```

B.2 Target Model Question Answering

Table 6 presents the prompt template for eliciting the target model's answers to the diagnostic questions. The inputs consist of the original requirement document and the question identifiers and texts from the checklist. Reference answers and extracted evidence are withheld from the target model.

Table 6: Prompt template for target model question answering.

Target Model Question Answering

Answer the diagnostic questions using the provided requirement document to express your understanding of the relevant software requirements.

Inputs

Original requirement document
{`requirements_document`}
Diagnostic questions, with each record containing only `id` and `question`
{`diagnostic_questions`}

Task Requirements

1. Read the complete requirement document and answer every question. Each answer must directly address the corresponding question.
2. Use information explicitly stated in the document or directly entailed by its content. Preserve applicable conditions, scope restrictions, and exceptions that affect the answer.
3. When relevant information appears in multiple locations, combine it into a complete answer. Do not treat statements in other questions as requirement facts.
4. Do not add functionality, default behavior, or implementation choices that the document does not specify. If the answer cannot be determined from the document, explicitly identify what remains undetermined without guessing.
5. Keep each answer concise while including the requirement details needed to answer the question. Code generation and design recommendations are not required.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. The `answers` array must contain exactly one record per input question, preserving the original order and identifiers. Each record must contain only `id` and `answer`, with `answer` represented as a string. If the question list is empty, return an empty `answers` array.

`id` Original question identifier

`answer` Answer based on the requirement document

```
{
  "answers": [
    {
      "id": "Q1",
      "answer": "<answer based on the requirement document>"
    }
  ]
}
```

B.3 Requirement Understanding Diagnosis

Table 7 presents the prompt template for evaluating the target model's requirement understanding. Diagnostic records pair each checklist question, reference answer, and supporting evidence with the target model's answer using the question identifier. Records judged incorrect are retained for factual hint generation.

Table 7: Prompt template for requirement understanding diagnosis.

Requirement Understanding Diagnosis

You are an expert in software requirements evaluation. Using the original requirement document, reference answers, and supporting evidence, determine whether the target model correctly understands the requirements addressed by each diagnostic question.

Inputs

Original requirement document
{`requirements_document`}
Diagnostic records, each containing `id`, `question`, `reference_answer`, `evidence`, and `target_answer`
{`diagnostic_records`}

Task Requirements

1. Evaluate each record independently and assess only the requirements addressed by its question. Treat the original requirement document as the factual authority, and use the reference answer and evidence to identify the information required for a complete answer.
2. Judge semantic correctness. Accept differences in wording, presentation order, and equivalent summaries without requiring an exact match to the reference answer.
3. Check whether the answer accurately describes the required behavior and its relevant triggering conditions, scope, restrictions, and exceptions. Mark the answer as incorrect if it omits information necessary to answer the question.
4. Check whether the answer contradicts the document or presents unspecified functionality, default behavior, or implementation choices as requirements. Mark the answer as incorrect if it contains such statements.
5. Mark an answer as correct only when it includes all necessary information and contains no contradictions or unsupported requirement claims. For a question clearly answerable from the document, a missing answer or a claim that the answer cannot be determined is also incorrect.

6. Provide a brief, specific explanation of the judgment. For an incorrect answer, identify the misinterpreted, omitted, or unsupported content and state the corresponding facts specified in the document. Do not generate corrective hints or implementation recommendations at this stage.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. The `diagnoses` array must contain exactly one record per input record, preserving the original order and identifiers. Each record must contain `id`, a Boolean `is_correct`, and a string `explanation`. If the input records are empty, return an empty `diagnoses` array.

`id` Original question identifier
`is_correct` Whether the answer correctly expresses the required information
`explanation` Brief judgment rationale grounded in the document

```
{
  "diagnoses": [
    {
      "id": "Q1",
      "is_correct": false,
      "explanation": "<discrepancy and document facts>"
    }
  ]
}
```

B.4 Factual Hint Generation

Table 8 presents the prompt template for generating factual hints from incorrect answer records. Each record combines the question, reference answer, evidence, target model answer, and diagnostic explanation. The same template is reused for regeneration with feedback from hint verification.

Table 8: Prompt template for factual hint generation.

Factual Hint Generation

You are an expert in software requirements analysis. Using the original requirement document and records of incorrect answers, generate factual hints for requirements that the target model has misinterpreted or omitted.

Inputs

Original requirement document
{`requirements_document`}
Incorrect answer records, each containing `id`, `question`, `reference_answer`, `evidence`, `target_answer`, and the diagnostic `explanation`
{`incorrect_records`}
Verification feedback, empty for initial generation and otherwise containing the corresponding question identifier, previous hint, recovered content, and reason for verification failure
{`verification_feedback`}

Task Requirements

1. Identify the facts needing correction from the question, reference answer, incorrect answer, and diagnosis. Verify them using the document and supporting evidence.
2. Clarify misinterpretations, supply omitted facts, and correct unsupported requirement claims. Do not treat unspecified behavior as explicitly prohibited.
3. Use only information stated or directly entailed by the document. Preserve relevant conditions, scope, restrictions, and exceptions. Add no new requirements or implementation advice.

4. Write each hint as a concise, self-contained requirement statement. Name the relevant entity or behavior. Do not mention question identifiers, answers, or the diagnostic process.
5. Generate one hint per incorrect record, covering the facts needing correction. Do not repeat unrelated requirements or rewrite the document.
6. When verification feedback is provided, revise the hint using discrepancies between the recovered content and the original text. Clarify previously unclear facts while remaining grounded in the document.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. The `hints` array must contain exactly one record per input incorrect answer record, preserving the original order and identifiers. Each record must contain only `id` and a string `hint`. If the incorrect answer records are empty, return an empty `hints` array.

`id` Original question identifier
`hint` Self-contained requirement statement grounded in the document

```
{
  "hints": [
    {
      "id": "Q1",
      "hint": "<self-contained requirement statement>"
    }
  ]
}
```

B.5 Masked Content Recovery

Table 9 presents the prompt template for recovering masked requirement content. The target model receives only the masked local excerpt and the factual hint, together with the question identifier. The original unmasked excerpt is withheld.

Table 9: Prompt template for masked content recovery.

Masked Content Recovery

Recover the masked requirement content using the local context and factual hint.

Inputs

Original question identifier
{id}
Local requirement excerpt with masked spans
{masked_excerpt}
Factual hint
{hint}

Task Requirements

1. Fill every masked span using only the supplied context and hint.
2. Preserve relevant conditions, restrictions, and exceptions. Add no unsupported details.
3. Return the completed excerpt with all unmasked text unchanged.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. Preserve the input `id` and represent `recovered_content` as a string.

```
{
  "id": "Q1",
  "recovered_content": "<completed excerpt>"
}
```

B.6 HINT VERIFICATION

Table 10 presents the prompt template for judging masked recovery. The judge compares the recovered content with the original excerpt. For failed verification, the explanation is paired with the previous hint and recovered content as feedback for hint regeneration.

Table 10: Prompt template for hint verification.

Hint Verification

Determine whether the recovered content preserves the original requirement semantics.

Inputs

Original question identifier
{id}
Original requirement excerpt
{original_excerpt}
Corresponding excerpt with masked spans
{masked_excerpt}
Completed excerpt from the target model
{recovered_content}

Task Requirements

1. Compare each recovered span with its original content in context.

2. Accept equivalent wording. Reject omissions, contradictions, or unsupported additions.

3. Pass only when every span is correct. Otherwise, briefly identify the discrepancies and original facts.

Output Requirements

Return only a valid JSON object without Markdown formatting or additional commentary. Preserve the input `id`. Use a Boolean `is_valid` and a string `explanation`.

```
{
  "id": "Q1",
  "is_valid": false,
  "explanation": "<specific discrepancies>"
}
```

C ARCHITECTURE DESIGN TEMPLATES

Table 11 presents a unified prompt for generating the complete architecture design trajectory $(M, A^{(0)}, \Delta, A)$. The four stages share the aligned requirement document and a requirement list with fixed identifiers. For example, `R1` identifies one requirement throughout the trajectory.

Table 11: Prompt template for repository architecture design.

Repository Architecture Design

You are a software architect. Design a repository architecture from the aligned requirement document by completing the following four stages in order.

Inputs

Aligned requirement document
{`aligned_requirements_document`}
Requirement records, each containing `id` and `text`
{`requirements`}

Task Requirements

1. **Allocate requirements.** Assign every requirement to a functional module and define its responsibility. Group related behavior and establish clear module boundaries. Preserve the supplied requirement identifiers.
2. **Compose the initial architecture.** Translate the module responsibilities into directories and files. Specify each file's responsibility, imports, and function or class signatures. Map every requirement to the files responsible for implementing it. Preserve documented interfaces and runtime constraints.
3. **Diagnose the initial architecture.** Examine whether file responsibilities and interfaces cover the requirements. Identify uncovered requirements, misplaced or duplicated responsibilities, inconsistent interfaces, and inappropriate dependencies. Record each issue and its proposed correction.
4. **Revise the architecture.** Apply the corrections supported by the requirements and diagnosis. Update affected files, imports, signatures, and requirement assignments consistently. Preserve unaffected designs and return the complete final architecture.

Do not add new product requirements or generate implementation bodies.

Output Requirements

Return only a valid JSON object containing the four stages below, without Markdown formatting or additional commentary. Use repository-relative file paths. Keep requirement identifiers unchanged throughout the trajectory. Return an empty `issues` array when no defects are found. The initial and final architectures must use the same file schema.

```
{
  "requirement_allocation": {
    "modules": [
      {
        "id": "M1",
        "name": "<module name>",
        "responsibility": "<module responsibility>",
        "requirement_ids": ["R1"]
      }
    ]
  },
```

```
  "initial_architecture": {
    "files": [
      {
        "path": "<relative path>",
        "responsibility": "<file responsibility>",
        "imports": ["<import declaration>"],
        "signatures": ["<function or class signature>"],
        "requirement_ids": ["R1"]
      }
    ]
  },
  "diagnostic_report": {
    "issues": [
      {
        "requirement_ids": ["R1"],
        "files": ["<relative path>"],
        "issue": "<specific defect>",
        "correction": "<proposed change>"
      }
    ]
  },
  "final_architecture": {
    "files": [
      {
        "path": "<relative path>",
        "responsibility": "<revised file responsibility>",
        "imports": ["<import declaration>"],
        "signatures": ["<function or class signature>"],
        "requirement_ids": ["R1"]
      }
    ]
  }
}
```

D REWARD COMPUTATION

The reward is computed from the `final_architecture` field of the design trajectory. Requirement allocation, the initial architecture, and the diagnostic report receive no separate rewards.

D.1 REQUIREMENT COVERAGE

The input requirement list defines a fixed, nonempty set $\mathcal{R}$ whose identifiers remain unchanged across trajectories for the same task. Let F denote the files declared in the final architecture. For each file f, $\Gamma(f)$ contains the identifiers in its `requirement_ids` field. The covered requirement set $\mathcal{C}$ is obtained by taking the union of these identifiers and intersecting it with the input requirement set.

$$\mathcal{C} = \mathcal{R} \cap \bigcup_{f \in F} \Gamma(f), \qquad R_{\text{cov}}(A) = \frac{|\mathcal{C}|}{|\mathcal{R}|}. \tag{5}$$

Each requirement contributes at most once, even when assigned to multiple files. Identifiers absent from the input list do not increase coverage. This score measures requirement allocation in the architecture and does not assess the functional correctness of subsequent code.

Assigning all valid identifiers to one file yields full coverage even when its responsibility and interfaces do not support those requirements. Identifier filtering and deduplication therefore constrain counting but do not verify semantic support. Appendix **??** specifies an evaluation protocol for examining this gap without modifying the training reward.

D.2 FILE DEPENDENCY GRAPH

Each file in the final architecture forms one node. Internal dependencies are extracted from its `imports` field by matching imported modules or classes to the declared file paths and signatures under the naming rules of the target language. For example, an import from `service/core.py` in `service/api.py` establishes a dependency between those two files. Standard library and third-party imports are excluded from the internal dependency graph.

The graph is undirected and unweighted. An edge connects two distinct files whenever either file imports the other. Repeated imports and reciprocal imports are merged into a single edge, and self-imports are excluded. Files without internal dependencies remain as isolated nodes.

Files are grouped by their immediate parent directory, using the full repository-relative directory path. For example, `service/api.py` and `service/core.py` belong to the same group, while `storage/store.py` belongs to another group. Files directly under the repository root form one root group. Nested directories remain separate from their parents. These path-based groups define the partition used for modularity computation, with no additional community detection.

D.3 DEPENDENCY MODULARITY

Let m be the number of unique edges. For each directory group c, L_c counts edges with both endpoints in that group, and d_c sums the degrees of all files in the group. An internal edge contributes one to L_c and two to d_c. An edge between two groups contributes one to each group's degree sum and contributes to neither group's internal edge count. The dependency reward is

$$R_{\text{dep}}(A) = \begin{cases} \displaystyle\sum_c \left[\frac{L_c}{m} - \left(\frac{d_c}{2m} \right)^2 \right], & m > 0, \\ 0, & m = 0. \end{cases} \tag{6}$$

For a graph with at least one edge, the raw modularity value is retained, including negative values, without clipping or normalization.

D.4 REWARD AGGREGATION

The final reward uses the weights specified in the experimental setup.

$$R(A) = 0.7 R_{\text{cov}}(A) + 0.3 R_{\text{dep}}(A). \tag{7}$$

Illustrative calculation. Consider four input requirements, three of which are assigned to files, giving $R_{\text{cov}} = 3/4$. Suppose the architecture has four files in two directories, with one internal edge in each directory and one edge connecting the directories. Then $m = 3$, and each directory has $L_c = 1$ and $d_c = 3$. The resulting scores are

$$\begin{aligned} R_{\text{dep}} &= 2\left[\frac{1}{3} - \left(\frac{3}{6}\right)^2\right] = \frac{1}{6}, \\ R(A) &= 0.7 \times \frac{3}{4} + 0.3 \times \frac{1}{6} = 0.575. \end{aligned} \tag{8}$$

D.5 Structural Interpretation and Failure Modes

Modularity compares the fraction of edges inside each directory with its expectation under a random graph model that preserves node degrees (Newman & Girvan, 2004). It can describe dependency concentration when directories represent functional modules. A dependency from a request handler to a service in another directory can be necessary for correct behavior. The reward does not classify such individual dependencies as correct or incorrect, and a layered microservice architecture may have lower modularity despite an appropriate separation of responsibilities. Its suitability therefore depends on how directory boundaries correspond to functional boundaries.

The graph contains imports declared in the architecture, so omitted dependencies and artificial directory isolation can distort the reward. Consider k disconnected directories, each containing two files and one internal edge. Then $m = k$, $L_c = 1$, and $d_c = 2$ for every directory. With full declared requirement coverage, the scores are

$$R_{\text{dep}} = 1 - \frac{1}{k}, \qquad R(A) = 1 - \frac{0.3}{k}. \tag{9}$$

For $k = 10$, the reward is 0.97 regardless of whether any required behavior is implemented. This construction establishes a vulnerability of the proxy objective. Its occurrence in sampled training trajectories requires separate empirical investigation.

E IMPLEMENTATION DETAILS

Table 12 summarizes the implementation and training hyperparameter settings used in our experiments. The prompt templates and output formats are provided in Appendices B and C.

Table 12: Implementation and hyperparameter settings for requirement alignment, architecture training, and repository generation.

Setting	**Value**
Requirement Alignment	
Question, reference answer, and hint generation	GPT-5.5
Requirement diagnosis and hint verification	Claude-Opus-4.6
Diagnostic questions per document	At most 20
Verification attempts per hint	At most 3
Alignment sampling parameters	Temperature 0.2, top-p 0.95
Architecture Design RL Training	
Base models	Qwen3-Coder-30B, Qwen3-30B, CodeLlama-34B
Training algorithm and framework	GRPO implemented with verl
Training projects	52 from NL2Repo-Bench and 53 from RepoGenesis
Group size G (trajectories per document)	8
Optimizer	AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.01)
Learning rate	1×10^{-5} (cosine schedule, 5% warmup)
GRPO clipping parameter ε	0.2
KL divergence coefficient β	0.01
Reward weights (λ_{cov}, λ_{dep})	0.7 and 0.3
RL rollout sampling parameters	Temperature 0.7, top-p 0.95
Hardware and Evaluation	
Training hardware	8 $\times$ NVIDIA H100 (80GB SXM5), 1TB CPU RAM
Repository generation environment	Isolated Docker container
Inference sampling parameters	Temperature 0.2, top-p 0.95
Time limit per task	1 hour

F Evaluation Methods and Dataset Splits

This section describes the benchmark datasets, project-level splits, execution environments, and metric calculations used in our evaluation.

F.1 Benchmark Datasets and Project Splits

We evaluate repository-level code generation on NL2Repo-Bench (Ding et al., 2026) and RepoGenesis (Peng et al., 2026). Table 13 summarizes the dataset statistics and split sizes.

Table 13: Statistics and project-level splits of evaluated benchmarks.

Attribute	**NL2Repo-Bench**	**RepoGenesis**
Software type	Python libraries	Microservice repositories
Target languages	Python	Python and Java
Total projects	104	106
Training projects	52	53
Evaluation projects	52	53
Evaluation test suite	Upstream pytest suites	Black-box REST API tests
Split random seed	42	42

Both benchmarks are split at the repository level with fixed random seed 42, ensuring that files and modules from the same repository never appear across splits. All annotated examples used in the question checklist prompt are sampled exclusively from the training projects.

F.2 Execution Environment and Protocol

Each task begins with an empty workspace inside a dedicated Docker container. The container is allocated 8 CPU cores, 32GB RAM, and 50GB local disk space.

The 1-hour wall-clock timeout bounds the end-to-end repository generation lifecycle for each task. For baseline methods, this duration covers repository planning, tool invocation, code writing, and test execution. For ReArch-Coder, this duration covers requirement alignment, architecture trajectory generation, multi-file code synthesis, and test execution. When a generated repository exhibits syntax errors, missing dependencies, or runtime timeouts during test execution, all unexecuted test cases receive a score of zero.

F.3 Evaluation Metrics and Aggregation

Let N denote the number of evaluated repositories in a benchmark, where $N = 52$ for NL2Repo-Bench and $N = 53$ for RepoGenesis. For repository k, let T_k be the total number of test cases and P_k be the number of passed test cases. In an evaluation run s, the pass rate and resolved rate are computed as

$$\text{Passed}_s = \frac{1}{N}\sum_{k=1}^{N}\frac{P_k}{T_k} \times 100\%, \qquad \text{Resolved}_s = \frac{1}{N}\sum_{k=1}^{N}\mathbb{I}(P_k = T_k) \times 100\%. \tag{10}$$

G API ENDPOINT REACHABILITY AND CORRECTNESS

Table 14 separates interface availability from functional correctness on RepoGenesis. An API endpoint is an operation identified by an HTTP method and a route, such as `POST /files` for uploading a file. Reachability concerns whether a request reaches the intended endpoint handler in the generated service. Correctness concerns whether that endpoint satisfies its specified behavior, including response content, status codes, and persistent data changes. An endpoint can therefore be reachable while producing incorrect results.

Table 14: API endpoint reachability and correctness on RepoGenesis.

Agent	**Model**	**Reachability (%)**		**Correctness (%)**	
		Base	ReArch-Coder	Base	ReArch-Coder
OpenHands	Qwen3-Coder-30B	34.2	**58.6**	22.4	**39.8**
	Qwen3-30B	29.8	**54.1**	19.6	**36.7**
	CodeLlama-34B	25.1	**49.7**	16.3	**32.9**
Qwen-Agent	Qwen3-Coder-30B	36.5	**61.2**	24.1	**42.3**
	Qwen3-30B	31.8	**56.8**	20.8	**38.9**
	CodeLlama-34B	27.2	**52.4**	17.5	**34.8**
Codex	Qwen3-Coder-30B	41.8	**74.5**	28.6	**53.1**
	Qwen3-30B	37.4	**69.8**	25.3	**48.7**
	CodeLlama-34B	32.0	**63.2**	21.0	**43.4**

For Qwen3-Coder-30B using Codex, the values increase from 41.8% to 74.5% for reachability and from 28.6% to 53.1% for correctness. These correspond to differences of 32.7 and 24.5 percentage points. The 21.4 percentage point gap between ReArch-Coder's reachability and correctness represents endpoints that can be called but fail at least one associated functional test.

H Analysis of Diagnostic Question Budget

This section evaluates the sensitivity of requirement alignment to the maximum number of diagnostic questions generated per requirement document. We vary the question limit across 5, 10, 15, and 20 while keeping all downstream architecture design and code generation settings identical, using Qwen3-Coder-30B with Codex. Table 15 reports the actual number of generated questions, the number of diagnosed requirement misunderstandings, and the resulting functional test metrics on both benchmarks.

Table 15: Effect of diagnostic question budget on requirement diagnosis and repository generation with Qwen3-Coder-30B under Codex.

Limit	**NL2Repo-Bench**				**RepoGenesis**			
	Questions	Errors	Passed (%)	Resolved (%)	Questions	Errors	Passed (%)	Resolved (%)
5	4.8	2.1	23.6	8.8	4.6	1.8	38.2	33.5
10	9.2	3.8	25.4	9.7	8.7	3.2	41.8	37.8
15	13.5	5.2	26.8	10.6	11.9	4.1	44.1	40.9
20	16.8	6.1	**27.4**	**11.0**	13.8	4.6	**45.2**	**42.8**

Limit denotes the maximum number of diagnostic questions generated per document. Questions and Errors report average counts of generated questions and identified misunderstandings per task.

Across both benchmarks, increasing the question limit expands the number of diagnosed misunderstandings and improves functional test pass rates. On NL2Repo-Bench, raising the limit from 5 to 20 increases average generated questions from 4.8 to 16.8 and diagnosed errors from 2.1 to 6.1, with Passed increasing from 23.6% to 27.4% and Resolved from 8.8% to 11.0%. On RepoGenesis, the same increase raises generated questions from 4.6 to 13.8 and diagnosed errors from 1.8 to 4.6, improving Passed from 38.2% to 45.2% and Resolved from 33.5% to 42.8%.

The rate of performance improvement decreases as the question budget grows. On NL2Repo-Bench, increasing the limit from 5 to 10 yields a 1.8 percentage point gain in Passed, whereas increasing from 15 to 20 yields a 0.6 percentage point gain. Similarly, on RepoGenesis, the gain in Passed is 3.6 percentage points between limits 5 and 10, compared with 1.1 percentage points between limits 15 and 20. The average question count remains below the upper limit of 20 because shorter requirement documents produce fewer independently verifiable questions that satisfy the evidence extraction criteria.

I Limitations

This work has the following limitations. First, our experiments cover Python libraries and microservices implemented in Python and Java. ReArch-Coder consistently improves performance on these tasks, but its applicability to other programming languages, software types, and settings with evolving requirements requires further validation. Second, requirement alignment currently relies on proprietary teacher models, so deployment must account for external service availability and the privacy of requirement documents. Although ReArch-Coder reduces token expenditure per resolved task, actual costs and latency still depend on model pricing and service response times. Future work could explore locally deployed teacher models and evaluate their effectiveness and cost.