RAL-Bench: Benchmarking for Application-Level Functional Correctness and Non-Functional Quality Attributes

RUWEI PAN, Chongqing University, China and Peking University, China
YAKUN ZHANG, Harbin Institute of Technology (Shenzhen), China
QINGYUAN LIANG, Peking University, China
YUEHENG ZHU, Chongqing University, China and Peking University, China
CHAO LIU, Chongqing University, China
LU ZHANG, Peking University, China
HONGYU ZHANG*, Chongqing University, China

Project Page   Evaluation Code   HuggingFace

Code generation has been an active research area for decades, and recent studies increasingly leverage large language models (LLMs) to produce more complex code. With the rapid progress of code-focused LLMs, snippet-level code generation has improved substantially. These advances primarily reflect improvements on snippet-level outputs (e.g., function or repository completion), whereas application-level generation requires producing a runnable repository by coordinating multi-file implementations with proper structure, dependency management, and end-to-end executability. However, real-world software engineering demands not only functional correctness but also non-functional quality attributes (e.g., maintainability and security). Existing benchmarks provide limited support for evaluating, under execution, whether generated code meets both functional correctness and non-functional quality attributes. Although quality attributes are important at all levels, we focus on application-level generation. These limitations motivate the following research question: Can current LLMs generate application-level repositories that satisfy functional correctness and non-functional quality attributes? To answer this, we propose `RAL-Bench`, an evaluation framework and benchmark for application-level code generation. For each task, we distill a concise natural-language requirement from a high-quality reference project. We then construct black-box system tests that cover both functional correctness and non-functional quality attributes. Finally, we retain only candidate tests that pass on the reference repository. This filtering ensures a sound test oracle and an end-to-end executable test suite. Functional correctness is quantified by a functional score, defined as the system-test pass rate. Non-functional quality attributes are evaluated along five ISO/IEC 25010–inspired dimensions and aggregated using an Analytic Hierarchy Process (AHP)–derived weight vector, with per-dimension diagnostics. In addition, we collect baseline non-functional metrics on the reference repository to enable baseline-normalized scoring. We comprehensively evaluate 16 LLMs in zero-shot settings using greedy decoding. We find that functional correctness is the primary bottleneck: under our requirement-driven, reference-validated black-box system tests, no LLM exceeds a 45% functional pass rate. Our findings suggest that widely used evaluation results may not faithfully reflect LLM performance on application-level code generation, and they also point to new directions for improving generation capabilities. We have open-sourced our benchmark at **https://github.com/Wwstarry/RAL-Bench** to facilitate and accelerate future research on LLM-based code generation.

Additional Key Words and Phrases: Code Generation, End-to-End Application Generation, Non-functional Requirements

*Corresponding author.

Authors' Contact Information: Ruwei Pan, Chongqing University, China and Peking University, China; Yakun Zhang, Harbin Institute of Technology (Shenzhen), China; Qingyuan Liang, Peking University, China; Yueheng Zhu, Chongqing University, China and Peking University, China; Chao Liu, Chongqing University, China; Lu Zhang, Peking University, China; Hongyu Zhang, Chongqing University, China.

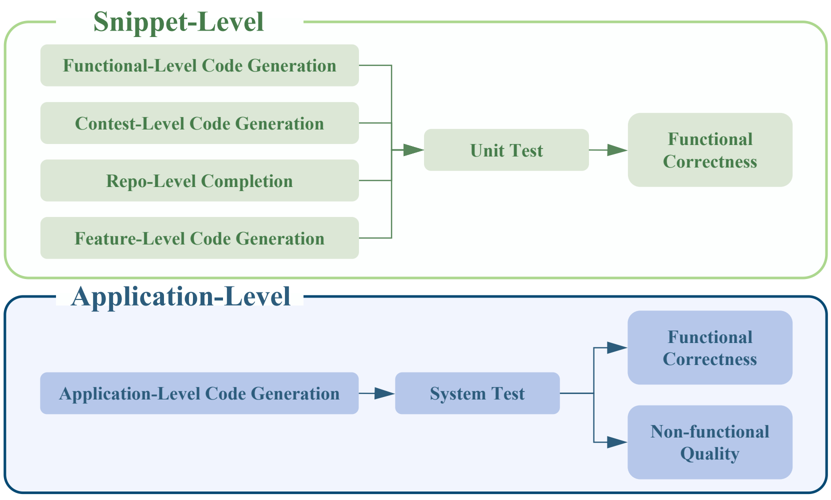


Fig. 1. Scope comparison between snippet-level and application-level code generation.

1 Introduction

In daily development, we often articulate our needs in the form of a single, highly compressed requirement like *"implement a pure Python steganography library"*. For human developers, the real difficulty usually does not lie in implementing a particular function, but in integrating it into a fully working application [39, 42]. This process requires decisions about project organization, dependency management, functional completeness, and compliance with non-functional quality attributes such as robustness requirements [4, 12, 25]. In contrast, LLM-generated code can appear as a complete project while failing during dependency installation or runtime execution, which snippet-level evaluations typically do not capture.

Automatically generating applications from a natural-language requirement is a long-term goal of LLM-based code generation. With the rapid development of code LLMs, snippet-level code generation has made significant progress [5, 30, 43]. However, real-world software engineering requires more than functional correctness. It also demands non-functional quality attributes [15, 40]. Existing code benchmarks shown in Table 1 (e.g., HumanEval [7], MBPP [2], APPS [17], LiveCodeBench [21], CoderEval [47], DevEval [29], and NoCodeBench [10]) focus on snippet-level generation (e.g., function-level, contest-level, repo-level, and feature-level code generation), but they remain insufficient for evaluating application-level generation, where LLMs must produce a runnable repository with proper structures, dependency management, and end-to-end executability. Fig. 1 summarizes the evaluation scope difference between snippet-level and application-level benchmarks. Repo-level and feature-level generation focus on completing a codebase with the expected module structure, where evaluation is typically based on unit tests that check snippet-level correctness. Consequently, we argue that current code benchmarks are insufficient for assessing the ability to generate application-level code. Specifically, we identify the following common limitations in existing LLM-for-code benchmarks [2, 10, 24, 29, 37, 47]:

- **Lack of application-level evaluation.** Most benchmarks focus on evaluating functional correctness using snippet-level outputs under predefined unit tests. They do not test whether an LLM can generate a runnable project with proper structures, dependency management and end-to-end executability.

Table 1. Comparison with existing code generation benchmarks.

Benchmark	**Task Type**	**Functional Correctness**	**Non-functional Quality Attributes**	**Real-world Project**
HumanEval [7]	Function-level	✓	✗	✗
MBPP [2]	Function-level	✓	✗	✗
EvalPlus [36]	Function-level	✓	✗	✗
EvalPerf [37]	Function-level	✓	✓	✗
EffiBench [18]	Function-level	✓	✓	✗
APPS [17]	Contest-level	✓	✗	✗
LiveCodeBench [21]	Contest-level	✓	✗	✗
RepoEval [38]	Repo-level	✓	✗	✓
CoderEval [47]	Repo-level	✓	✗	✓
DevEval [29]	Repo-level	✓	✗	✓
FEA-Bench [31]	Feature-level	✓	✗	✓
NoCodeBench [10]	Feature-level	✓	✗	✓
FeatBench [6]	Feature-level	✓	✗	✓

- **No assessment of non-functional quality attributes.** Applications are constrained not only by functional correctness but also by non-functional quality attributes. However, existing benchmarks almost exclusively focus on functional tests. While a few snippet-level benchmarks have started to consider non-functional aspects, they focus on a single attribute (e.g., efficiency or security) on isolated code snippets, rather than assessing multi-attribute non-functional quality end to end. Real-world application quality is a multi-attribute construct that cannot be reduced to functional correctness alone. An implementation may pass functional tests yet remain unusable in practice due to latency, resource inefficiency, robustness issues, security risks or poor maintainability.

These limitations are shared by many widely used code generation benchmarks. They not only call into question the validity of the impressive performance reported in prior work, but also highlight the open challenge of how to properly evaluate LLM-based code generation. In this paper, we aim to address this fundamental evaluation gap and pose an introspective question: *Can current LLMs generate application-level repositories that satisfy functional correctness and non-functional quality attributes?*

Our proposal. In this work, we set out to answer this question by revisiting how evaluation datasets are constructed and by building a more realistic evaluation framework. Specifically, we build RAL-Bench (**R**eal-World **A**pplication-**L**evel Code Generation **Bench**), an evaluation framework for assessing LLMs' ability to generate executable applications from a natural-language requirement in three steps. First, instead of relying on curated function signatures, we extract concise task descriptions directly from real GitHub repositories, mirroring how developers articulate needs in the real world. Second, we construct black-box system tests that verify not only functional correctness but also key non-functional quality attributes. Third, for each task, we select a widely used, high-quality GitHub project as the ground-truth repository and execute all candidate tests on the reference implementation, retaining only those that pass. This filtering step ensures that the test oracle is sound and that the suite is end-to-end executable, before we apply the same tests to generated code. This framework ensures that all evaluation artifacts are grounded in real-world software practice.

Contributions. Our work revisits the problem of evaluating LLM-based code generation and proposes a new benchmark that measures whether LLMs can generate executable applications.

- **Study:** We are the first to systematically analyze the evaluation gap between existing code benchmarks and the requirements of real-world application development. Our study also opens up a new research direction for precisely and rigorously evaluating application-level code generation.
- **Approach:** We propose RAL-Bench, a benchmark and evaluation framework for application-level code generation grounded in real-world GitHub repositories. For each task, we extract a concise natural-language requirement from a high-quality reference project and construct black-box system tests covering both functional correctness and key non-functional quality attributes. We execute all candidate tests on the reference repository and retain only those that pass, ensuring a sound test oracle and end-to-end executability. Functional score is computed as the system test pass rate. Non-functional quality is measured along five ISO/IEC 25010-inspired dimensions and aggregated using an AHP-derived weight vector with per-dimension diagnostics [3]. In addition, baseline non-functional metrics are collected on the reference repository to enable baseline-normalized scoring.
- **Results:** We comprehensively evaluate 16 LLMs (standard and thinking) under zero-shot settings with greedy decoding. First, we find that functional correctness is the dominant bottleneck: under our requirement-driven, reference-validated black-box system tests, no LLM surpasses a 45% functional pass rate. Second, although non-functional scores are generally higher, they cannot offset functional failures. Third, our failure-pattern dataset comprises 446 successfully generated repositories and over 4,500 test-case execution logs. It shows that failures are dominated by Requirement–Implementation Mismatch and Non-functional Quality Failures (82.8% combined), whereas Executability & Dependency Failures account for 17.2%. Fourth, we quantify cost. Thinking LLMs are more expensive on average, yet they do not yield consistent functional improvements. This suggests that higher-cost "thinking" does not yet translate into effective reasoning for application-level generation. Finally, the results show that when tasks scale to the application level, mainstream code generation strategies are no longer effective.

2 RAL-Bench

2.1 Overview

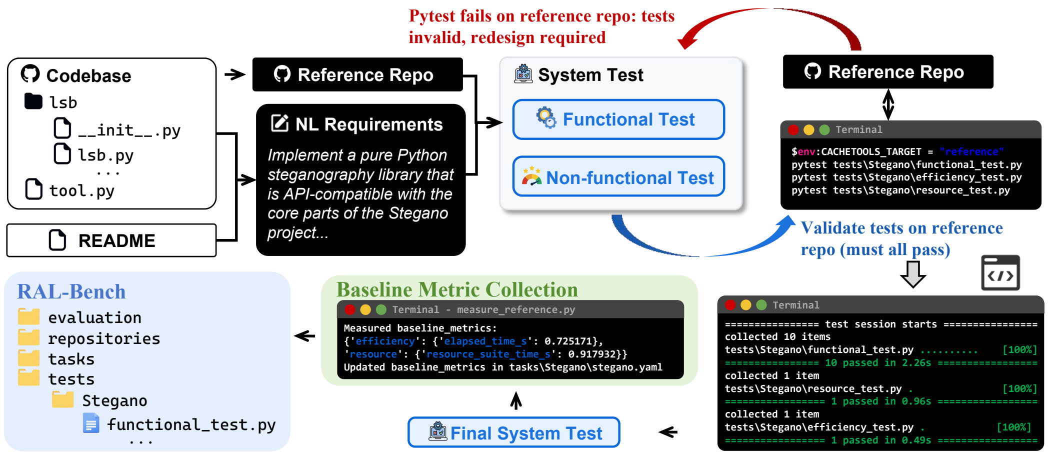
We introduce RAL-Bench to quantitatively evaluate the ability of LLMs to perform application-level code generation. RAL-Bench comprises over 450 functional and non-functional evaluation points derived from 38 real projects and covers seven distinct real-world usage scenarios (e.g., Data and Security), as summarized in Table 2. These projects are sourced from actively maintained GitHub repositories with an average popularity of more than 1,000 stars, ensuring that they are widely used and vetted by the open-source community. Each project provides clearly specified functional requirements and measurable non-functional quality attributes (e.g., efficiency and security). Moreover, none of the projects are drawn from existing code generation benchmarks or public evaluation suites, which helps mitigate potential benchmark contamination and data leakage for LLMs. RAL-Bench is iteratively reviewed to minimize requirement ambiguity and to ensure that all manually designed test cases are precise, suitable, and executable for rigorous evaluation.

Table 2. Repository manifest grouped by the seven real-world usage scenarios. Pinned commit SHA fixes the exact repository snapshot for reproducible evaluation.

Scenario	Repository	Commit (40-hex SHA)	#Files	LoC
Tooling	astanin/python-tabulate	`74885be915e2ac611585f5398f23c402532c1059`	1	2.4k
	martinblech/xmltodict	`f7d76c96fc0141238947abcc5fa925d3ffd9eb78`	3	1.4k
	mkaz/termgraph	`b86ccfddf55eda83bb2c3b9afe4c74478559bcb1`	8	1.0k
	pallets/click	`cdab890e57a30a9f437b88ce9652f7bfce980c1f`	16	7.7k
	pygments/pygments	`28ec10c5e154ee27201997feb573a0b065f74082`	339	109.5k
	python-cmd2/cmd2	`926636a38156e4a7e07dc1cee04bca7b019776ec`	21	9.1k
Data	CamDavidsonPilon/lifelines	`47afb1c1a272b0f03e0c8ca00e63df27eb2a0560`	47	20.8k
	dateutil/dateutil	`e081f6725fbb49cae6eedab7010f517e8490859b`	37	14.4k
	jazzband/tablib	`7d6c58a574782b2525af17225e93bdf4efa0f376`	31	5.0k
	petl-developers/petl	`482fc04fd2589ac9404a8e6a21601b956aa64a2f`	124	27.0k
	pudo/dataset	`b2ab09e58c6f17334e4286009b20887b6a8a8fac`	6	1.1k
	python-pendulum/pendulum	`754ed58a6c232b2335a967d5e266fc48c448b8f3`	172	19.6k
Web	Python-Markdown/markdown	`e5fa5b86e8ec380cbc520cfc637d72c779e5c601`	33	5.7k
	fastapi/sqlmodel	`1e4bf5c190e849e5d070f437a061667776788013`	12	2.3k
	psf/requests	`70298332899f25826e35e42f8d83425124f755a5`	35	8.4k
	python-visualization/folium	`0ff5d993bf2c60dbf8f56c12bbc67f104b97687c`	48	8.2k
Security	cedricbonhomme/Stegano	`3ffac4782b8e4e5893c3919e1eabb921867220cb`	18	1.1k
	fail2ban/fail2ban	`9887ee441215a2a1f889b0e1baaa43cd3a956d26`	78	24.9k
	jpadilla/pyjwt	`04947d75dc45ba1a4a66eaa2b24fbb0eb512ceab`	12	2.1k
	mitmproxy/mitmproxy	`a7621021191011de7c95771459688de3ecd67c10`	242	39.1k
	sqlmapproject/sqlmap	`876f14199ecb33ee16ed09b40f9ebe5225a1cd74`	102	27.6k
Automation	celery/celery	`4d068b5667b75e21844f0748d13627e70d5a42ac`	156	32.2k
	dbader/schedule	`82a43db1b938d8fdf60103bd41f329e06c8d3651`	1	0.7k
	fastapi/typer	`a8c425b613aeb645553a418b4aa6e3b7f4a8db77`	16	4.1k
	msiemens/tinydb	`2283a2b556d5ef361b61bae8ed89c0ce0730d7c8`	10	1.4k
	nvbn/thefuck	`c7e7e1d884d3bb241ea6448f72a989434c2a35ec`	206	4.8k
	tkem/cachetools	`9983ef8bd76758707ab9d197d4bd9fa47b4fb8bd`	5	1.0k
Observability	Delgan/loguru	`764cd30d4b285ca01718ab3162a9f1a022bc49c6`	19	4.0k
	Textualize/rich	`36fe3f7ca9becca4777861d5e6e625f5a4a37545`	78	23.2k
	gorakhargosh/watchdog	`3f8a12f9cf6d9f8091e05b61952409ed9623257d`	54	8.2k
	nicolargo/glances	`7cd3eeb1a67c8a1623887e0d8b7de70797a9696e`	129	18.7k
	python-humanize/humanize	`c2b096a96fa8850399a84401267d5c174e57e7e3`	12	2.6k
Content	imageio/imageio	`761929cc6ae6503f93f2431e0b47d504837ba77a`	49	26.4k
	mailpile/Mailpile	`741e610f3e765f03bd0c452c92c6758280e7f99f`	130	43.2k
	py-pdf/pypdf	`19735763b856cccf0f69630d0f582a448ec5d8bb`	56	37.3k
	quodlibet/mutagen	`905e8152d57efc1eb80ae3cb2b557735ede8d070`	56	14.5k
	sffjunkie/astral	`ac23ab5c0c69837d8fa1a5bb184c2d6d125b26b3`	36	4.9k
	un33k/python-slugify	`7b6d5d96c1995e6dccb39a19a13ba78d7d0a3ee4`	5	0.3k

2.2 Benchmark Construction

We develop an executable benchmark construction pipeline to curate real-world tasks, derive natural-language requirements, build executable system test suites, and evaluate generated repositories at scale, as shown in Fig. 2.


Pytest fails on reference repo: tests invalid, redesign required
Codebase
lsb
__init__.py
lsb.py
...
tool.py
README
Reference Repo
NL Requirements
Implement a pure Python steganography library that is API-compatible with the core parts of the Stegano project...
System Test
Functional Test
Non-functional Test
Reference Repo
Validate tests on reference repo (must all pass)
RAL-Bench
evaluation
repositories
tasks
tests
Stegano
functional_test.py
...
Baseline Metric Collection
Final System Test

```
Terminal
$env:CACHETOOLS_TARGET = "reference"
pytest tests\Stegano\functional_test.py
pytest tests\Stegano\efficiency_test.py
pytest tests\Stegano\resource_test.py
```

```
Terminal - measure_reference.py
Measured baseline_metrics:
{'efficiency': {'elapsed_time_s': 0.725171},
'resource': {'resource_suite_time_s': 0.917932}}
Updated baseline_metrics in tasks\Stegano\stegano.yaml
```

```
Terminal
================ test session starts ================
collected 10 items
tests\Stegano\functional_test.py ..........     [100%]
================ 10 passed in 2.26s =================
collected 1 item
tests\Stegano\resource_test.py .                [100%]
================= 1 passed in 0.96s =================
collected 1 item
tests\Stegano\efficiency_test.py .             [100%]
================= 1 passed in 0.49s =================
```

Fig. 2. Overall construction pipeline of RAL-Bench.

Reference Repository Selection. We begin by selecting well-established GitHub projects as reference repositories, each with clearly documented functionality and active real-world use. For every chosen project, we pin a specific commit and snapshot its codebase within our evaluation framework. We treat this pinned snapshot as the ground-truth implementation because it is a stable, widely adopted application-level project.

Natural-Language Requirement Distillation. Given a reference repository, we distill a natural-language requirement description (NL requirement) from its README and source code. First, we synthesize a concise requirement description that summarizes the project's core intent. Second, we deliberately phrase the requirement to mirror how developers articulate intent in practice. It takes the form of a single, highly compressed request, such as *implement a pure Python steganography library*. Third, the requirement specifies what the application should do and how it is expected to be used. However, it intentionally avoids prescribing a concrete project layout or implementation strategy. Under this setting, LLMs must expand a terse intent into a fully structured, executable project and infer the architecture, dependencies, and non-functional considerations without further guidance.

System Test Generation. Next, we construct system test suites for each task, consisting of black-box functional tests and non-functional tests (e.g., efficiency and security). Functional tests aim to exercise diverse functional requirements derived from real-world projects. Non-functional tests assess key non-functional quality attributes via static analysis and controlled runtime checks. Each candidate test case is first executed against the corresponding reference repository. If a test fails on the reference implementation, we deem it invalid and discard it. We retain only test cases that pass on the reference repository. This filtering ensures a sound test oracle and guarantees that the suite can be executed end-to-end in a fully automated pipeline.

Baseline Metric Collection. We run the non-functional test suite on the reference repository to collect baseline metrics for each task. These statistics are stored alongside the task configuration and later used when evaluating generated repositories. This enables fair and consistent comparisons across models and tasks.

2.3 Metric Design

We evaluate application-level code generation using two complementary metrics: functional correctness and non-functional quality attributes.

Functional correctness. The functional score of functional correctness is defined as the functional test pass rate, computed as the number of passed tests divided by the total number of tests for each task.

Non-functional quality attributes. Following the ISO/IEC 25010 quality model [13], we additionally assess five non-functional dimensions that are critical in practice: maintainability, security, robustness, efficiency, and resource usage.

1 **Maintainability** is measured by the lower-bound Maintainability Index (MI) from static analysis. We operationalize maintainability using MI as a static proxy for structural complexity. Higher MI generally reflects lower structural complexity and therefore improved maintainability. Let g and b denote the MI lower bounds of the generated code and the reference implementation, respectively. To avoid hard saturation when directly using g/b, we apply a smooth compression function:

$$\mathrm{M} = \frac{g/b}{1+g/b}. \tag{1}$$

2 **Security** is quantified by the number of high-risk issues reported by static security analysis. Let g and b denote the number of high-risk findings in the generated code and the reference implementation, respectively. We compute a reference-aligned inverse ratio with smoothing:

$$\mathrm{S} = \min\left(1, \frac{b+1}{g+1}\right). \tag{2}$$

3 **Robustness** is evaluated using a dedicated robustness test suite targeting invalid, boundary, and unexpected inputs. Let *passed* and *total* denote the number of passed tests and the total number of robustness tests. The robustness score is defined as:

$$\mathrm{Rb} = \frac{passed}{total}. \tag{3}$$

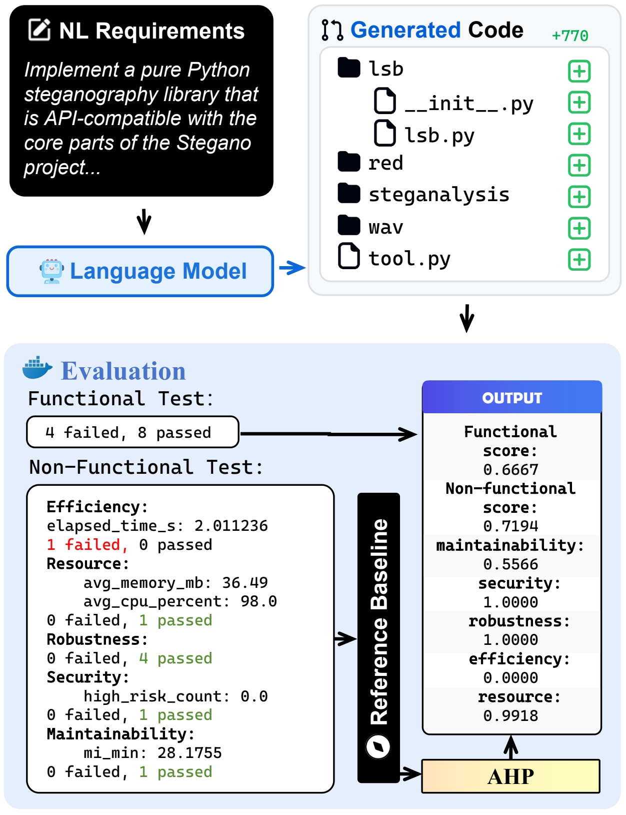
4 **Efficiency** is measured from the runtime statistics of the *efficiency suite*, using the elapsed execution time reported by the suite. Let T_{gen} and T_{ref} denote the efficiency-suite execution times of the generated code and the reference implementation, respectively. If the efficiency suite fails to execute successfully (e.g., failed tests), we set $\mathrm{E} = 0$; otherwise:

$$\mathrm{E} = \min\left(1, \frac{T_{\mathrm{ref}}}{T_{\mathrm{gen}}}\right). \tag{4}$$

5 **Resource usage** is measured using a *dedicated resource suite*. During execution, we sample resident set size (RSS) memory usage and CPU utilization of the `pytest` subprocess and its child processes. We then compute the average memory (MB) and average CPU utilization (%) over the run. Let $M_{\mathrm{gen}}, C_{\mathrm{gen}}$ and $M_{\mathrm{ref}}, C_{\mathrm{ref}}$ denote the corresponding averages for the generated code and the reference implementation. If the resource suite fails to execute successfully (e.g., failed tests), we set $\mathrm{Ru} = 0$. Otherwise, we compute reference-aligned ratios and average memory and CPU when both are available:

$$\mathrm{Ru} = \begin{cases} \frac{1}{2}\left(\min\left(1, \frac{M_{\mathrm{ref}}}{M_{\mathrm{gen}}}\right) + \min\left(1, \frac{C_{\mathrm{ref}}}{C_{\mathrm{gen}}}\right)\right), & \text{CPU available,} \\ \min\left(1, \frac{M_{\mathrm{ref}}}{M_{\mathrm{gen}}}\right), & \text{memory only.} \end{cases} \tag{5}$$

To reflect the fact that non-functional quality attributes correspond to qualitatively different engineering risks (e.g., security vulnerabilities and efficiency regressions), we determine attribute weights using the Analytic Hierarchy Process (AHP). We ground the relative importance ordering on Botchway et al., who apply AHP to expert questionnaires and report maintainability and security as the highest-priority quality attributes, followed by robustness and efficiency [3]. Based on this



```
4 failed, 8 passed
```

```
Efficiency:
elapsed_time_s: 2.011236
1 failed, 0 passed
Resource:
    avg_memory_mb: 36.49
    avg_cpu_percent: 98.0
0 failed, 1 passed
Robustness:
0 failed, 4 passed
Security:
    high_risk_count: 0.0
0 failed, 1 passed
Maintainability:
    mi_min: 28.1755
0 failed, 1 passed
```

```
Functional
score:
0.6667
Non-functional
score:
0.7194
maintainability:
0.5566
security:
1.0000
robustness:
1.0000
efficiency:
0.0000
resource:
0.9918
```

Fig. 3. Evaluation pipeline overview.

ordering, we construct a pairwise comparison matrix using the standard Saaty scale [14]. We compute the normalized principal eigenvector to obtain the weight vector and verify consistency using the AHP consistency ratio (CR). In our setting, CR = 0.030 (< 0.1), indicating an acceptable level of consistency. The full AHP derivation (matrix, eigenvector solution, CI/RI/CR) is documented in **our repository** for reproducibility. The non-functional score aggregates five normalized quality dimensions using a weighted sum. Here, M, S, Rb, E, and Ru denote the normalized scores of maintainability, security, robustness, efficiency, and resource usage, respectively:

$$\mathrm{NF} = 0.36\,\mathrm{M} + 0.24\,\mathrm{S} + 0.16\,\mathrm{Rb} + 0.12\,\mathrm{E} + 0.12\,\mathrm{Ru}. \tag{6}$$

This non-functional score provides a more reliable and discriminative measure of the practical quality of generated applications, beyond functional correctness. In Section 3.2, we show that it is both discriminative across LLMs and stable under reruns.

2.4 Evaluation Pipeline

We evaluate application-level code generation with an end-to-end pipeline, as illustrated in the evaluation workflow (Fig. 3).

Input. First, for each target project, we provide the LLM with a concise natural-language requirement that captures the core functionality to be implemented. To ensure strict interface alignment and avoid evaluation artifacts caused by mismatched entry points, we also provide the reference repository's expected module and package surface, guiding the model to generate a repository that conforms to the same API boundary. Given this requirement and the interface constraints, the LLM generates a complete repository from scratch.

Functional Correctness via System Tests. We then evaluate functional correctness using system test suites. Specifically, we execute functional tests against the generated repository and compute the functional score as the pass ratio across all functional test cases, which directly measures whether the generated application behavior matches the reference implementation's intended functionality.

The Role of Interface Constraints. On STEGANO with GPT-5.2, omitting the interface specification yields a functional score of 0.1667. Providing the interface specification increases the score to 0.6667. Inspecting the corresponding failure cases shows that this improvement primarily stems from alleviating test-interface mismatches (e.g., incorrect entry points or module paths) rather than from failing to satisfy the task requirements. As a result, the evaluation more accurately reflects the LLM's ability to produce an executable repository that satisfies the end-to-end functional requirements.

Non-functional Measurement. Beyond functional correctness, we further assess non-functional quality attributes by running a set of tests and analyses that produce measurable signals used by our scoring functions. These include execution time for efficiency (e.g., *elapsed_time_s*), resource utilization statistics (e.g., *avg_memory_mb* and *avg_cpu_percent*), static security findings (e.g., the number of high-risk issues *high_risk_count*), and maintainability indicators (e.g., the minimum maintainability index *mi_min*). For robustness, we define it as the system's ability to remain stable under stress and edge-case execution paths. We quantify robustness using the pass ratio on our robustness-oriented test suite. A higher score indicates the generated project remains stable under stress and edge-case execution paths.

Reference-normalized Scoring. Because different projects have very different scales, we compute a per-project baseline by running the non-functional suite on the reference repository. We then score each generated repository by comparing its non-functional metrics against this baseline, making results comparable across projects. Concretely, for efficiency, resource usage, security, and maintainability, we first run the same measurement pipeline on the reference repository to obtain a task-specific baseline for each dimension. We then evaluate the generated repository under the same environment and normalize each raw measurement against its baseline using fixed, dimension-specific rules, yielding a score in [0, 1]. A higher score indicates closer-to-reference (or better-than-reference) behavior on that dimension. The exact scoring rules and normalization functions are defined in Section 2.3.

Aggregation and Outputs. Finally, we aggregate the five non-functional dimension scores into an overall non-functional index using an AHP-derived weight vector [3]. This produces a weighted assessment of non-functional quality attributes that emphasizes practically important attributes while remaining consistent across projects. The pipeline outputs both the functional score and the non-functional score, together with the per-dimension breakdown (maintainability, security, robustness, efficiency, and resource usage), enabling the subsequent analyses to localize bottlenecks.

3 Benchmarking on RAL-Bench

We evaluate the performance of state-of-the-art (SOTA) LLMs on application-level code generation with RAL-Bench. Specifically, we aim to address the following research questions (RQs):

- RQ1: What is the performance of current LLMs in application-level code generation?
- RQ2: Where do LLMs fail in application-level code generation?

Table 3. Evaluation scores reported as percentages (higher is better). Best values per column are in bold.

LLM	Provider	Scores (%)						
		Func.	Non-func.	Maint.	Sec.	Robust.	Eff.	Res.
Gemini-2.5-Pro	Google	27.63	35.15	26.27	54.39	41.39	24.84	25.28
Gemini-3-Pro-preview	Google	**43.86**	50.57	31.50	76.75	66.36	**43.69**	41.23
GPT-3.5-Turbo	OpenAI	29.76	55.08	36.92	**97.37**	79.62	29.80	17.57
GPT-4o-2024-11-20	OpenAI	27.48	57.59	**37.50**	96.05	**84.07**	35.54	27.71
GPT-4-Turbo	OpenAI	36.42	**58.13**	34.90	**97.37**	78.32	42.24	38.31
GPT-5-2025-08-07	OpenAI	38.50	57.51	33.84	94.74	77.02	41.81	43.73
GPT-5.2	OpenAI	41.81	57.05	33.90	92.11	77.19	42.79	**43.83**
Claude-Haiku-4.5-20251001	Anthropic	34.74	56.40	34.52	**97.37**	79.66	37.19	28.32
Claude-Sonnet-4.5	Anthropic	39.32	49.64	30.19	84.21	66.93	31.50	33.94
DeepSeek-V3-0324	DeepSeek	24.37	46.34	32.29	81.58	65.89	22.90	15.36
DeepSeek-V3.2	DeepSeek	29.92	51.26	36.07	89.47	63.49	26.91	28.46
Average	–	33.98	52.25	33.45	87.40	70.90	34.47	31.25

- RQ3: What is the cost of application-level code generation with current LLMs?
- RQ4: Are mainstream code generation strategies effective for application-level code generation?

3.1 Evaluation Setup

Benchmark Models. We comprehensively evaluate 16 LLMs across diverse model families. For standard LLMs, we consider Gemini-2.5-Pro [9], Gemini-3-Pro-Preview, GPT-3.5-Turbo, GPT-4o-2024-11-20 [19], GPT-4-Turbo [1], GPT-5 [27], GPT-5.2, Claude-Haiku-4.5, Claude-4.5-Sonnet, DeepSeek-V3-0324 [33], and DeepSeek-V3.2 [34]. For thinking LLMs, we evaluate Gemini-2.5-Pro-Thinking, GPT-o1 [20], GPT-o3-2025-04-16, Claude-3.7-Sonnet-Thinking, and DeepSeek-R1 [16]. All evaluations are conducted in a zero-shot setting using each model's default prompt template. For each task, LLMs output the entire repository in one pass, within the LLM's context-window limits. To ensure reproducibility, we repeat each evaluation three times under identical settings and report the mean score across runs for all metrics.

3.2 RQ1: What is the performance of current LLMs in application-level code generation?

Table 3 and Table 4 show overall model performance on `RAL-Bench`. Our key observations are as follows:

Main results. Across all evaluated LLMs, functional correctness remains low. Standard LLMs achieve an average functional score of 33.98%, whereas thinking LLMs achieve 30.23%. Even the top-performing LLMs stay below 45% functional score, indicating that a large fraction of application-level tasks cannot be solved end-to-end by current LLMs.

Functional correctness. Functional correctness is the primary bottleneck. No LLM surpasses a 50% functional score, indicating persistent difficulty in completing end-to-end application logic and satisfying the full set of functional requirements exercised by the test suite.

Table 4. Evaluation scores for thinking models reported as percentages (higher is better). Best values per column are in bold.

LLM	Provider	Scores (%)						
		Func.	Non-func.	Maint.	Sec.	Robust.	Eff.	Res.
Gemini-2.5-Pro-Thinking	Google	**43.44**	**56.09**	34.80	89.47	**75.16**	**42.68**	**41.15**
GPT-o1	OpenAI	33.88	55.44	**37.44**	92.11	74.07	36.35	30.37
GPT-o3-2025-04-16	OpenAI	30.27	55.44	35.01	**96.05**	73.60	35.89	30.80
Claude-3.7-Sonnet-Thinking	Anthropic	27.34	52.94	37.29	92.11	63.34	30.65	29.95
DeepSeek-R1	DeepSeek	16.23	27.95	12.36	56.84	38.42	18.57	12.36
Average	–	30.23	49.57	31.38	85.32	64.92	32.83	28.92

Non-functional quality attributes. Non-functional scores are generally higher than functional scores (52.25% on average for standard LLMs and 49.57% for thinking LLMs). This suggests that once a generated repository is runnable, it often exhibits acceptable non-functional quality attributes, yet still fails to satisfy the end-to-end functional requirements. Importantly, the non-functional scores are also discriminative. When functional correctness is similar, they can differentiate LLMs by a wide margin. For example, Gemini-2.5-Pro and GPT-4o-2024-11-20 achieve nearly identical functional scores (27.63% vs. 27.48%), yet their non-functional scores differ substantially (35.15% vs. 57.59%). This gap is driven by differences in the per-dimension breakdown. In particular, many LLMs show relatively good performance on security and robustness checks. Nevertheless, such non-functional strengths are complementary rather than substitutive. Non-functional quality attributes cannot compensate for failures in core functionality in application-level code generation.

Efficiency and resource usage. Efficiency and resource usage vary substantially across LLMs. Some LLMs achieve competitive efficiency-suite runtimes and resource behavior when the workload executes successfully. In contrast, others exhibit pronounced inefficiency. This variance suggests that producing not only runnable but also efficient and resource-aware applications remains an open challenge.

Thinking models. Thinking mechanisms do not yield uniform gains. Gemini-2.5-Pro-Thinking shows a large improvement over Gemini-2.5-Pro (e.g., functional score 43.44% vs. 27.63%). However, other thinking LLMs do not consistently outperform standard LLMs. On average, thinking LLMs remain slightly below standard LLMs in both functional and non-functional scores. This indicates that reasoning-enhanced strategies are not yet a reliable lever for improving application-level generation.

Rerun Stability. We evaluate the stability of our metrics by rerunning the entire benchmark pipeline five times for three representative models (GPT-5.2, Gemini-3-Pro-Preview, and Claude-4.5-Sonnet). For each generation, we measure variability across reruns using the standard deviation (Std) and coefficient of variation (CV), and then summarize the distribution across the 38 projects by reporting the median (med) and 95th percentile (p95). Table 5 yields three key observations. ❶ **Functional scores are perfectly reproducible.** For all LLMs and projects, the functional score exhibits zero variance across reruns, with Std and CV both equal to 0. This indicates that functional evaluation is strictly deterministic and free of randomness-induced statistical drift. ❷ **The aggregated non-functional score is highly stable.** The overall standard deviation is 0.0003 (med) and 0.0012 (p95). The overall CV is 0.0007 (med) and 0.0028 (P95). In relative terms, this corresponds to about 0.07% variation at the median project and 0.28% at the 95th percentile. This

Table 5. Rerun stability ($K = 5$ full reruns). For each LLM, we compute the standard deviation (Std) and coefficient of variation (CV) over K reruns per project, then summarize across 38 projects using median (med) and 95th percentile (p95).

Model	K	**Proj.**	**Functional Score**				**Non-functional Score**			
			Std		**CV**		**Std**		**CV**	
			med	**p95**	**med**	**p95**	**med**	**p95**	**med**	**p95**
Claude-4.5-Sonnet	5	38	0.0000	0.0000	0.0000	0.0000	0.0003	0.0009	0.0007	0.0026
Gemini-3-Pro-Preview	5	38	0.0000	0.0000	0.0000	0.0000	0.0003	0.0017	0.0010	0.0050
GPT-5.2	5	38	0.0000	0.0000	0.0000	0.0000	0.0002	0.0006	0.0005	0.0017
Overall	5	38	0.0000	0.0000	0.0000	0.0000	0.0003	0.0012	0.0007	0.0028

Model	K	**Proj.**	**Runtime Metrics (CV)**					
			Time		**Memory**		**CPU**	
			med	**p95**	**med**	**p95**	**med**	**p95**
Claude-4.5-Sonnet	5	38	0.0208	0.0501	0.0072	0.0134	0.0111	0.0270
Gemini-3-Pro-Preview	5	38	0.0319	0.0795	0.0080	0.0136	0.0137	0.0353
GPT-5.2	5	38	0.0198	0.0468	0.0074	0.0138	0.0098	0.0214
Overall	5	38	0.0224	0.0557	0.0075	0.0138	0.0127	0.0270

suggests that non-functional comparisons are largely insensitive to rerun noise. ❸ **Runtime metrics vary slightly across reruns** due to normal execution variability, such as OS scheduling, caching effects, and background processes on the machine. This variability does not materially affect the final non-functional evaluation. Across projects, the CV for runtime time is 0.0224 (med) and 0.0557 (p95), i.e., about 2.24% median relative fluctuation and 5.57% at the tail. For memory, CV is 0.0075 (med) and 0.0138 (p95). For CPU, CV is 0.0127 (med) and 0.0270 (p95). Importantly, we normalize each non-functional dimension against its reference baseline and then aggregate normalized dimension scores. This design reduces the influence of small run-to-run fluctuations, so the final non-functional results remain stable and do not change model-to-model conclusions.

Answer to RQ1: *Application-level generation is constrained by functional requirements and non-functional quality attributes under end-to-end execution, rather than by producing a superficially runnable repository. Moreover, thinking LLMs are not a reliable way to improve application-level code quality. They can yield large gains for specific model families (e.g., Gemini) but do not produce consistent improvements across providers. This suggests that additional deliberation does not reliably improve application-level code quality.*

3.3 RQ2: Where do LLMs fail in application-level code generation?

Setup. To understand where and why current LLMs fail in application-level code generation, we conduct a fine-grained failure analysis based on our evaluation results. We analyze 446 successfully generated repositories and over 4,500 test-case execution logs. Based on these artifacts, we construct a failure-pattern dataset for application-level code generation.

Main Results. Our analysis reveals three dominant failure modes in this setting: **executability and dependency failures**, **requirement–implementation mismatches**, and **non-functional quality Failures**. Fig. 4 visualizes the distribution of these failure modes, showing a model-wise breakdown on the left and the overall composition on the right. **At the aggregate level**, requirement–implementation mismatches and non-functional quality failures constitute the primary

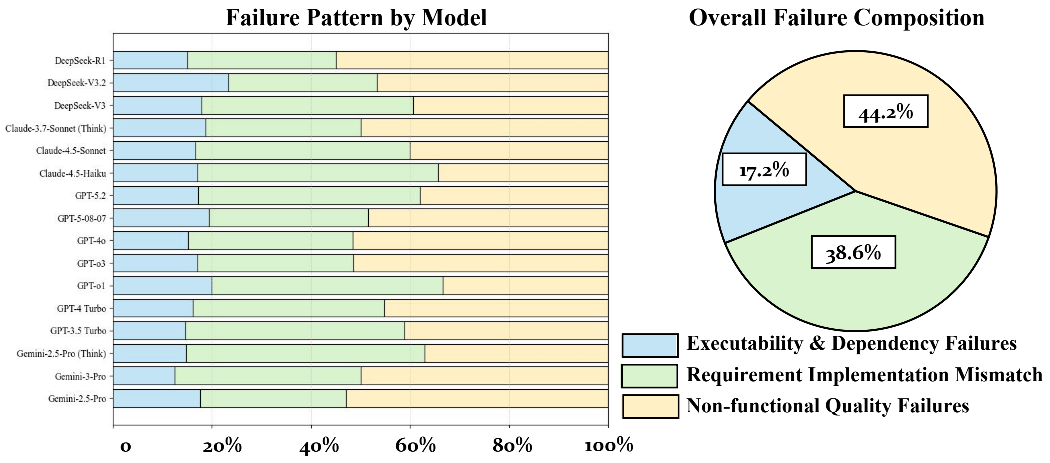
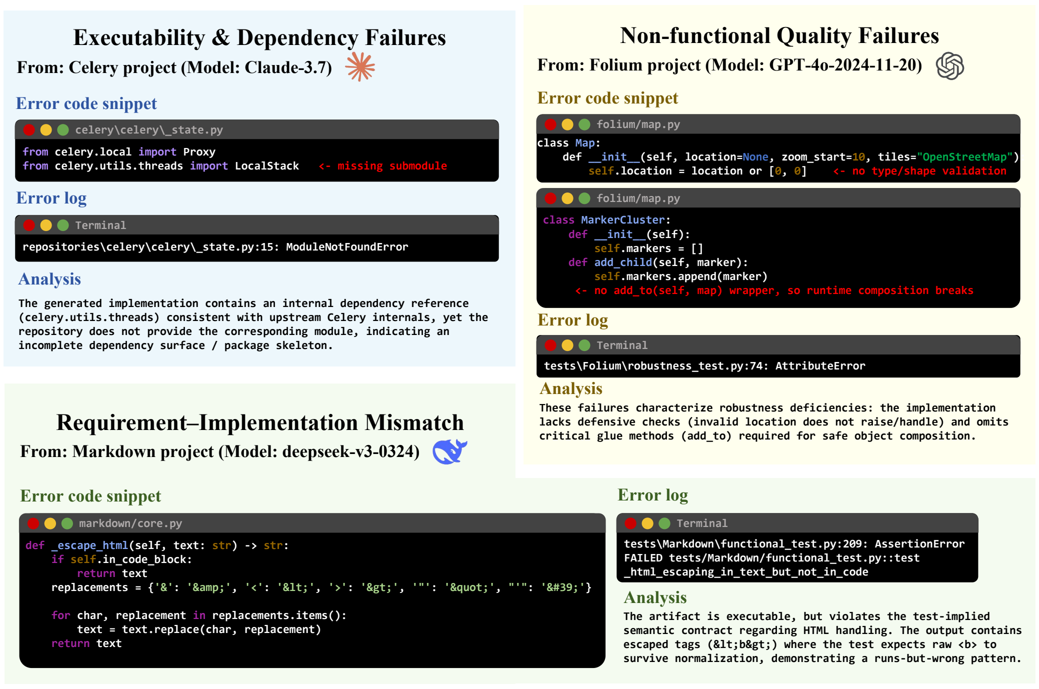


Fig. 4. Failure-pattern distribution for application-level code generation (left: breakdown by model; right: overall composition).

sources of failures, accounting for 82.8% of all cases. In contrast, executability and dependency failures account for 17.2%. **At the model level**, Fig. 4 (left) shows substantial cross-model heterogeneity in how failures distribute across the three patterns. Executability and dependency failures are consistently the smallest share (roughly 15% to 20%). By contrast, requirement–implementation mismatches and non-functional quality failures dominate for every LLM. However, LLMs differ in which application-level bottleneck prevails. Some models are primarily limited by requirement–implementation mismatches even after achieving executability. Others fail more often due to runtime instability or timeouts.

Case Studies. Fig. 5 presents three representative failures, one per category. Together, they show that application-level generation can fail at different stages, including import-time collection, assertion contract checking, and runtime robustness evaluation. **In the executability and dependency failure case,** the Celery repository generated by Claude-3.7-Sonnet-Thinking fails during pytest collection. The code imports `celery.utils.threads.LocalStack`, but the referenced submodule is missing, raising a `ModuleNotFoundError`. This failure is structural rather than semantic. The repository does not expose a complete, importable dependency surface, and functional verification therefore cannot meaningfully begin. **In the requirement implementation mismatch case,** the Markdown repository generated by DeepSeek-V3-0324 runs the tests but fails on assertions. Its HTML escaping routine produces escaped tags such as `&lt;b&gt;` where the test implied contract expects raw <b> to be preserved in specific contexts. The artifact is executable, yet its behavior diverges from the specification encoded in assertions. As a result, it runs successfully but produces incorrect outputs. **In the non-functional quality failure case,** the Folium repository generated by GPT-4o-2024-11-20 reaches runtime but breaks under robustness-oriented execution paths. Missing defensive checks and incomplete compositional interfaces such as omitting `add_to` cause runtime exceptions including `AttributeError`. This illustrates that stability and API completeness are central to application-level usability even when basic functionality seems to work.

Error Analysis. To characterize where LLMs fail in application-level code generation, our taxonomy separates failures that arise before functional verification from those that arise during functional execution, and further distinguishes requirement and runtime-quality bottlenecks:


Executability & Dependency Failures

From: Celery project (Model: Claude-3.7)

Error code snippet

```
celery\celery\_state.py
from celery.local import Proxy
from celery.utils.threads import LocalStack   <- missing submodule
```

Error log

```
Terminal
repositories\celery\celery\_state.py:15: ModuleNotFoundError
```

Analysis

The generated implementation contains an internal dependency reference (celery.utils.threads) consistent with upstream Celery internals, yet the repository does not provide the corresponding module, indicating an incomplete dependency surface / package skeleton.

Non-functional Quality Failures

From: Folium project (Model: GPT-4o-2024-11-20)

Error code snippet

```
folium/map.py
class Map:
    def __init__(self, location=None, zoom_start=10, tiles="OpenStreetMap")
        self.location = location or [0, 0]    <- no type/shape validation
```

```
folium/map.py
class MarkerCluster:
    def __init__(self):
        self.markers = []
    def add_child(self, marker):
        self.markers.append(marker)
  <- no add_to(self, map) wrapper, so runtime composition breaks
```

Error log

```
Terminal
tests\Folium\robustness_test.py:74: AttributeError
```

Analysis

These failures characterize robustness deficiencies: the implementation lacks defensive checks (invalid location does not raise/handle) and omits critical glue methods (add_to) required for safe object composition.

Requirement–Implementation Mismatch

From: Markdown project (Model: deepseek-v3-0324)

Error code snippet

```
markdown/core.py
def _escape_html(self, text: str) -> str:
    if self.in_code_block:
        return text
    replacements = {'&': '&amp;', '<': '&lt;', '>': '&gt;', '"': '&quot;', "'": '&#39;'}

    for char, replacement in replacements.items():
        text = text.replace(char, replacement)
    return text
```

Error log

```
Terminal
tests\Markdown\functional_test.py:209: AssertionError
FAILED tests/Markdown/functional_test.py::test
_html_escaping_in_text_but_not_in_code
```

Analysis

The artifact is executable, but violates the test-implied semantic contract regarding HTML handling. The output contains escaped tags (&lt;b&gt;) where the test expects raw <b> to survive normalization, demonstrating a runs-but-wrong pattern.

Fig. 5. Case studies of three representative failure modes in application-level code generation.

1. Executability & Dependency Failures. The generated repository cannot be built, imported, or collected for testing, due to missing, incorrect dependencies or malformed code structure (including syntax-level issues), preventing functional verification from starting.
2. Requirement-Implementation Mismatch. Tests execute, but the implementation violates application functional requirements as operationalized by the test-implied behavioral contract, leading to functional incorrectness.
3. Non-functional Quality Failures. Execution reaches the runtime stage but breaks down due to instability or inefficiency (e.g., exceptions, hangs, or timeouts), indicating robustness, liveness, or performance issues rather than semantic misalignment alone.

Answer to RQ2: *Application-level failures concentrate in a post-executability regime, where the repository runs but fails to satisfy end-to-end functional requirements or runtime non-functional quality attributes. Specifically, once repositories become executable, the dominant barrier is meeting the full set of application-level functional requirements and sustaining reliable runtime behavior, rather than resolving dependencies. Consequently, this yields a characteristic failure progression from buildability to functional requirement violations, and finally to runtime fragility. In other words, in this regime, artifacts often execute but remain incorrect or brittle. Moreover, models mostly fail for different reasons. The key difference is not whether they fail, but what fails first: some produce runnable code that violates functional requirements, while others reach correct functionality but break due to instability, timeouts, or inefficiency. Overall, this shows that bottlenecks differ across models beyond mere executability.*

Table 6. Average per-run benchmark cost (USD) for standard vs. thinking LLMs.

Standard LLMs		
LLM	**Provider**	**Avg. ($)**
Gemini-2.5-Pro	Google	**12.11**
Gemini-3-Pro-Preview	Google	8.14
GPT-3.5-Turbo	OpenAI	0.54
GPT-4o-2024-11-20	OpenAI	0.42
GPT-4-Turbo	OpenAI	2.89
GPT-5	OpenAI	4.04
GPT-5.2	OpenAI	2.67
Claude-Haiku-4.5	Anthropic	0.95
Claude-4.5-Sonnet	Anthropic	11.84
DeepSeek-V3-0324	DeepSeek	1.34
DeepSeek-V3.2	DeepSeek	0.15
Average	–	**4.10**

Thinking LLMs		
LLM	**Provider**	**Avg. ($)**
Gemini-2.5-Pro-Thinking	Google	8.07
GPT-o1	OpenAI	14.05
GPT-o3-2025-04-16	OpenAI	7.72
Claude-3.7-Sonnet-Thinking	Anthropic	**16.63**
DeepSeek-R1	DeepSeek	8.52
Average	–	**11.00**

3.4 RQ3: What is the cost of application-level code generation with current LLMs?

To better understand the practical cost of application-level code generation, we perform a one-round generation for each task without iterative refinement. We compute costs using providers' input and output token pricing and report the mean cost over three independent benchmark runs. Table 6 reports the average per-run monetary cost of evaluating each LLM on RAL-Bench. Our key observations are as follows:

Cost variability. Per-run costs range from 0.15$(DeepSeek - V3.2)to$16.63 (Claude-3.7-Sonnet-Thinking), a difference of more than 100×. This indicates that the cost of application-level code generation can differ substantially depending on the chosen model.

Standard vs. thinking cost. On average, thinking LLMs incur higher per-run costs ($11.00) than standard LLMs ($4.10). Nonetheless, exceptions exist (e.g., Gemini-2.5-Pro-Thinking vs. Gemini-2.5-Pro), suggesting that the observed cost reflects both provider-specific pricing and realized input/output token volumes. The results also show that thinking LLMs consume more tokens to solve the same application-level generation tasks. This directly translates into higher monetary cost. However, this higher cost does not yield a consistent improvement in functional correctness, as shown in Tables 3 and 4. This highlights that enabling LLMs to reason more effectively for application-level code generation remains a key challenge.

Answer to RQ3: *Application-level code generation shows a clear mismatch between cost and performance. Across models, the monetary cost per run differs by more than 100×, but paying more does not reliably lead to better application-level functional correctness. Thinking LLMs usually generate many more tokens, which increases cost substantially. Yet this extra computation does not consistently improve application-level requirement satisfaction.*

Table 7. Impact of automated generation strategies on application-level quality (GPT-5.2). Scores are reported as percentages (%).

Strategy	Func.	Non-func.	Maint.	Sec.	Rob.	Eff.	Res.
S1 Feedback-Driven Self-Repair	37.70	46.10	35.30	78.90	63.70	15.00	20.40
S2 Automated Environment Repair	40.20	49.90	36.10	84.20	67.60	21.00	28.00
S3 Planning-Driven Generation	41.20	49.60	35.60	86.80	69.60	14.40	25.30
Baseline	41.80	57.10	33.90	92.10	77.20	42.80	43.80

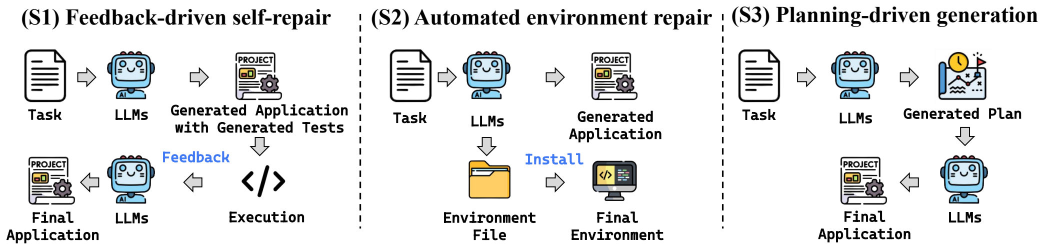


Fig. 6. Overview of the three automated generation strategies evaluated in RQ4.

3.5 RQ4: Are mainstream code generation strategies effective for application-level code generation?

Setup. To examine whether mainstream automated code generation strategies are effective at the application level, we conduct a controlled comparison on GPT-5.2 under identical settings and the same evaluation pipeline. We use GPT-5.2 as the base model because it attains strong baseline performance while remaining cost-efficient, enabling scalable controlled experiments. We instantiate three representative strategy families. **(S1) Feedback-driven self-repair** performs a single post-generation repair using generated test cases. **(S2) Automated environment repair** patches missing dependencies and build artifacts to improve executability. **(S3) Planning-driven generation** distills requirements into an explicit plan before staged implementation. All strategies are evaluated on the same task set using the same functional and non-functional suites as the baseline. Fig. 6 summarizes the end-to-end workflows of the three evaluated strategies.

Results. Table 7 shows that the mainstream strategies we evaluated do not provide reliable gains over the baseline in application-level code generation. The baseline remains the strongest overall, achieving 41.80% functional correctness and 57.10% non-functional quality score. Notably, **all** strategies underperform the baseline on non-functional quality attributes by a large margin. In other words, matching functional correctness is not sufficient. These strategies systematically degrade non-functional quality attributes, leading to worse overall application-level quality. ❶ For **S1 feedback-driven self-repair**, the drop in functional score (37.70%) highlights a mismatch between the feedback signal and application-level failure modes. In snippet-level benchmarks, generated unit tests are often localized and deterministic, and they align closely with the intended contract. This makes one-step repair effective. In contrast, application-level tasks expose cross-module interactions, configuration- and environment-dependent behaviors, and implicit interface contracts. Generated tests in this setting are more likely to be incomplete, mis-specified, or biased toward the model's initial implementation. As a result, the repair step can optimize for an unreliable target. This can induce patch overfitting or regressions that break previously correct components. Consequently, a single feedback-driven repair is not only insufficient but can be harmful. ❷ For **S2 automated environment repair**, this strategy often helps in snippet-level generation because many failures are executability-related, such as missing packages, incorrect imports, or incomplete build files. Fixing these issues can quickly make an otherwise correct solution executable. In application-level generation, however, being executable is only the first step. Most remaining failures arise from missing or incorrect end-to-end behaviors and unstable runtime interactions across modules and configurations. Environment patching does not address these system issues. As a result, it rarely improves overall application-level quality. ❸ For **S3 planning-driven generation**, planning does not yield functional gains over the baseline (41.20% vs. 41.80%). It also significantly reduces non-functional quality score (49.60% vs. 57.10%). This differs from function-level tasks, where planning is effective because the plan maps directly to a small, explicit specification and correctness is dominated by local reasoning. It also differs from many repo-level generation settings, where a plan can reliably improve module decomposition and file scaffolding when evaluation emphasizes structural completeness. In the application-level setting, however, success is governed by distributed, test-implied system contracts, including cross-module interactions, configuration, and runtime behavior. A one-shot plan neither exposes these latent constraints nor prevents plan–execution drift during multi-file implementation, so it fails to improve correctness and can lock in inefficient design choices, which is consistent with the large drop in non-functional scores.

Answer to RQ4: *Mainstream one-shot strategies do not work reliably for application-level code generation. Single-step repair, environment patching, and upfront planning rarely improve results because they do not address the core difficulty of generating a runnable project with proper structure, dependency management and end-to-end executability. Even when these strategies achieve similar functional correctness, they often make the system worse in practice. They can introduce new crashes, slowdowns, or resource waste. To solve application-level tasks, generation must look like a software engineering process, which should clarify missing requirements, apply changes consistently across modules, and check non-functional quality attributes during execution.*

4 Discussion

From runnable snippets to runnable application. `RAL-Bench` reveals a clear gap between current LLM capabilities and application-level requirements. The main challenge is not generating runnable code snippets, but generating a runnable project with proper structure, dependency management and end-to-end executability. Application-level success is governed by cross-module consistency, configuration sensitivity, and runtime interactions, rather than the isolated correctness of individual functions or files.

Limitations of mainstream generation strategies under application-level generation. Our experiments in Section 3.5 further clarify why common code generation strategies stop working at the application level. Feedback-driven self-repair, automated environment repair, and planning-driven generation are all forms of isolated intervention. They apply a single corrective step before or after generation, but they do not maintain application-level functional correctness and non-functional quality attributes.

Next-step directions for application-Level code generation. These findings motivate a shift from one-shot code generation to process-centric code generation, in which generation is treated as a software development workflow rather than a single completion step. Concretely, we identify three testable research directions.

❶ **Application-level Functional Requirement Analysis.** LLMs should analyze end-to-end functional requirements at the application level, identify missing or ambiguous details, and resolve these gaps. Concretely, they should produce an explicit functional requirement list that specifies expected outputs, core behaviors, and edge-case handling. This helps ensure that the implementation remains aligned with the functional requirements and non-functional quality attributes throughout application-level generation.

❷ **Cross-Module Alignment Repair.** It refers to an application-level repair workflow. The workflow uses system tests to assess functional correctness and then performs coordinated multi-file repairs to align all affected modules with the functional requirements. There are two key steps. First, we construct and validate system tests that evaluate application-level functional requirements to produce feedback signals, covering the concise functional requirements rather than snippet-level correctness. Second, we translate the application-level feedback into a coordinated multi-file patch that aligns all involved modules with the requirements.

❸ **Quality-Aware Generation.** LLMs actively keep non-functional quality attributes (robustness, efficiency, resource usage, maintainability, and security) within acceptable bounds during application-level generation. In practice, we treat non-functional quality attributes as constraints during generation. We limit time or memory to avoid unexpected slowdowns and excessive memory use. Also, we can add security or robustness checks during generation, especially at module boundaries (e.g., input validation and exception handling), to expose early warning signs such as error propagation. These approaches can help us detect non-functional quality issues early.

5 Related Work

LLMs for code. Code generation has been long studied with recent approaches focused on pretrained LLMs for code, prompt engineering, retrieval-augmented generation, and agents [22, 23]. Firstly, powerful foundation models for code (e.g., CodeX [5], Code Llama [43], StarCoder [30]) are built by pretraining on large-scale code and natural language corpora, sometimes followed by instruction or domain-specific fine-tuning. On top of these models, prompt engineering techniques such as CodeChain [26], SCoT [28] and MoT [41] aim to improve functional correctness without changing model parameters. To support repository-level understanding, retrieval-augmented methods like REPOFUSE [32], DraCo [8] and GraphCode [44] retrieve relevant files or snippets and feed them as additional context so that generated code better aligns with existing projects. More recently, agentic frameworks embed LLMs into interactive development environments where models can iteratively plan, edit code, run tests and incorporate execution feedback, with multi-agent variants assigning specialized roles to further improve robustness and code quality [35, 45, 46].

Coding benchmarks for LLMs. Benchmarks for code generation are the primary tools for measuring and comparing the coding abilities of LLMs. For functional requirements, researchers have proposed function-level (e.g., HumanEval [7], MBPP [2], EvalPlus [36]), contest-level (e.g., APPS [17], LiveCodeBench [21]), class-level (e.g., ClassEval [11]), repository-level (e.g., RepoEval [38], CoderEval [47], DevEval [29]), and feature-level (e.g., FEA-Bench [31], NoCodeBench [10], FeatBench [6]) benchmarks, all of which rely on execution-based metrics such as pass@k. Beyond functional correctness, several benchmarks, such as EvalPerf [37] and EffiBench [18], begin to assess the efficiency of LLM-generated code by additionally measuring runtime and resource usage. However, these benchmarks still operate mostly at the function or contest level, or focus on modifying existing repositories, and thus do not evaluate end-to-end application generation. In contrast, our work targets application-level code generation, where LLMs must synthesize complete multi-file projects from scratch given only natural-language requirements. We further evaluate the generated programs using black-box system tests that simultaneously capture functional correctness and non-functional properties against real-world reference repositories.

6 Conclusion and Future Work

We introduce `RAL-Bench`, an application-level benchmark grounded in widely used GitHub projects that evaluates end-to-end functional correctness and ISO/IEC 25010-inspired non-functional quality attributes with reference-validated oracles. Our evaluation across 16 frontier LLMs reveals a consistent capability gap between current LLMs and real-world application-level code generation. Functional correctness remains the dominant bottleneck. No LLM exceeds a 45% functional score. By analyzing 446 generated repositories and over 4,500 execution logs, we further quantify that requirement–implementation mismatch and non-functional quality failures account for 82.8% of failures, while executability and dependency failures contribute 17.2%. We find that higher-cost reasoning variants and common code generation strategies do not reliably improve application-level generation and often degrade non-functional quality attributes. Going forward, we will expand `RAL-Bench` to broader projects and settings. Also, we will develop more effective workflows to better align generation with application-level functional requirements and non-functional quality attributes.

7 Data Availability

To ensure the reproducibility of our results and to provide transparency in our research, we have made all related scripts and data publicly available. All resources can be accessed as part of our anonymized artifact, which is available at **https://github.com/Wwstarry/RAL-Bench**.

References

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

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Ivy Belinda Botchway, Akinwonmi Akintoba Emmanuel, Nunoo Solomon, and Alese Boniface Kayode. 2021. Evaluating software quality attributes using analytic hierarchy process (AHP). *International Journal of Advanced Computer Science and Applications* 12, 3 (2021).

[4] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[6] Haorui Chen, Chengze Li, and Jia Li. 2025. FeatBench: Evaluating Coding Agents on Feature Implementation for Vibe Coding. *arXiv preprint arXiv:2509.22237* (2025).

[7] Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782* (2024).

[9] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[10] Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. 2025. Nocode-bench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130* (2025).

[11] 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* (2023).

[12] Jonas Eckhardt, Andreas Vogelsang, and Daniel Méndez Fernández. 2016. Are" non-functional" requirements really non-functional? an investigation of non-functional requirements in practice. In *Proceedings of the 38th international conference on software engineering*. 832–842.

[13] John Estdale and Elli Georgiadou. 2018. Applying the ISO/IEC 25010 quality models to software product. In *European Conference on Software Process Improvement*. Springer, 492–503.

[14] Jiří Franek and Aleš Kresta. 2014. Judgment scales and consistency measure in AHP. *Procedia economics and finance* 12 (2014), 164–173.

[15] David Garlan. 2000. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 91–101.

[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[17] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[18] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems* 37 (2024), 11506–11544.

[19] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[20] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[21] 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* (2024).

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

[23] Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic Software Issue Resolution with Large Language Models: A Survey. *arXiv preprint arXiv:2512.22256* (2025).

[24] 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* (2023).

[25] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. 2015. A case study in locating the architectural roots of technical debt. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 2. IEEE, 179–188.

[26] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992* (2023).

[27] Maikel Leon. 2025. GPT-5 and open-weight large language models: Advances in reasoning, transparency, and control. *Information Systems* (2025), 102620.

[28] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.

[29] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*. 3603–3614.

[30] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[31] Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680* (2025).

[32] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, et al. 2024. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323* (2024).

[33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[34] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556* (2025).

[35] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).

[36] 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. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[37] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450* (2024).

[38] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[39] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 439–451.

[40] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 35–46.

[41] Ruwei Pan and Hongyu Zhang. 2025. Modularization is Better: Effective Code Generation with Modular Prompting. *arXiv preprint arXiv:2503.12483* (2025).

[42] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355.

[43] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[44] Florian Russold and Michael Kerber. 2024. Graphcode: Learning from multiparameter persistent homology using graph neural networks. *Advances in Neural Information Processing Systems* 37 (2024), 41103–41131.

[45] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

[47] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.