Benchmarking LLMs on File System Design and Implementation: The Good, The Bad, and The Ugly

Yuqi Xue*
yuqixue2@illinois.edu
University of Illinois
Urbana-Champaign, USA

Daixuan Li*
daixuan2@illinois.edu
University of Illinois
Urbana-Champaign, USA

Jian Huang
jianh@illinois.edu
University of Illinois
Urbana-Champaign, USA

Abstract

Large Language Models (LLMs) are fundamentally transforming computer system research and development. As we employ LLMs in file system (*fs*) development, it is essential to understand their capabilities, limitations, and operational efficiency for domain-specific tasks. We present *φ-Bench*, an LLM benchmarking framework for *fs*-specific tasks. To facilitate benchmarking, we develop six types of tasks in *φ-Bench*: basic understanding, basic implementation, performance modeling, debugging, optimization, and new feature development. Each type emphasizes different LLM capabilities: instruction following, knowledge recall, reasoning, or coding. To create high-quality tasks while achieving broad coverage with minimal human effort, we develop a new AI-assisted task generation pipeline in addition to expert-written and textbook-adapted tasks. With 505 tasks in *φ-Bench*, we conduct an empirical study with both open source (DeepSeek-V4-Flash, GLM-5.1, and MiniMax-M2.7) and proprietary (Claude-Opus-4.7, GPT-5.2, and Gemini-3.1-Pro) LLMs. Our study discloses the model efficiency for different tasks, causes of failed *fs* tasks, and techniques for mitigating LLM failures. We will open source *φ-Bench* to facilitate public research on using LLMs for *fs* development.

1 Introduction

Recently, large language models (LLMs) have been increasingly employed for computer system research and development, including software engineering, hardware development (RTL coding), OS configuration tuning, kernel debugging, and OS verification [10, 18, 29, 35, 37]. Similar to other domain-specific engineering tasks that benefit from LLMs, developing and maintaining a file system (*fs*) is often labor-intensive and time-consuming, as modern file systems have evolved into large and complex codebases. Hence, it is desirable to ask the question: can we employ LLMs to help reduce the human effort in file system development?

While recent coding models and agents [3, 5, 18, 44] have shown great promise for general software engineering, they mainly target code completion tasks, and application-level development and debugging. It is unclear how well they can handle *fs* design and implementation tasks. A recent work SYSSPEC [32] shows that LLMs can generate *fs* code from human-written specifications, but the LLM still relies on precise specifications written by human developers. It remains unclear whether LLMs can support broader and more difficult *fs* development tasks that require them to explore design trade-offs and make design choices.

To effectively use LLMs for *fs* development, we must first understand their capabilities and limitations in this domain. Although many LLM benchmarks have been developed to evaluate domain-specific engineering tasks, from mathematical reasoning and general software engineering to scientific research [11, 16, 18, 29, 34, 35, 37, 47, 48, 59], they cannot be used to evaluate LLMs on *fs* development tasks, due to the fundamental differences in the domain knowledge, task types, development goals, and evaluation methodologies.

The φ-Bench framework. We develop *φ-Bench* (Phi-Bench)[1], the first benchmarking framework for evaluating LLMs on *fs* design and implementation. *φ-Bench* consists of 505 Q&A and coding tasks (see examples in Figure 1). The tasks cover all major components and functions in a production-grade file system (Table 2) and are organized into six types: basic understanding, basic implementation, performance modeling, debugging, optimization, and new feature development (§3.1). They represent typical tasks in the *fs* development lifecycle. Each type examines different LLM capabilities.

To ensure both high quality and broad coverage with reduced human effort, we develop a new AI-assisted task generation pipeline, alongside human expert-written and textbook-adapted tasks (§3.2). The key principle is to let an LLM draft and review the tasks, while keeping human experts in the loop for reviewing tasks and writing comments to guide the LLM to revise tasks, and making final accept/reject decisions.

To minimize human review effort, each task is generated with two steps (Figure 2). First, we ask the LLM to generate *task outlines*, which serve as "proposals" for easier human review and must be approved before generating the full tasks. The generation is grounded by OS textbooks and Linux kernel source code to ensure the generation is not biased toward the model's own knowledge. Second, we ask the LLM to expand the approved outlines to full tasks, including the task description, reference answer, and test harness. For coding tasks, the test harness provides a self-contained *fs* environment that implements the APIs and data structures for the task. It initializes and warms up the involved *fs* data

*Co-primary authors.

[1] In the name *φ-Bench*, *φ* is pronounced "phi," echoing "file" in file systems.

structures before evaluating the answer. Each full task first goes through an autonomous self-revision loop, where the LLM reviews and revises the task itself under a set of rules. Then, the task will be reviewed: it is either accepted, rejected, or commented by human experts for LLM revision.

All tasks in *φ-Bench* undergo double-blind peer review by human experts to validate task quality. Our AI-generated tasks achieve review scores comparable to expert-written and textbook-adapted tasks, and human reviewers cannot reliably distinguish AI-generated from human-created tasks (§3.3). We will open source *φ-Bench* and its construction pipeline, enabling future studies to adapt the pipeline to build high-quality benchmarks for other domains.

Empirical study with φ-Bench. We use *φ-Bench* to evaluate six frontier LLMs, including both open-source (DeepSeek-V4-Flash [14], GLM-5.1 [2], and MiniMax-M2.7 [39]) and proprietary models (Claude-Opus-4.7 [4], GPT-5.2 [43], and Gemini-3.1-Pro [13]). Our study covers model performance (pass rate), output stability, failure cause analysis, and token/API cost (§4.2–§4.4). We also evaluate representative prompt and context engineering techniques, including RAG, self-review, and error feedback (§4.5). We summarize our key takeaways as follows.

- Current frontier LLMs perform well on simple task types, with the best pass rates reaching 95.8%, 87.4%, and 88.6% on basic understanding, basic implementation, and performance modeling tasks. This is expected as textbook knowledge is covered in the pretraining corpus, and modern LLM training emphasizes coding and reasoning. However, their performance drops sharply on debugging (61.6%), performance optimization (37.6%), and new feature development (41.9%) tasks. Even though a model possesses relevant domain knowledge and general coding and reasoning capabilities, it may not be able to correctly apply them to solve *fs* design and implementation tasks (§4.2).
- The dominant failures are not simple instruction following mistakes or missing domain knowledge. Instead, they are caused by (1) misuse of semantically similar but functionally irrelevant knowledge (19.1% of all failures), (2) logical inference errors (22.3%), (3) missing edge cases due to incomplete reasoning (14.5%), and (4) functionally correct code with suboptimal performance (17.5%) (§4.3).
- For most models, token usage is dominated by LLM reasoning. However, longer reasoning traces do not necessarily mean better performance (pass rate), as they often reflect redundant reconsideration of trivial design decisions rather than deeper analysis of the task. The API cost is generally positively correlated with pass rate: low-cost models such as DeepSeek-V4-Flash can be 16×–107× cheaper than the best-performing models across task types, despite the 3.5%–27.1% lower pass rates (§4.4).
- Because LLM generation is stochastic, a model may solve a task in one trial but fail in another. Resampling (running a task multiple times and picking the best result) improves pass rate by up to 21%. However, 75%–79% of failures remain unresolved, showing that most failures are persistent capability failures rather than unlucky samples (§4.5).
- To mitigate failures, the effective prompt/context engineering techniques include (1) prompting the model to self-review its recalled knowledge, reasoning trace, and solution; and (2) providing compiler/runtime errors and the number of passed/failed unit tests to the model for iterative refinement. These techniques improve the pass rate from 46%/64% to 83%/89% for DeepSeek-V4-Flash/Gemini-3.1-Pro. The remaining failures are dominated by incomplete reasoning and suboptimal implementations (§4.5).

We anticipate *φ-Bench* to benefit the community. First, our study provides practical guidance for both using and improving LLMs in *fs* development. For *fs* developers, our study provides insights for building an agentic workflow with separate agents for gathering knowledge, generating design, writing code, and reviewing. For model builders, our study suggests two optimization goals for future LLMs: systematic reasoning about edge cases and performance-aware code generation. Second, beyond evaluating current models, *φ-Bench* can serve as a dataset for fine-tuning *fs*-specific LLMs. Third, our AI-assisted task construction pipeline can be adapted to other domains to reduce the human effort in building domain-specific benchmarks and datasets (§5). Overall, we make the following contributions in this paper.

- We construct *φ-Bench*, the first LLM benchmarking framework on *fs* design and implementation, with a fine-grained taxonomy of *fs* development tasks and LLM capabilities.
- We design an AI-assisted task generation pipeline to produce high-quality *fs* benchmark tasks with reduced manual drafting effort. Our methodology would inspire the benchmark construction for other domains.
- We conduct a thorough empirical study with *φ-Bench* with popular frontier LLMs. We expect our study results would shed light on the LLM for *fs* research and development.

2 Background and Motivation

2.1 Large Language Models

Large language models (LLMs) are auto-regressive generative models that predict the next word (or sub-word unit, called a *token*) given all preceding tokens [9, 21]. To solve downstream tasks beyond next-token prediction, modern LLMs are trained and evaluated for four basic capabilities:

Instruction following: The LLM needs to understand the task written in natural language and follow the instructions (e.g., design constraints and output format) in the prompt [45, 62].

Knowledge recall: The LLM needs to retrieve correct and relevant facts encoded in its parameters (model weights), such as domain-specific concepts, thus, it can rely on the recalled knowledge to solve the given task [17, 20, 46].

Reasoning: The LLM can perform chain-of-thought (CoT) reasoning, which generates intermediate reasoning tokens to decompose a complex problem into sub-tasks and solve each sub-task before producing the final answer [25, 63].
Coding: Given a detailed design specification or a design derived from its own reasoning, the LLM can generate correct and efficient code for the intended functions [11, 28, 52].

2.2 Using LLMs for File System Development

The long history of file system (*fs*) development has produced large and complex codebases [27, 38, 50, 51, 57, 64]. Taking Linux as an example, *fs* components span across multiple layers of the OS kernel and involve numerous critical functionalities, from the virtual file system (VFS) layer, page cache, core *fs* logic (e.g., metadata and data indexing), to the block layer and device drivers (see our taxonomy in Table 2). As file systems continue to evolve, developing and maintaining them becomes increasingly difficult and costly, requiring intensive human labor and time for adding new features, optimizing performance, and patching bugs [7, 24, 33, 40].

Although coding models and agents [3, 5, 18, 44] have shown great promise for general software engineering tasks, they mainly target code completion and application-level development. It remains unclear how well they can handle *fs* design and implementation. SYSSPEC [32] takes a closer step by using human-written specifications to guide LLMs in generating *fs* code, however, writing such specifications requires substantial domain expertise and human effort, and the model primarily translates a well-specified design into code rather than making design decisions. As a result, how to apply LLMs to *fs* development remains an open problem.

2.3 Benchmarking LLMs on File System Development

To use LLMs for *fs* development, we need to understand their capabilities and limitations in this domain. Recent domain-specific LLM benchmarks cover domains such as mathematical reasoning [16, 34, 47], general software engineering [11, 18], computer architecture [48], hardware (RTL) coding [35, 59], and OS debugging and verification [29, 37]. However, they cannot be directly used to evaluate LLMs for *fs* development, due to the fundamental differences in the required domain knowledge, task types, development goals, and evaluation methodologies. Moreover, most benchmarks focus on gathering a collection of domain-specific tasks and mainly aim to provide an overall accuracy metric for the model under evaluation, rather than a fine-grained breakdown of the model's capabilities and limitations.

In this work, we aim to build a new benchmarking framework to systematically quantify an LLM's capabilities and limitations on *fs* design and implementation. We hope that our study will inspire future research and development of using LLMs for *fs* development, such as training specialized models and developing customized LLM agents.

Table 1. Model capabilities tested by each task type in *φ-Bench*. "✓" denotes that a capability is tested by a task type.

I: Instruction Following K: Knowledge Recall
R: Reasoning C: Coding

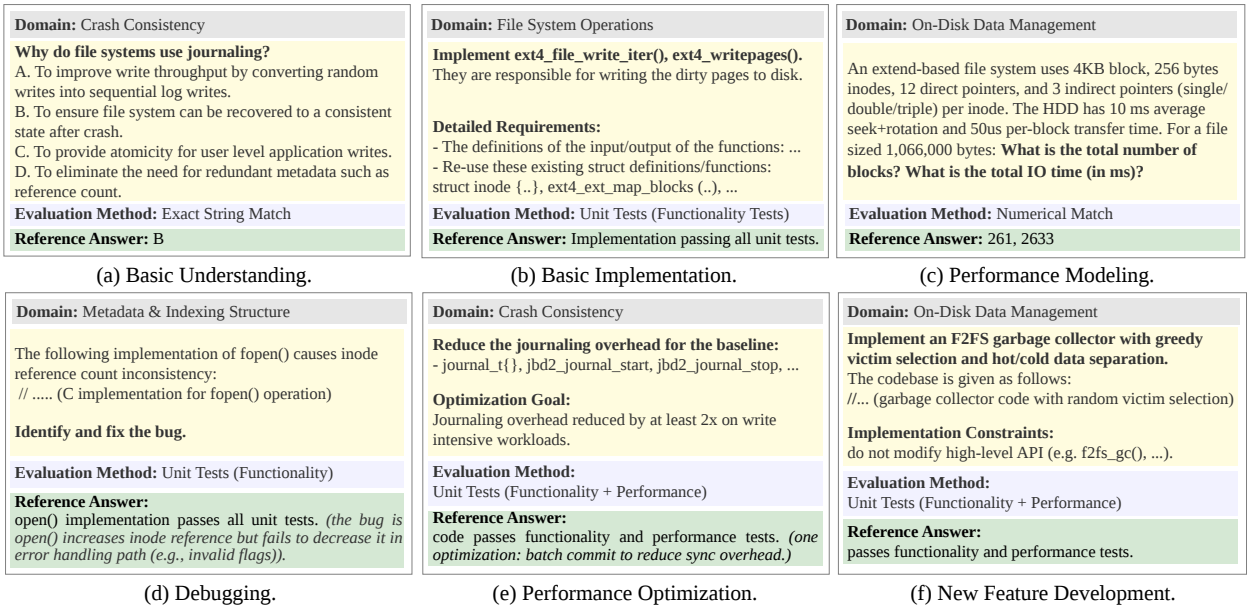
Task Type	I	K	R	C
Basic Understanding (BU)	✓	✓	–	–
Basic Implementation (BI)	✓	✓	–	✓
Performance Modeling (PM)	✓	✓	✓	–
Debugging (DE)	✓	✓	✓	✓
Performance Optimization (PO)	✓	✓	✓	✓
New Feature Development (NF)	✓	✓	✓	✓

3 *φ-Bench* Construction Methodology

We build *φ-Bench*, the first benchmark to evaluate LLM capabilities in *fs* understanding, design, and implementation. We first present a taxonomy to characterize the task types in the *fs* development lifecycle and the model capabilities each task type requires (§3.1). Based on our taxonomy, we create textbook-adapted, expert-written, and AI-generated tasks. We build an AI-assisted task generation pipeline grounded by OS textbooks and Linux kernel code and supervised by human experts, which scales the benchmark construction while guaranteeing task quality (§3.2). All tasks are peer-reviewed by human experts. Our AI-generated tasks achieve quality comparable to human-created tasks (§3.3).

3.1 *φ-Bench* Taxonomy

To analyze LLM capabilities and limitations in *fs* development, *φ-Bench* organizes tasks along two axes: which stage in the *fs* development lifecycle it represents and what model capabilities it tests. As shown in Table 1, the lifecycle axis (rows) yields six task types that cover different development activities, such as debugging, developing new features, and optimizing performance. The capability axis (columns) shows whether each type assesses instruction following, knowledge recall, reasoning, coding, or a combination of capabilities.
Basic Understanding (BU) tasks (Figure 1a) test whether a model has the basic *fs* knowledge (i.e., knowledge recall) needed before performing complex design and implementation tasks. They use multiple-choice, true/false, or short calculation formats, allowing the benchmark to isolate knowledge recall from reasoning and coding capabilities.
Basic Implementation (BI) tasks (Figure 1b) test an LLM's ability to translate a well-specified design into C code. The prompt provides step-by-step workflow/pseudocode, function interfaces, available APIs, and implementation constraints. The model must write code and pass unit tests. This separates implementation (i.e., coding) skill from design reasoning. *φ-Bench*'s BI tasks differ from generic coding tasks because the model needs basic *fs* concepts to generate the correct implementation, such as how each field in an inode should be queried or modified to implement the provided design.



(a) Basic Understanding. (b) Basic Implementation. (c) Performance Modeling.

(d) Debugging. (e) Performance Optimization. (f) New Feature Development.

Figure 1. Representative tasks in *φ-Bench*.

Table 2. Distribution of tasks by type, domain, and source.

		Count	%
Type	Basic Understanding	62	12.3%
	Basic Implementation	57	11.3%
	Performance Modeling	74	14.7%
	Debugging	85	16.8%
	Performance Optimization	124	24.6%
	New Feature Development	103	20.4%
Domain	Virtual File System	50	9.9%
	File System API	50	9.9%
	Memory Management	43	8.5%
	Page Cache	43	8.5%
	File System Core	264	52.3%
	File System Operations	35	6.9%
	Metadata & Indexing	62	12.3%
	On-Disk Data Layout	36	7.1%
	Concurrency	44	8.7%
	Crash Consistency	37	7.3%
	Data Integrity	50	9.9%
	Block Layer	86	17.0%
	I/O Scheduling	46	9.1%
	Virtual Block Devices	40	7.9%
	Device Driver	62	12.3%
Source	AI-Generated	305	60.4%
	Human-Created	200	39.6%
	Expert-Written	93	18.4%
	Textbook-Adapted	107	21.2%
Total		**505**	**100%**

Performance Modeling (PM) tasks (Figure 1c) assess a model's quantitative reasoning capability about *fs* performance. Tasks describe a workload pattern, hardware configuration, or code snippet, and the model must analyze performance, compare designs, and make design decisions.

Debugging (DE) tasks (Figure 1d) test the model's ability to identify and fix bugs. The model is given a buggy code snippet and a bug report that includes the inputs that trigger the bug and the expected vs. observed behavior. The model must understand the bug report, reason about the root cause of the bug, and implement the correct patch.

Performance Optimization (PO) tasks (Figure 1e) require the model to improve an existing implementation to satisfy a performance target while preserving correctness. Unlike PM tasks, which focus on reasoning about the optimization strategy, PO tasks also require the model to implement the optimization. Our tasks cover diverse metrics, such as latency, throughput, I/O amplification, and disk-space utilization.

New Feature Development (NF) tasks (Figure 1f) ask the model to add new functionality to an existing file system, such as a new POSIX API or data management policy. Unlike BI tasks, NF tasks state the new feature's goal rather than a step-by-step implementation plan. For performance related features (e.g., adding hot/cold data separation mechanism), unit tests include correctness and performance tests. NF tasks involve understanding the required feature, reasoning about functionality and performance, and generating correct code.

Overall, our taxonomy enables an ablation study of different capabilities of LLMs. BU, BI, and PM isolate knowledge recall, reasoning, and coding, respectively. DE, PO, and NF test the combination of these capabilities in various end-to-end development tasks. Instruction following is shared by all categories because each task requires parsing the task specification and producing the specified output format.

φ-Bench also organizes tasks by domain (Table 2). These domains cover major layers in the *fs* stack. We also include relevant OS layers: page cache, block layer, and device drivers, as developing a file system often requires interacting with them. The domain/task-type distribution roughly reflects the topic coverage in OS textbooks and Linux source code, as they grounded our benchmark construction (§3.2).

3.2 Benchmark Construction

φ-Bench needs to achieve two goals: high task quality and broad coverage of domains and task types. A standard way to construct an LLM evaluation benchmark is to crowdsource from a large number of human authors [47, 48]. However, while human experts can write high-quality tasks, this method is both time-consuming and labor-intensive. Another way is to adapt from textbook exercises and exams. However, these exercises are often too simple and not designed for testing coding capability. To achieve both high quality and broad coverage while minimizing human effort, *φ-Bench* introduces a new AI-assisted task generation pipeline in addition to the expert-written and textbook-adapted tasks. Table 2 reports the number of tasks from each source.

3.2.1 Textbook-Adapted Tasks. We use LLM to extract all *fs*-related exercises from two popular OS textbooks [6, 55]. We include only multiple-choice, true/false, calculation, and coding exercises. For Python coding exercises, we prompt the LLM to convert them into Linux kernel-style C code. We omit open-ended free-response questions because deterministic test harnesses cannot automatically evaluate them[2].

3.2.2 Expert-Written Tasks. Five graduate students who have experience in systems research (i.e., "human experts") manually created the 93 tasks in *φ-Bench*, taking ≈580 person-hours in total. All tasks are peer-reviewed and revised in a double-blinded manner to cross-validate task quality.

The human-based workflow has five steps: (1) Pick a *fs* component/functionality based on the author's own knowledge or related material such as OS textbooks, real *fs* code, or research papers. (2) Formulate a task outline, including the core idea, evaluation method, and expected answer. (3) Based on the outline, draft the task, reference solution, and test harness (for coding tasks). (4) Verify the reference solution passes the tests and dry-run the task with LLMs to find potential issues, such as ambiguity, leaked answer cues, or buggy test harness. (5) Revise the task to fix all issues.

3.2.3 AI-Generated Tasks. Our AI-assisted task generation pipeline is inspired by the above human workflow. The key principle is to let an LLM agent perform the labor-intensive drafting/revision while keeping humans in the loop for reviewing tasks, writing comments to guide AI to revise the tasks, and making final decisions to accept/reject a task. Figure 2 shows the pipeline, which consists of three stages.

[2]Though it is possible to use LLM to evaluate free-response questions (i.e., LLM-as-a-judge), this introduces biased or incorrect evaluation results [58].

(1) Prepare knowledge corpus. We first construct the *fs* knowledge corpus, which will be fed to the LLM agent in later stages. It consists of textbook chapters and sections [6, 55], which provide conceptual knowledge and design principles, and code snippets from Linux kernel source code [31], which provide real-world coding examples. We curate a knowledge corpus for each domain in Table 2, and each is curated manually for factual correctness and relevance.

This reduces hallucination and minimizes bias by providing external factual knowledge. Without it, the LLM would generate tasks only from its own knowledge; consequently, the resulting task will only cover what the model already knows and fail to evaluate the model objectively.

(2) Generate task outlines. For each target domain and task type, the LLM agent generates a set of candidate *task outlines*. These outlines serve as "proposals" that must be approved by human supervisors before being expanded to full tasks. Each task outline contains the core task idea, the intended evaluation method, the high-level reference answer, and the knowledge corpus entries used to derive this task. An outline is much more concise than a full task and hence requires minimal human effort to review (typically ≤200 words need to be reviewed for each outline). Generating an outline before the full task is critical for improving final task quality and reducing human efforts, as we can quickly identify "promising" outlines that are more likely to yield high-quality tasks.

We prompt the LLM agent to generate at least 20 outlines for each domain and task type. The generation is grounded by the corresponding knowledge corpus, the task outline template, and 3–5 example human-written outlines. Each outline is reviewed by a human expert for correctness, relevance, and duplicates. An outline is either accepted/rejected, or the reviewer will write a revision prompt to guide the AI to revise it. This *supervised revision* process is iterative, and it finishes when all outlines are either accepted or rejected.

(3) Generate full tasks. For each accepted task outline, the LLM agent expands the task idea into a *full task*, which includes multiple files: the full task description that will be fed to the LLM-under-test (`input.txt`), the unit tests written in C code if this is a coding task (`test_harness.c`), the reference answer (`ref_answer.py`), and the evaluation script that takes the model output, invokes the unit tests, and gives the final score (`evaluation.py`).

For each coding task, the test harness provides a self-contained *fs* environment that implements the APIs and data structures required by the task. It warms up all involved *fs* state, such as in-memory caches and on-disk structures, before running the unit tests. The harness code is grounded by real Linux kernel code in the knowledge corpus and validated by human experts during task review.

For each generated full task, the LLM agent first launches an autonomous *self-revision* loop (1) to examine the task with a set of rules: the task description matches the evaluation script, the evaluation script and the reference answer

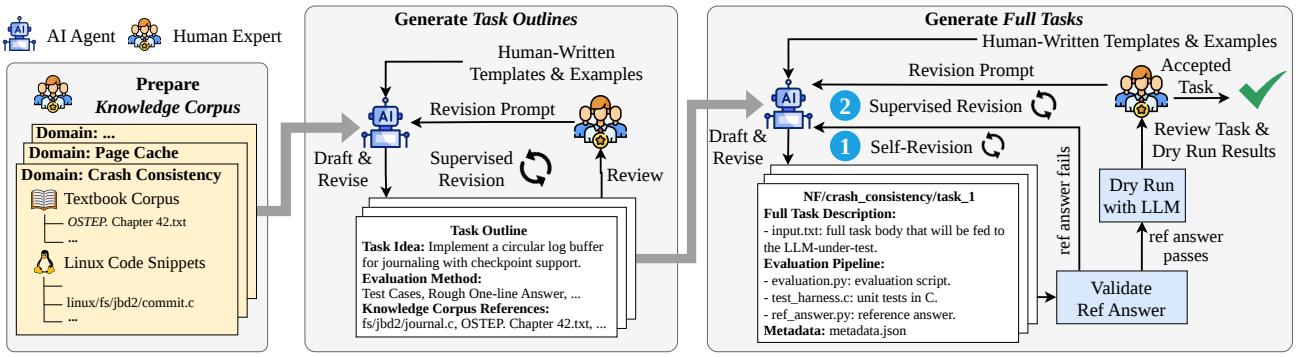


Figure 2. AI-assisted task generation pipeline in *φ-Bench*.

Table 3. Review rubrics used by human experts in *φ-Bench*'s benchmark construction and validation.

Technical Soundness	**Clarity**
1: Complete hallucination	**1:** Unintelligible
2: Contains factual errors	**2:** Idea present, logic unclear
3: No issue	**3:** Mostly clear
	4: Very clear
Technical Depth	**Overall Quality**
1: Trivial basic recall	**1:** Reject
2: Shallow familiarity	**2:** Major revision
3: Moderate knowledge	**3:** Minor revision
4: Multi-step reasoning	**4:** Accept as is
5: Cross-layer reasoning	**5:** Best task candidate

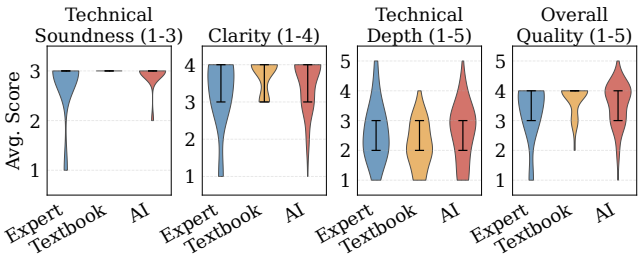


Figure 3. Review scores of tasks in *φ-Bench*. Vertical lines show the 25th/75th percentiles. Textbook-adapted tasks always score 3 in technical soundness, as they are adapted from publication-quality exercises.

Table 4. Double-blind test results for distinguishing between human-written and AI-generated tasks. Human testees are asked to annotate the source of 90 sampled tasks.

(a) Annotation accuracy.

Source	**Accuracy**
Human	50.5 %
AI	48.9 %
Overall	**49.5 %**

(b) Confusion matrix.

Source	**Human Annotation**	
	Human	AI
Human	**20.0 %**	19.6 %
AI	30.9 %	**29.5 %**

must compile and run, the reference answer passes all unit tests, and the task description does not leak answer cues. The agent will revise a task iteratively to address these issues.

After self-revision, a task undergoes *supervised revision* (2). We dry-run the task with an LLM, after which human supervisors review both the task and the dry-run results. The dry run ensures task failures reflect incorrect answers rather than task flaws, such as ambiguous instructions, buggy evaluation scripts, or faulty unit tests. Table 3 shows the review rubric. Note that technical depth is not considered for task acceptance; it is collected to validate that *φ-Bench* covers different difficulty levels.

3.3 Benchmark Quality Assessment

To coordinate the review process, we built a website to collect expert-written tasks/reviews and version-control tasks across review/revision cycles. All tasks are reviewed by ≥3 experts in a double-blinded manner. Final inclusion requires unanimous acceptance from all reviewers, and any revised task must undergo another review cycle. We will publish the website to crowdsource more tasks and public reviews.

Figure 3 compares the review scores of tasks across all three sources before any revision (all non-rejected tasks are revised and ultimately accepted). Every domain in *φ-Bench* contains at least three non-AI-generated tasks for a fair comparison. Since our pipeline filters out poor task outlines early, AI-generated tasks in the first revision iteration already achieve quality comparable to human-created ones. While expert-written and AI-generated tasks span a broad range of technical depth, textbook-adapted ones are simpler, as they focus on basic concepts for educational purposes rather than full-fledged design & implementation problems.

To further validate *φ-Bench*'s quality, we conducted a double-blind experiment asking four graduate students working on systems research to classify 30 randomly selected tasks per source (90 sampled tasks in total) as either human-

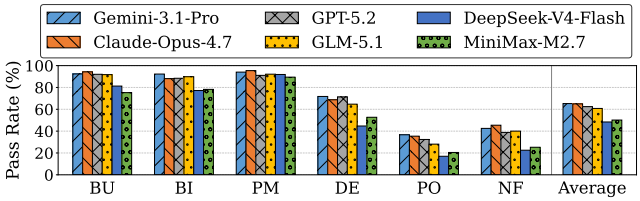


Figure 4. Pass rate of models across task types.

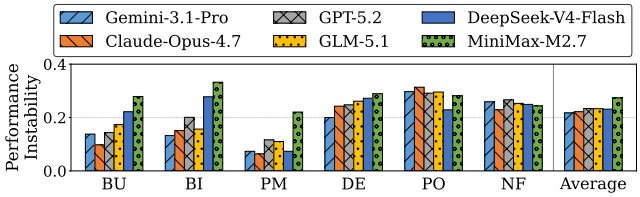


Figure 5. Performance instability of models across task types. Instability is measured as the pooled standard deviation [42] of 5 repeated trials, with pass scored as 1.0 and fail as 0.0. Lower instability values indicate the model consistently performs well or poorly on the target task type.

or AI-generated. Table 4 shows that testers correctly identified the source with roughly 50% accuracy — equivalent to random guessing. This confirms that our AI-generated tasks are indistinguishable from those created by human experts.

4 Empirical Study with *φ-Bench*

In this section, we use *φ-Bench* to evaluate six frontier LLMs. Our results show that current LLMs still struggle with the most complex design and implementation tasks, and that prompt and context engineering methods improve but do not close this gap. We highlight our key takeaways in boxes.

4.1 Experimental Setup

We evaluate both open-source (DeepSeek-V4-Flash [14], GLM-5.1 [2], and MiniMax-M2.7 [39]) and proprietary models (Claude-Opus-4.7 [4], GPT-5.2 [43], and Gemini-3.1-Pro [13]). For GLM-5.1 and MiniMax-M2.7, we use Together AI API [60]. For other models, we use their official APIs. All models use default temperatures and maximum reasoning efforts. The experiments run on a server with Intel Xeon E5-2687W v4 (24 cores) and 96 GB memory. To account for sampling variance of LLMs [19], we run five independent trials per task and report their average results, unless otherwise specified. To account for the task count difference across task types (Table 2), the overall pass rate is computed as the average of the per-type pass/fail rate weighted by the task counts.

4.2 Overall Model Performance

Figure 4 shows the average performance (pass rate) on different task types. All models perform well on BU (up to 95.8%), BI (up to 87.4%), and PM (up to 88.6%) tasks. This suggests that frontier models have strong textbook-level *fs* knowledge, can implement simple and well-specified *fs* functionality, and can analytically model *fs* performance when the task is abstracted out from a complex codebase. The results are expected: OS textbooks are well-covered in pretraining corpora (BU), modern frontier models are explicitly optimized for coding (BI), and they can apply basic quantitative reasoning to known *fs* concepts (PM).

Takeaway (T1): LLMs perform well on knowledge recall (BU), simple coding (BI), and performance modeling (PM). This is consistent with how they are trained: pretraining covers OS textbook knowledge, and modern training pipelines emphasize coding and reasoning.

All models perform substantially worse on DE, PO, and NF tasks, achieving up to 61.6% on DE, 37.6% on PO, and 41.9% on NF. These tasks require the model to jointly recall *fs* knowledge, reason about functionality and performance, and implement correct code. The difficulty is pronounced for PO and NF, where the model must preserve correctness while changing the design to satisfy performance and functionality requirements. We analyze the failed cases in detail in §4.3.

Takeaway (T2): LLMs struggle with complex file system tasks that require multiple capabilities. Possessing domain knowledge, reasoning, and coding capability does not guarantee a model can apply them correctly to solve concrete design and implementation problems.

LLMs are inherently stochastic: the same input prompt can produce different outputs across runs, so a model may answer a task correctly in one trial but fail in another. We quantify the impact of output instability of LLMs in Figure 5.

A model that achieves a higher pass rate also answers tasks more consistently. This is because when an LLM knows the correct answer, it tends to produce next-token distributions heavily skewed toward the correct answer, so the token corresponding to the correct answer has substantially higher probability to be selected than tokens corresponding to incorrect answers [19]. Intuitively, the model is more "confident" when it knows the answer. When the LLM is uncertain about the answer, it produces a distribution that is more uniform across correct and incorrect tokens, so it "guesses" a random answer. This trend is obvious for BU, BI, PM, and DE tasks. For the most difficult PO and NF tasks, all models suffer from low pass rates and have relatively high instability.

Takeaway (T3): Models with higher pass rates produce more stable and consistent outputs across multiple trials due to the stochastic sampling nature of LLMs.

Table 5. Definition of failure types across all failed trials for all models.

Code	Capability	Failure Type	Definition
F1	Instruction Following	Specification Violation	The prompt states an explicit constraint, such as an output format, a fixed function signature, or a specific API usage, but the model's response violates it and cannot be parsed, compiled, or evaluated.
F2	Knowledge Recall	Irrelevant/Misleading Knowledge	The model recalls a piece of knowledge that is factually correct but cannot be directly applied to solve the task, and this recalled fact misleads the reasoning process, leading to a wrong final solution.
F3	Knowledge Recall	Incorrect or Missing Knowledge	The model hallucinates or misses *fs* facts, making it impossible to reason about the correct solution based on these facts.
F4	Reasoning	Logical Inference Error	The model starts from a correct and sufficient premise, but makes a logical inference error, such as an incorrect causal relation, leading to a wrong final answer.
F5	Reasoning	Incomplete Reasoning	The model starts from a correct and sufficient premise, but misses some necessary reasoning steps. As a result, the final solution misses critical edge cases, such as a boundary check or error handling path.
F6	Coding	Reasoning-Solution Mismatch	The reasoning process concludes with a valid solution, but the submitted answer does not follow the proposed solution, such as using a different algorithm or data structure.
F7	Coding	Suboptimal Implementation	The functional design and optimization strategies are correct, but implementation introduces unnecessary overhead, such as redundant `malloc()/free()` and suboptimal choice between `qsort()` vs. insertion sort.
F8	Coding	Generic Coding Bugs	The code compiles, but fails at runtime due to generic, non-*fs*-specific coding bugs, such as a segmentation fault, double free, use-after-free, or heap corruption.

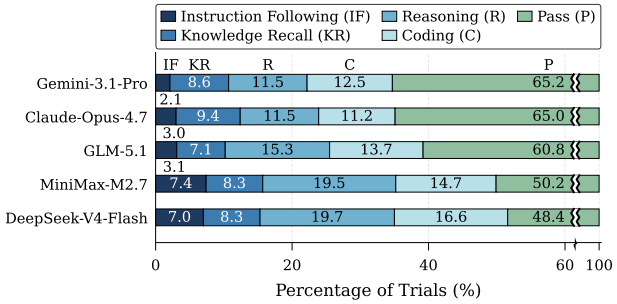


Figure 6. The percentage of tasks failed due to different failure reasons listed in Table 5. We exclude GPT-5.2, as its reasoning process is not available.

4.3 Failure Analysis

In this section, we analyze how and why a model fails to correctly solve a task. We manually inspected every failed trial and categorized their failures into 8 types (see Table 5). We omit GPT-5.2 as its API does not expose its reasoning process. Figure 6 shows the failure type breakdown.

Instruction following (F1). Instruction following failures are the least common failure type (2.1%–7.4% of all trials). Most of them are due to model output instability, as resampling resolved 72%–98% of them (§4.5). The rest reveal certain model-specific biases: for example, Claude-Opus-4.7 includes non-existent headers in a few tasks, and DeepSeek-V4-Flash sometimes includes Markdown syntax in the generated code, causing compilation failures. These failures can be solved by providing error messages for iterative refinement (see §4.5).

Takeaway (T4): Instruction following failures are the least common failure type in all failure types (2.1%-7.4%). These failures are due to output instability or model-specific biases. They can be resolved by retrying or feeding error messages back to the model for revision.

Knowledge recall (F2-F3). Knowledge recall errors account for 7.1%–9.4% of all failures. They fall into two categories.

As shown in Figure 7a, the majority (92%–99%) of knowledge recall failures are due to the model recalling irrelevant or misleading facts (F2). Even if the model has the correct knowledge, it may not be able to precisely recall it when needed. Instead, it may recall semantically similar knowledge, such as a memorized algorithm or code pattern, and apply it directly without reasoning. For example, in a BU task on deadlock detection for file locks, Gemini-3.1-Pro correctly recalls Linux's efficient chain-based deadlock detection mechanism in `fs/locks.c`. However, in an NF task requiring the same deadlock detection within a file lock manager, the model recalls a costlier dependency graph DFS algorithm, failing the performance test. To alleviate this, we can ask the model to carefully verify recalled knowledge before reasoning about the answer (see §4.5).

Takeaway (T5): The majority of knowledge recall failures (92%-99%) are due to recalling semantically similar but unsuitable knowledge, which misleads the reasoning process and leads to an incorrect solution.

In Figure 7a, 0.5%–8% of knowledge recall failures are due to incorrect or missing knowledge (F3). The missing knowledge is about specific production-level *fs* implementations rather than textbook-level generic design principles. For example, MiniMax-M2.7 knows that a write() to a file opened with O_DIRECT (i.e., direct I/O) bypasses the page cache, but it does not recall that in ext4, the write also invalidates stale page cache data. 19% of missing knowledge failures

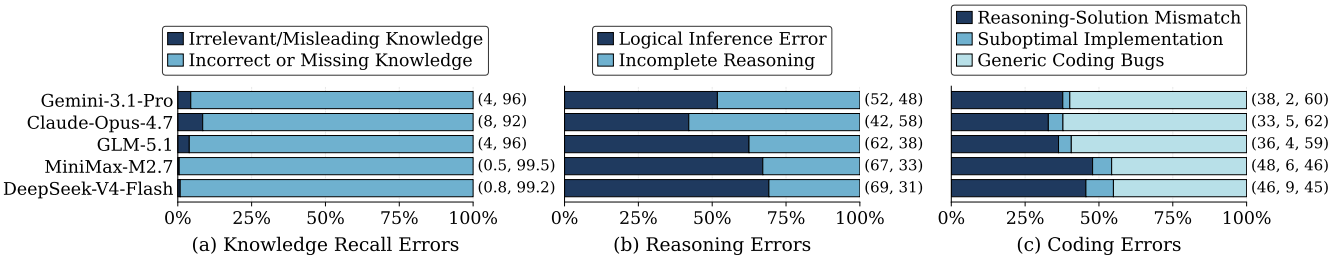


Figure 7. Failure type breakdown based on the taxonomy in Table 5. We exclude GPT-5.2, as its reasoning process is unavailable.

are resolved by resampling, indicating that they are due to output instability; the remaining ones can be addressed by providing the missing knowledge (e.g., via RAG) to the model to ground its output, or asking the model to verify its recalled knowledge and recall again when necessary (see §4.5).

> **Takeaway (T6):** The missing knowledge in frontier models is production-level file system implementation facts that a human developer acquires in real development experience, rather than textbook-level design principles.

Reasoning (F4-F5). Reasoning failures mainly fall into two types: either the model makes a wrong logical deduction in its reasoning process given a correct knowledge context (F4), or its reasoning is incomplete, which typically leads to missing edge cases in the final solution (F5). As shown in Figure 7b, both are significant within reasoning failures.

Logical inference error (F4) accounts for 42%–69% of all reasoning failures, as shown in Figure 7b. For example, in an NF task that requires implementing XFS-style inode allocation, GLM-5.1 first proposes a valid design that manages inodes in groups, each group reserving a fixed-size inode-number space. However, it then rejects this valid design, deeming the reserved space too large without calculating it.

We observe that some logical errors may be fixed later as the model performs iterative refinement of the solution in its reasoning process. In fact, across all trials where the reasoning process involves logical inference errors, 55% of them ultimately reach a correct final solution, despite significant token consumption (see §4.4). To reduce logical inference errors, prompting the model to do more self-refinement is a feasible way to improve reasoning quality (see §4.5).

> **Takeaway (T7):** 42%–69% of reasoning failures are due to logical inference errors. However, in 55% of the reasoning traces containing these errors, the model successfully self-corrects by iteratively verifying and refining its solution.

31%–58% of reasoning failures are due to incomplete reasoning (F5), which typically leads to missing edge cases in the final solution, even though the given context and recalled knowledge are sufficient for the model to derive all edge cases. For example, in a task on implementing a POSIX read(), GLM-5.1 does not reason about the boundary case where the read offset is greater than the file size.

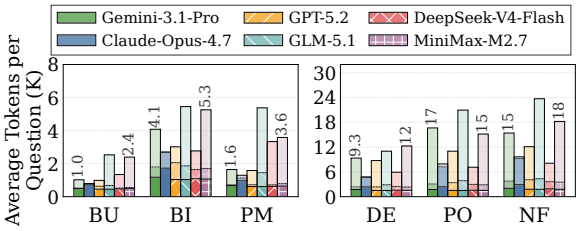


Figure 8. Token usage per task by model and category (stacked bars: input at bottom, response in the middle, and thinking at the top). We mark the absolute values for Gemini-3.1-Pro and MiniMax-M2.7 for reference.

This reveals a fundamental limitation of chain-of-thought (CoT) reasoning: CoT can help decompose a complex task into smaller, more solvable steps, but it does not guarantee that the model will systematically cover all edge cases [8, 9, 63]. Prompting the model to explicitly examine edge cases and verify the proposed solution can help alleviate this type of failure (see §4.5). To systematically eliminate this failure type in practice, we should leverage mature debugging, testing, and formal verification tools for file systems [24, 41, 54].

> **Takeaway (T8):** 31%–58% of reasoning failures are due to incomplete reasoning, which leads to missing edge cases in the generated code. This is an intrinsic limitation of CoT reasoning: CoT can decompose a complex task into smaller, more solvable steps, but does not guarantee coverage.

Coding (F6-F8). Coding failures happen when a model recalls sufficient knowledge and performs correct reasoning to derive a valid design, but fails to implement it correctly. We categorize these into three types:

First, 33%–48% of coding failures are due to a mismatch between a correctly-reasoned solution and its implementation (F6), as shown in Figure 7c. For example, in an NF task involving a write buffer queue, Claude-Opus-4.7 correctly reasons that the write counter should be incremented only for new queue entries, not for overwritten ones. However, the generated code still increments the counter on overwrite.

Second, 45%–62% of coding failures are due to suboptimal implementations (F7). For NF and PO tasks, which have performance requirements, the model may output a functionally

and algorithmically correct solution, but fail to implement it efficiently, such as sorting an already-sorted array or calling memcpy() twice on unchanged data.

Both F6 and F7 failures result from the stochastic nature of LLMs, which prevents any guarantee of perfect code generation. However, these failures can often be mitigated using repeated sampling and self-code review (see §4.5).

Finally, 2%–9% of coding failures are generic, non-*fs*-specific errors (F8), such as null/invalid pointer dereferences, double frees, use-after-free bugs, uninitialized variables, and off-by-one errors. Since these often lead to compilation or runtime errors, a straightforward fix is to provide error messages to the LLM for refinement (see §4.5).

> **Takeaway (T9):** Even if a model derives a correct solution in its reasoning process, it may fail to generate correct and efficient code for the solution, due to the intrinsic probabilistic nature of LLMs. Such coding failures can often be addressed by repeated sampling, self-code review, or providing compilation or runtime error feedback.

4.4 Cost Analysis

In this section, we analyze the token usage and API cost (i.e., monetary cost) across models and task types.

Token usage. As shown in Figure 8, reasoning tokens account for 79.8–91.4% of output tokens in all models, except for Claude-Opus-4.7 (only 9.5%). The reasoning traces of open-source models[3] reveal substantial redundancy. Even when a model solves a task correctly, it often overthinks the problem by repeatedly revisiting decisions after reaching a valid solution. For example, MiniMax-M2.7 spends over 47K tokens in a single trial, repeatedly reconsidering trivial factors like JSON-format constraints, whereas the core design decision only consumes 4.2K reasoning tokens. In contrast, Claude-Opus-4.7 only spends 251 tokens on the same task to derive the correct solution, indicating that the "quality" of reasoning matters more than the length.

> **Takeaway (T10):** For most models, the token consumption is dominated by reasoning tokens (79.8–91.4%). Open-source models overthink by repeatedly revisit trivial design decisions, leading to high token usage. As a result, more reasoning tokens do not necessarily lead to a higher pass rate; rather, the "quality" of reasoning matters more.

API (monetary) cost. Figure 9 shows the Pareto frontier of pass rate vs. cost for each task type[4]. On each task type, models' end-to-end API costs are positively correlated with their performance (i.e., pass rates). For example, DeepSeek-V4-Flash is 16×–107× cheaper than the best-performing model across task types, despite its 3.5%–27.1% lower pass rate.

> **Takeaway (T11):** In general, the API cost of a model is positively correlated with its performance (i.e., pass rate).

[3]Because proprietary models such as Claude-Opus-4.7 and Gemini-3.1-Pro encrypt their original reasoning traces and expose only summaries, we focus on open-source models for the reasoning token usage analysis.

[4]We use the following API prices per 1M input/output: $5/$25 (Claude-Opus-4.7), $2/$12 (Gemini-3.1-Pro), $1.75/$14 (GPT-5.2), $0.98/$3.08 (GLM-5.1), $0.30/$1.20 (MiniMax-M2.7), $0.14/$0.28 (DeepSeek-V4-Flash).

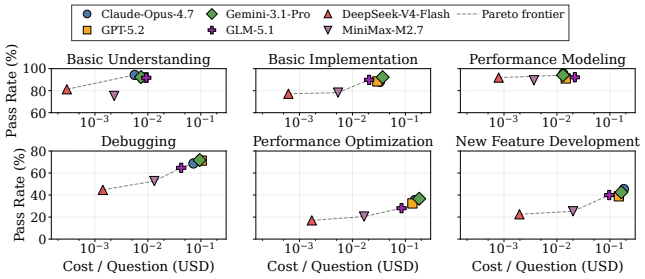


Figure 9. API cost vs. pass rate per task type.

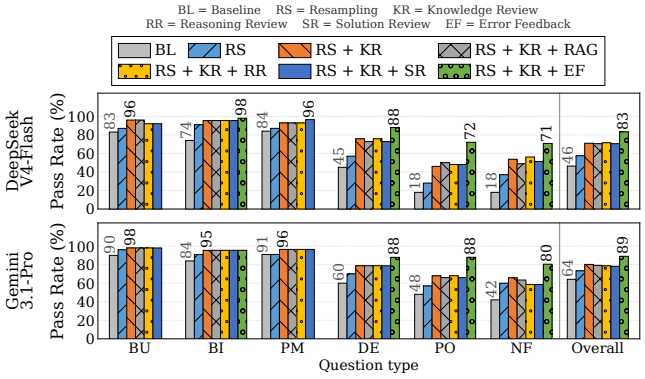


Figure 10. Pass rate improvement breakdown for each method in §4.5. We show two models as examples. The observations also generalize to other models. In the legend, "A + B" means we apply methods A and B together.

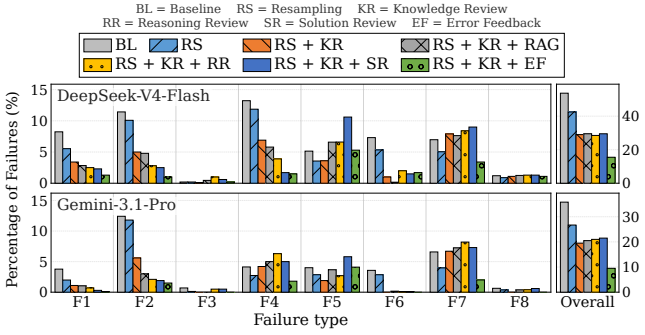


Figure 11. Percentage of failed trials after applying different combinations of prompt/context engineering methods.

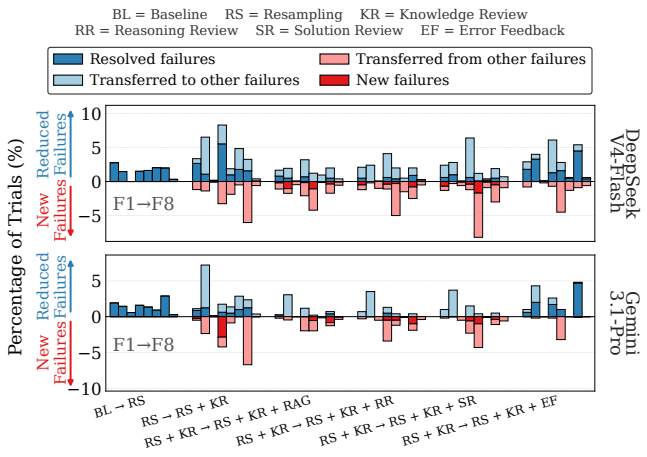


Figure 12. Failure type transitions after adding each method. For each transition $A \rightarrow B$, the bars show the changes in F1–F8 failures (Table 5). Upward bars count failures in A that disappear in B: the task either passes ("Resolved failures") or fails with another failure type ("Transferred to other failures"). Downward bars count failures in B that were absent as in A: they either come from another type ("Transferred from other failures") or from a task that previously passed in A ("New failures"). Percentages are over all trials.

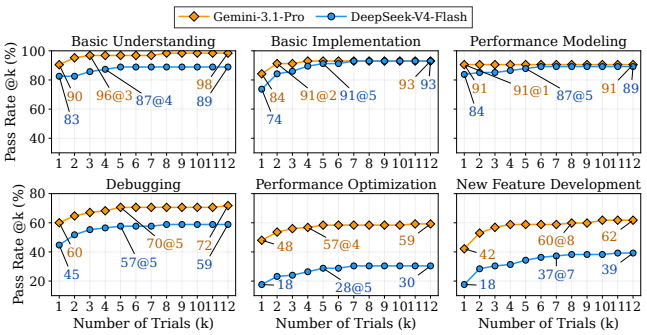


Figure 13. Overall pass rate at k resampling runs (i.e., up to k attempts for each task). We label the saturation point ("{pass rate}@{k runs}") at which more attempts improve the pass rate by ≤2% (equivalent to 1–2 tasks per task type).

4.5 Mitigating Failures with Prompt Engineering and Context Engineering

To mitigate the failures discussed in §4.3, prompt engineering and context engineering [1, 9, 17, 25, 63] are popular approaches that improve LLM performance at inference time without updating model weights. In this section, we experiment with various methods that provide additional instructions or external knowledge to the model and evaluate their effectiveness in addressing each failure type:

- **Resampling (RS) [61]**: We repeatedly run each task for up to 12 times, stopping once the model produces a passing solution. Resampling beyond 12 trials does not help solve more tasks. This mitigates model output instability (see Figure 5), targeting F1, F6, and F7 failures (see §4.3).
- **Knowledge Self-Review (KR)**: We prompt the model to carefully verify that Factual knowledge in its reasoning process is correct and relevant to the task. This targets F2.
- **Retrieval-Augmented Generation (RAG)**: We leverage RAG to provide extra knowledge to the model, targeting F3 failures. We build two RAG databases: (1) an OS textbook database [6, 55], and (2) an example code database that encodes *fs*-related code from Linux [31]. We retrieve the top-5 items from the two databases using hybrid BM25+semantic search [22, 23, 36, 49] (a state-of-the-art RAG approach). When applying KR+RAG together, we prompt the model to verify both the recalled and retrieved knowledge.
- **Reasoning Self-Review (RR)**: We add instructions in the prompt to ask the model to review its reasoning process sentence by sentence to find and fix reasoning errors (F4 and F5 failures), and then regenerate the solution.
- **Solution Self-Review (SR)**: We add instructions in the prompt to ask the model to review its solution (e.g., a code review) to fix F6–F8 failures before submitting the answer.
- **Error Feedback (EF)**: We feed the compiler error message, runtime exception message, and the number of failed tests back to the model and ask it to refine its answer [12], targeting F8 failures. Similar to resampling, the error feedback loop runs for multiple iterations (up to 20 by default).

Starting from the baseline, we progressively apply the above methods and examine whether each one improves the pass rate. If a method improves the pass rate for any task type, we preserve it when adding the next method. If it does not affect (or even hurts) the pass rate, we discard it.

Figure 10 shows the overall pass rate improvement of different combinations of methods. We focus on Gemini-3.1-Pro and DeepSeek-V4-Flash, as they represent the best-performing and the most cost-efficient models (see §4.4). The takeaways also generalize to other models. Overall, resampling, knowledge self-review, and error feedback yield the most significant pass rate improvements; RAG and reasoning/solution self-review have no significant impact or even hurt the pass rate. Figure 11 shows the percentage of failures of each type when applying these methods, and Figure 12 breaks down the changes in each failure type before and after adding a method. We analyze each of them as follows.

Resampling (RS). Resampling improves the pass rate over the baseline ($k = 1$) by up to 21%. As resampling mitigates model output instability, it helps reduce all types of failures (see Figure 11). More unstable outputs result in larger gains; resampling is more effective for more difficult tasks (DE, PO, and NF) where models are more unstable (see Figure 5). Similarly, it is also more effective for models whose output is more unstable. Intuitively, resampling allows the model to try different solutions, but it does not fundamentally improve a model's capability; 75%–79% of failures remain unresolved.

As shown in Figure 13, the improvement of resampling diminishes as we increase the number of trials. For Gemini-3.1-Pro, pass rate is saturated at 1–5 trials for BU, BI, and PM tasks and 5–8 trials for DE, PO, and NF tasks. In the remainder of this section, we configure the number of trials based on the saturation point for each task type in Figure 13.

> **Takeaway (T12):** Resampling improves the pass rate by up to 21% by mitigating output instability. The improvement is greater for difficult tasks and for models that are more unstable. However, substantial failures remain, as resampling cannot fundamentally improve model capability.

Knowledge Self-Review (KR). Prompting the model to verify all its recalled knowledge improves the overall pass rate by 13.5% for DeepSeek-V4-Flash and 6.9% for Gemini-3.1-Pro. Knowledge recall failures (F2 and F3) are reduced by 50%–53%, indicating that the model can self-correct some of the irrelevant or misleading knowledge during reasoning. However, the effectiveness of such self-review is largely affected by model-specific biases. For example, when prompted to self-review, Gemini-3.1-Pro over-critiques its recalled knowledge, falsely classifying correct knowledge as incorrect leading to reasoning failures (F4), as shown in Figure 12.

> **Takeaway (T13):** Prompting the model to self-review its recalled knowledge reduces the number of knowledge recall failures by 50%–53%. However, the effectiveness of self-review depends heavily on the model's reasoning capability and biases; the model may sometimes mistakenly classify correct knowledge as wrong.

As a side effect, knowledge self-review reduces other types of failures significantly. The review instructions in the prompt implicitly encourage the model to review its entire reasoning process, double-check all recalled knowledge is relevant to the task specification, and perform more in-depth reasoning. This eliminates up to 45% of instruction following failures, up to 32% of reasoning failures, and up to 11% of coding failures, although more suboptimal implementation failures (F7) become visible after fixing prior failures.

> **Takeaway (T14):** As a side effect, the knowledge self-review instructions encourage the model to double-check the task specification, review the entire reasoning process and the solution, and perform more in-depth reasoning. This significantly reduces the number of instruction following, reasoning, and coding failures as well.

RAG. RAG does not improve and even hurts the pass rate (−0.5% for DeepSeek-V4-Flash and −1.0% for Gemini-3.1-Pro). This is because both models already have sufficient *fs* knowledge and general coding capability (F3 failures are few). Moreover, RAG suffers from intrinsic retrieval inaccuracies, which introduce more F2 failures (irrelevant/misleading knowledge). Despite asking to verify retrieved knowledge, models are overconfident about the correctness of RAG and directly apply textbook solutions without adapting them properly, leading to more missing edge cases (F5 failures) and suboptimal implementations (F7 failures), as shown in Figure 12. Our observations align with recent studies, which suggest RAG may not always be the best way for retrieving external knowledge for coding tasks [15, 30, 65].

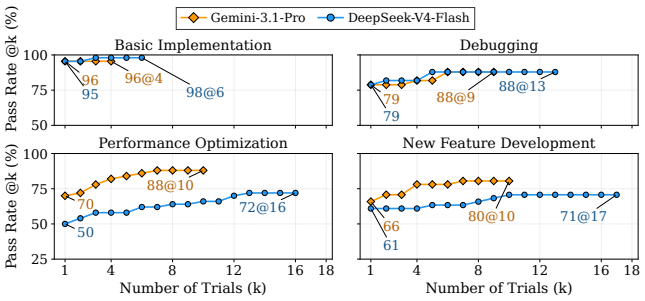


Figure 14. Pass rate at k error feedback iterations. We only evaluate up to the saturation point ("{pass rate}@{k iterations}") at which 4 consecutive iterations do not further increase the pass rate by more than 1%.

> **Takeaway (T15):** For frontier models, RAG does not offer significant improvements (and sometimes causes more failures due to retrieval inaccuracies); these models already have sufficient knowledge for most *fs* development tasks.

Reasoning Self-Review (RR) & Solution Self-Review (SR). Neither reasoning self-review (RS+KR+RR) nor solution self-review (RS+KR+SR) improves the pass rate over knowledge self-review (RS+KR). As a side effect of KR, the model already fixes many reasoning and coding failures (**T14**). As the model reviews its own reasoning and solution, its chain of thought becomes longer, and the model becomes more likely to make mistakes.

> **Takeaway (T16):** Since knowledge self-review already verifies the generated solution and intermediate reasoning steps, performing more self-review has diminishing or negative returns. The model will likely make more mistakes as its chain of thought lengthens during further review.

Error Feedback (EF) improves the overall pass rate by 8.8%–12.3%, with the largest gain on DE, PO, and NF tasks (+9% to 26%). Iterative refinement allows the model to try different solutions (similar to resampling), and feedback helps the model filter out wrong paths. This helps resolve diverse types of failures, as shown in Figure 12. However, the effectiveness of error feedback still relies on the model's ability to reason about feedback and propose solutions. If a model cannot correctly interpret the feedback to propose a correct solution, more iterations yield diminishing returns (see Figure 14). Hence, reasoning failures (F4 and F5) and suboptimal implementations (F7) dominate remaining failures.

Takeaway (T17): Providing the model with error feedback for iterative refinement improves the overall pass rate by up to 12.3%, as the feedback helps the model locate failures and try different solutions. However, effectiveness is limited by the model's intrinsic reasoning capability, leaving up to 58% of failures unresolved.

5 Discussion

Implications for AI-driven *fs* development. Our study provides takeaways for both *fs* developers who use LLMs and ML researchers who develop future models.

For *fs* developers, our study in §4.5 suggests a potential agentic workflow for *fs* development, with separate agents for gathering and verifying *fs* knowledge, deriving the design, writing the code, and conducting reviews. The developers still need to provide comprehensive unit tests for error feedback and iterative refinement, and human supervision may be necessary when performance is critical or exhaustive edge-case coverage is required. We wish to explore building agents for *fs* design and implementation as future work.

For the ML community, the remaining gaps are systematic edge-case reasoning and performance-aware code generation. Future models should therefore improve the ability to reason exhaustively about correctness constraints and translate designs into efficient implementations.

Using and extending *φ-Bench*. *φ-Bench* provides representative *fs* development tasks. It can serve as a reference for evaluating new models and agents. As each task in *φ-Bench* includes a description, reference answer, and test harness, *φ-Bench* can also serve as a dataset for post-training (supervised fine-tuning or reinforcement learning) LLMs. To improve its coverage and quality, we will open source *φ-Bench* and publish our website to collect more tasks and reviews.

Generalizability of *φ-Bench*'s construction pipeline. Our benchmark construction pipeline can be adapted to other domains. Given the knowledge corpus, task templates, and review forms prepared by the domain experts, they can use a similar workflow to leverage AI agents to generate domain-specific tasks. We hope our methodology inspires the development of more domain-specific benchmarks with reduced human effort. We will open source our pipeline, including prompts, orchestration scripts, and website source code.

6 Related Work

LLM benchmarking. Existing domain-specific LLM benchmarks fall into two categories. The first evaluates knowledge-based Q&A [16, 29, 34, 48, 56] but does not assess end-to-end engineering tasks. The second evaluates end-to-end tasks [11, 18, 35, 37, 52, 59]. None of them systematically cover *fs* development tasks. Moreover, they focused on providing a representative set of tasks, without deliberately designing and organizing tasks for analyzing model capabilities and limitations. *φ-Bench* is the first LLM benchmark for *fs* development, enabling us to conduct a comprehensive study to break down the capabilities and limitations of LLMs.

LLM for OS development. Recent works have shown great promise for using LLMs to automate OS development [26, 29, 32, 37, 53]. AutoOS [10] utilizes LLMs to tune kernel parameters. SYSSPEC [32] allows developers to write formal specifications to guide LLM to generate a file system, but it still requires intensive manual prompting. This highlights the critical need to systematically analyze LLMs' ability to autonomously design and implement low-level software systems. Our study evaluates LLMs on *fs* development, facilitating future research on AI-driven OS development.

File system design and implementation. File system development has long been notoriously labor-intensive, often requiring months to years of engineering effort [6, 7, 27, 32]. File systems have been continuously evolving to exploit new hardware characteristics, such as the shift from FFS for HDDs to F2FS for flash-based SSDs [6, 27]. Recent advancements in LLMs pave the way for AI-driven *fs* development, yet it is still unclear how LLMs should be applied in this domain. *φ-Bench* introduces the first benchmark and conducts an empirical study to analyze LLM capabilities on *fs* development.

7 Conclusion

We present *φ-Bench*, an LLM benchmarking framework for file system design and implementation tasks. With *φ-Bench*, we evaluate various representative LLMs. We wish our study results would shed light on future research and development of using LLMs for file system development.

References

[1] Rishabh Agarwal, Cem Anil, Jean-Baptiste Alayrac, et al. 2024. Many-Shot In-Context Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. Spotlight Presentation.

[2] Zhipu AI. 2025. GLM-5: An Open Bilingual Pre-Trained Model. *arXiv preprint* (2025).

[3] Anthropic. 2025. Anthropic Claude-Code. https://code.claude.com/docs/en/overview

[4] Anthropic. 2025. The Claude 4.6 Model Family. *Anthropic Technical Report* (2025).

[5] Inc. Anysphere. 2024. Cursor: The AI Code Editor. https://www.cursor.com.

[6] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[7] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 69–86.

[8] Bradley Brown, Jordan Juravsky, Ryan Saul Ehrlich, Ronald Clark, Quoc V Le, Christopher Re, and Azalia Mirhoseini. 2025. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. https://openreview.net/forum?id=0xUEBQV54B

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 1877–1901.

[10] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, Qi Guo, and Yunji Chen. 2024. AutoOS: make your OS more powerful by exploiting large language models. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 292, 15 pages.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *International Conference on Learning Representations (ICLR)*.

[13] Google DeepMind. 2025. Gemini 3 Pro: Next-Generation Frontier Models. *Google Technical Report* (2025).

[14] DeepSeek-AI. 2026. DeepSeek-V4 Technical Report. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf

[15] Wenchao Gu, Juntao Chen, Yanlin Wang, Tianyue Jiang, Xingzhe Li, Mingwei Liu, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. *arXiv preprint arXiv:2503.20589* (2025). https://arxiv.org/abs/2503.20589

[16] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving with the MATH Dataset. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. 2179–2194.

[17] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. 2020. How Can We Know What Language Models Know? arXiv:1911.12543 [cs.CL] https://arxiv.org/abs/1911.12543

[18] 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).

[19] Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, et al. 2022. Language Models (Mostly) Know What They Know. *arXiv preprint arXiv:2207.05221* (2022).

[20] Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. 2022. Language Models (Mostly) Know What They Know. *arXiv preprint arXiv:2207.05221* (2022).

[21] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020).

[22] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781.

[23] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 39–48.

[24] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, Huntsville, ON, Canada.

[25] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. arXiv:2205.11916 [cs.CL] https://arxiv.org/abs/2205.11916

[26] Nancy Lau, Louis Sloot, Jyoutir Raj, Giuseppe Marco Boscardin, Evan Harris, Dylan Bowman, Mario Brajkovski, Jaideep Chawla, and Dan Zhao. 2026. ZeroDayBench: Evaluating LLM Agents on Unseen Zero-Day Vulnerabilities for Cyberdefense. arXiv:2603.02297 [cs.CR] https://arxiv.org/abs/2603.02297

[27] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.

[28] 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).

[29] Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. 2025. OSVBench: Benchmarking LLMs on Specification Generation Tasks for Operating System Verification. arXiv:2504.20964 [cs.CL] https://arxiv.org/abs/2504.20964

[30] Yanzhou Li, Shangqing Liu, Kangjie Chen, Tianwei Zhang, and Yang Liu. 2025. Impact-driven Context Filtering for Cross-file Code Completion. *arXiv preprint arXiv:2508.05970* (2025). https://arxiv.org/abs/2508.05970

[31] Linus Torvalds. 2026. Linux Kernel Source Code. https://github.com/torvalds/linux Linux kernel.

[32] Qingyuan Liu, Zou Mo, Hengbin Zhang, Dong Du, Yubin Xia, and Haibo Chen. 2025. Sharpen the Spec, Cut the Code: A Case for Generative File System with SYSSPEC. *arXiv preprint arXiv:2512.13047* (2025).

[33] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. 31–44.

[34] Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. 2024. MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts. In *International Conference on Learning Representations (ICLR)*.

[35] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 722–727. doi:10.1109/ASP-DAC58780.2024.10473904

[36] Xinyu Ma, Chen Zhang, Tian Xie, Jia Chen, Jiaqing Shen, Ruiguang Yang, and Zhihua Lou. 2021. Learning Dense Representations for Sparse Retrieval. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1605–1614.

[37] Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2024. Kgym: A platform and dataset to benchmark large language models on linux kernel crash resolution. *Advances in Neural Information Processing Systems* 37 (2024), 78053–78078.

[38] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. 1984. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984), 181–197.

[39] MINIMAX. 2026. MiniMax M2.7. https://www.minimax.io/news/minimax-m27-en

[40] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 33–50.

[41] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA.

[42] Douglas C Montgomery. 2017. *Design and Analysis of Experiments* (9th ed.). John Wiley, Sons.

[43] OpenAI. 2025. GPT-5.2 Technical Report. *OpenAI Technical Report* (2025).

[44] OpenAI. 2025. OpenAI Codex. https://chatgpt.com/codex

[45] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 27730–27744.

[46] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 2463–2473. doi:10.18653/v1/D19-1250

[47] Long Phan, Alice Gatti, Nathaniel Li, and et al. 2026. A benchmark of expert-level academic questions to assess AI capabilities. *Nature* 649, 8099 (Jan. 2026), 1139–1146. doi:10.1038/s41586-025-09962-4

[48] Shvetank Prakash, Andrew Cheng, et al. 2025. QuArch: A Question-Answering Dataset for AI Agents in Computer Architecture. *IEEE Computer Architecture Letters* 24, 1 (2025).

[49] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.

[50] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage* 9, 3 (2013), 1–32.

[51] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.

[52] 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).

[53] Kareem Shehada, Yifan Wu, Wyatt D. Feng, Adithya Iyer, Gryphon Kumfert, Yangruibo Ding, and Zhiyun Qian. 2025. Rethinking Kernel Program Repair: Benchmarking and Enhancing LLMs with RGym. arXiv:2511.15757 [cs.SE] https://arxiv.org/abs/2511.15757

[54] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 1–16.

[55] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th ed.). Wiley.

[56] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, et al. 2023. Large language models encode clinical knowledge. *Nature* 620, 7972 (2023), 172–180.

[57] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *Proceedings of the 1996 USENIX Annual Technical Conference*. 1–14.

[58] Annalisa Szymanski, Noah Ziems, Heather A Eicher-Miller, Toby Jia-Jun Li, Meng Jiang, and Ronald A Metoyer. 2024. Limitations of the LLM-as-a-Judge Approach for Evaluating LLM Outputs in Expert Knowledge Tasks. *arXiv preprint arXiv:2410.20266* (2024).

[59] Shailja Thakur, Baleegh Ahmad, Zhenxm Fan, Pearce Hammond, et al. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[60] Together AI. 2024. Together AI API. https://api.together.xyz

[61] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *International Conference on Learning Representations (ICLR)*.

[62] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2022. Finetuned Language Models Are Zero-Shot Learners. In *International Conference on Learning Representations (ICLR)*.

[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Le, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[64] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320.

[65] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 53270–53290. https://proceedings.mlr.press/v235/wu24a.html