Perish or Flourish? A Holistic Evaluation of Large Language Models for Code Generation in Functional Programming

Anonymous ACL submission

Abstract

Functional programming provides strong foundations for developing reliable and secure software systems, yet its adoption remains not widespread due to the steep learning curve. Recent advances in Large Language Models (LLMs) for code generation present new opportunities to lower these barriers. However, extensive evaluations of LLMs primarily focus on imperative programming languages, and their capabilities in functional programming languages remain under-explored. To address this gap, we introduce FPEval, a holistic evaluation framework equipped with FPBench, a new benchmark of 721 programming tasks across three difficulty levels on three mainstream FP languages: Haskell, Ocaml and Scala. FPEval provides comprehensive evaluation infrastructures with both test validations with comprehensive test suites and static analysis tools to assess both functional correctness and code style and maintainability. Using this framework, we evaluate seven state-of-the-art LLMs, including GPT-5 and Claude Opus 4.6 for code generation in functional programming languages and Java as an imperative baseline. Our results demonstrate that LLM performance in functional programming improves substantially with model advancement; however, error rates remain significantly higher in purely functional languages (Haskell and OCaml) than in hybrid (Scala) or imperative (Java) languages. More critically, LLMs frequently generate non-idiomatic functional code that follows imperative patterns, raising concerns about code style and maintainability. Finally, we showed that LLMs can partially self-repair both correctness and quality issues when provided with static analysis feedback and hand-crafted instructions for common types of issues, paving the way for future improvements

1 Introduction

Functional programming is an emerging programming paradigm that conceptualizes computation as the evaluation of mathematical functions rather than a sequence of state-mutating commands in widely-used imperative programming. This model of computation offers unique advantages over imperative programming, resulting in software systems that are modular, deterministic, and susceptible to formal reasoning. For example, functional programming offers immutability, which ensures that a variable's value cannot be changed during its existence. Immutability enables functional programming to mitigate common sources of software bugs in imperative programming such as side effects and race conditions. Consequently, functional programming is increasingly recognized as a future paradigm for reliable and secure software development (Achten, 2013; Hughes, 1989; Ray et al., 2014). Despite these benefits, functional programming poses a steep learning curve. The paradigm requires a fundamental shift from imperative commands with mutable state to high-level abstractions of program behaviors such as recursion, higher-order functions, and monads, which many developers find difficult. Consequently, functional programming remains unduly less popular in practice than imperative programming.

The recent rise of coding assistants powered by Large Language Models (LLMs), such as GitHub Copilot (Chen et al., 2021), Cursor (Anysphere, 2023), and Claude Code (Anthropic, 2025a), presents new opportunities for functional programming. By supporting software developers in tasks such as code generation (Li et al., 2022; Chen et al., 2021), bug fixing (Xia et al., 2023; Le-Cong et al., 2025b), and technical question answering (Xu et al., 2023), these tools offer particular benefits to novice programmers and have the potential to reduce the learning curve associated with programming in general (Prather et al., 2023; Kazemitabaar et al., 2023). If similar benefits extend to functional programming, these LLM-based coding assistants could contribute to increasing the adoption of the pro-

gramming paradigm in practice. Unfortunately, existing studies on LLM-based coding (Fan et al., 2023) mainly focus on imperative programming languages, such as Python and Java. In contrast, the applicability of LLMs to functional programming languages, such as Haskell, OCaml, and Scala, remains significantly under-explored.

In this work, we conduct the first comprehensive empirical study to evaluate the capabilities of LLMs for code generation in functional programming. Our objective is three-fold: (1) to assess the ability of LLMs for generating correct and high-quality code in functional programming; (2) to evaluate the quality of LLM-generated code beyond functional correctness, with particular attention to coding style and maintainability; and (3) to evaluate the effectiveness of LLMs on self-repairing their mistakes. To this end, we structure our study using the following research questions:

- **RQ1:** How effective are LLMs for code generation in functional programming languages?
- **RQ2:** What are common coding style and maintainability issues in code generated by LLMs in functional programming languages?
- **RQ3:** How effective are self-repair mechanisms in improving the correctness and code style and maintainability of LLM-generated functional code?

To answer these questions, we introduce FPEval, a holistic evaluation framework designed for code generation in functional programming. FPEval is powered by FPBench, a new multi-language benchmark consisting of 721 programming tasks distributed across three difficulty levels (184 easy, 346 medium, and 191 hard) and three languages: Scala, Haskell, and OCaml. To ensure a rigorous assessment on LLM-generated code, we construct a comprehensive test suite for each task, augmenting public test cases from LeetCode with private test cases, which target specific boundary conditions and edge cases for each programming task. Furthermore, FPEval also integrates well-known static analysis tools, including HLint (Mitchell, 2024), OCamlFormat (OCamlFormat Developers, 2024), and Scalastyle (Farwell and Scalastyle Contributors, 2024), enabling a dual assessment of both the functional correctness and code style and maintainability of the generated code.

Using FPEval, we evaluate seven state-of-the-art LLMs including GPT-5 and Claude Opus 4.6 on functional programming tasks, with Java included as an imperative baseline. Our evaluations show that LLM performance on functional programming improves substantially with model advancement, with Claude Opus 4.6 and GPT-5 achieving a substantially higher number of functionally-correct code over older and smaller models such as GPT 3.5. However, a consistent performance gap persists between purely functional languages (Haskell and OCaml) and hybrid or imperative languages (Scala and Java). Beyond correctness, we also found that LLM-generated code frequently exhibits poor style and maintainability, favoring imperative patterns over best practices of functional programming. Such low-quality code becomes more prevalent as correctness improves, suggesting a "reward hacking" phenomenon in which models optimize for correctness at the expense of code quality. These findings highlight the need for future LLM training and evaluation to explicitly incorporate functional programming best practices. Finally, we show that LLMs can partially self-repair both correctness and quality issues when provided with static analysis feedback and hand-crafted instructions targeting common error types, paving the way for future improvements in LLM-based code generation for functional programming.

In summary, our main contributions include:

- We introduce FPEval, a holistic framework for evaluating LLM code generation in the functional programming paradigm. FPEval is powered by FPBench, a new curated dataset of 721 programming tasks with executable test suites in Scala, Haskell, and OCaml, and an automated evaluation pipeline that leverages software testing and static analysis tools to assess both the correctness and the code style and maintainability of generated code.
- We present a comprehensive evaluation of LLMs for functional programming code generation, systematically assessing both functional correctness and code style and maintainability of generated code by LLMs.
- We provide a detailed analysis of code style and maintainability issues in LLM-generated functional code and demonstrate the potential LLM's self-repair for improving the code style and maintainability issues with static analysis feedbacks and detailed instructions for common issues.

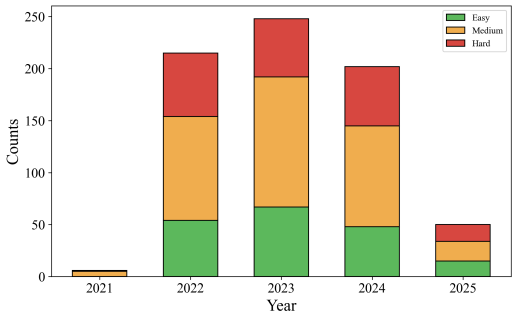

Figure 1: Task distribution across time

- We advance research in code generation for functional programming by releasing FPEval under Apache 2.0 License, lowering barriers for academic research and establishing foundational benchmarks and metrics for future work. An early access version of FPEval can be found at `https://anonymous.4open.science/r/FPEval-5621`.

2 Benchmark Construction and Evaluation Pipeline

2.1 Benchmark Construction

Data Collection. To support a rigorous empirical study in the code generation of LLMs in functional programming, we curated a comprehensive benchmark dataset from LeetCode (LeetCode, Inc., 2025), a widely-used platform for competitive programming and technical interviews. From the full collection, we filtered out premium-only content to select 721 publicly available algorithmic tasks. These tasks span three distinct difficulty levels consisting of 184 easy, 346 medium, and 191 hard programming tasks. For each task, we extracted the problem description, I/O constraints, and public test cases via LeetCode's public API and custom HTML parsers. Collected public test cases were incorporated into prompt construction and later reused for initial correctness checks. Figures 1 illustrate the diversity of the dataset, i.e., FPBench. Our data collection period from 2021 to 2025 enables the capture of evolving trends in algorithm design and problem types. FPBench also include programming tasks with diverse difficulty levels, enabling us to systematically evaluate model performance under diverse problem complexity.

Language Template Generation. A critical challenge to evaluating functional programming on LeetCode tasks is the lack of support for widely-used functional programming languages such as Haskell and OCaml. Specifically, LeetCode do not provide any templates for these functional programming, yielding difficulties for LLMs to understand I/O and typing requirements. To bridge this gap, we implemented a transpiler that constructs syntactically and semantically valid starter templates for Haskell and OCaml from Python templates provided by LeetCode. This translation was challenging due to fundamental differences in programming paradigms and type system, e.g., dynamic versus static typing, imperative versus functional control flows, and the lack of one-to-one mappings for many language constructs. Our transpiler addresses these discrepancies by (1) performing structural translation of Python function definitions to typed signatures in Haskell and OCaml, including parameter binding and return types; (2) inferring types for nested and compound data structures (e.g., mapping `List[List[int]]` to `[[Int]]`); (3) normalizing I/O semantics to conform to functional patterns while preserving the semantics of the original task specification. To ensure the correctness of this transpilation, we carefully designed our transpiler, incorporated automated validation through syntax and type checking, and manually reviewed and rectified any remaining errors. Full verification details are provided in Appendix J.

2.2 Evaluation Pipeline

Test validation. The FPEval pipeline begins by assessing the functional correctness of the generated code. While prior works (Liu et al., 2024) typically utilize public test cases from LeetCode, this test suite is often limited in coverage and may fail to detect subtle logical errors. To mitigate this, we augment each task with a suite of *private test cases*. Following Jain et al. (Jain et al., 2025), we employ GPT-4o to synthesize these rigorous test cases directly from task descriptions to target boundary conditions, corner cases, and degenerate inputs (see Appendix A for the specific prompts). Note that, executing these tests within the functional paradigm also presents a critical challenge. Unlike Python, which supports flexible and dynamic execution, statically typed languages like Haskell and OCaml require rigorous type scaffolding, strict compilation, and runtime isolation. To address this, we developed a specialized test infrastructure for functional programming languages using test templates with the standard library and a Docker-based isolated runtime environment, as detailed in Appendix B.

Code Style and Maintainability Evaluation. Beyond functional correctness, the code style and maintainability of the generated code is a critical dimension of our evaluation. A solution may satisfy all test cases while still being low quality with respects to best practices in functional programming. Such code accumulates technical debt, rendering systems difficult to maintain (Kruchten et al., 2012) and even potentially introducing latent safety issues (Perry et al., 2023). To systematically quantify this dimension, FPEval integrates widely-used static analysis tools that check whether a code snippet strictly adhere functional programming best practices. Specifically, we utilize `HLint` and `GHC` for Haskell, `Dune` and `OCamlFormat` for Ocaml, and `Scalastyle` for Scala. An example categorization of the rules applied by these tools is provided in Appendix D.

2.3 LLM Selection and Configurations

To ensure a comprehensive evaluation across varying levels of capability and cost, we evaluate three Large Language Models of OpenAI GPT series including GPT-3.5-turbo (OpenAI, 2022), GPT-4o (OpenAI, 2023), GPT-5 (OpenAI, 2025), DeepSeek-V3.2 (DeepSeek-AI et al., 2025), Devstral Small 1.1 (Rastogi et al., 2025), Qwen3-Coder-Next (Cao et al., 2026), Claude Opus 4.6 (Anthropic, 2025b). We adopt *zero-shot prompting*, providing each model only with the natural language task description and a language-specific starter template (see Listing G.1). All models use `max_tokens = 2048` and `temperature = 0.7`, except GPT-5, which fixes `temperature = 1.0` by design; we also disable its reasoning for a fair comparison. For each task, we generate a single candidate solution and extract code blocks by discarding any conversational text.

3 Experiments

3.1 RQ1: Effectiveness?

To evaluate the effectiveness of LLMs for code generation in functional programming, we execute the generated code against the comprehensive test suite for each task. Following (Jain et al., 2025; Liu et al., 2024), we evaluate their effectiveness via `pass@1`, i.e., the proportion of tasks where the model's first generated solution satisfies all test cases, and classify unsuccessful generations into: (1) *Compilation Errors* (syntax/type mismatches); (2) *Test Failures* (logic errors); and (3) *Timeouts* (execution limits). We adopt pass@1 as the primary metric because it reflects realistic single-query deployment scenarios; a preliminary pass@10 analysis (Appendix I) confirms that broader sampling raises the correctness ceiling but amplifies the quality trade-offs documented in RQ2.

Main Results. Table 1 presents LLM performance across Haskell, OCaml, Scala, and Java. Overall, code generation in functional programming improves substantially with model advancement, with GPT-5 achieving roughly 3-5x the pass rate of GPT-3.5. However, a consistent performance gap persists: most models perform notably worse in purely functional languages (Haskell, OCaml) compared to hybrid/imperative ones (Scala, Java). Specifically, smaller or older models exhibit limited functional capabilities, with Devstral Small 1.1 and Qwen3-Coder-Next achieving pass rates of only 10.8% and 12.8% in Haskell, respectively. While newer models demonstrate strong upward trend, e.g., DeepSeek-V3.2 reaching 55.5% pass rates in Java, their performance gap to their own imperative counterparts remains wide. For instance, GPT-5's pass rates in Haskell (42.3%) and OCaml (52.2%) still lag behind its Java performance (61.1%) by approximately 19 and 9 percentage points. A striking exception is Claude Opus 4.6, which breaks this pattern by achieving a peak pass rate of 73.4% in OCaml outperforming its own Java capability by 17.8 percentage points. However, as we will discuss in Section 3.2, this comes at a significant cost to code quality and maintainability. These results suggest that despite continuous LLM scaling, the performance gap between functional and imperative code generation remains significant unless specialized optimizations are applied.

Compilation Errors. Unlike imperative code generation, LLMs frequently generate uncompilable functional code with syntax or type errors. For instance, GPT-3.5 exhibits high compilation error rates of 43.1% and 25.5% in OCaml and Haskell. This issue is magnified in models like Qwen3-Coder-Next and Devstral Small 1.1, where compilation error rates exceed 50% in both languages (reaching nearly 80% for Qwen3 in Haskell). While newer models like GPT-4o, DeepSeek-V3.2, and GPT-5 reduce these errors with GPT-5 lowering OCaml compilation error rate to 13.7% and Java to a low 4.6%, they still struggle with Haskell's rigid type system, where GPT-5's error rate remains high at 24.3%. Similarly, Claude Opus 4.6 faces a steep 42.7% compilation error

rate in Haskell though it achieves a remarkably low 1.5% in OCaml.

Model	Metric	Haskell	OCaml	Scala	Java
GPT-3.5	Pass	14.2%	9.4%	19.3%	22.2%
	Fail	50.2%	43.1%	65.5%	57.3%
	CE	25.5%	43.1%	14.0%	10.4%
	Timeout	10.1%	4.3%	1.3%	10.1%
GPT-4o	Pass	27.2%	36.2%	38.8%	43.7%
	Fail	45.1%	39.1%	51.6%	43.0%
	CE	21.4%	21.1%	9.6%	6.2%
	Timeout	6.4%	3.6%	0.0%	7.1%
GPT-5	Pass	42.3%	52.2%	58.4%	61.1%
	Fail	32.6%	33.4%	32.4%	27.5%
	CE	24.3%	13.7%	9.3%	4.6%
	Timeout	0.8%	0.7%	0.0%	6.8%
DeepSeek-V3.2	Pass	34.5%	46.3%	44.4%	55.5%
	Fail	30.3%	34.7%	42.7%	38.7%
	CE	33.2%	17.3%	13.0%	10.8%
	Timeout	2.1%	1.7%	0.0%	0.0%
Devstral Small 1.1	Pass	10.8%	9.5%	25.0%	35.0%
	Fail	25.5%	29.4%	51.7%	44.7%
	CE	59.3%	55.7%	22.7%	18.7%
	Timeout	4.5%	5.4%	0.6%	1.7%
Qwen3-Coder-Next	Pass	12.8%	31.9%	33.3%	35.9%
	Fail	7.0%	17.0%	28.0%	27.2%
	CE	80.0%	50.8%	38.2%	36.9%
	Timeout	0.3%	0.3%	0.4%	0.0%
Claude Opus 4.6	Pass	40.7%	73.4%	41.6%	55.6%
	Fail	16.4%	25.1%	34.6%	28.7%
	CE	42.7%	1.5%	23.9%	15.7%
	Timeout	0.3%	0.0%	0.0%	0.0%

Table 1: Effectiveness of studied LLMs for code generation in Haskell, OCaml, Scala, and Java. CE = Compile Error (syntax/type error).

3.2 RQ2: Code style and maintainability

To evaluate the code style and maintainability of LLM-generated code, we focus on the subset of generated code that is functionally correct, i.e., those that pass all test cases. Within this subset, we utilized a suite of language-specific static analysis tools to evaluate the adherence of LLM-generated code to best practices for code style and maintainability in functional programming. Specifically, we used HLint and GHC warnings for Haskell, dune and ocamlformat for Ocaml, Checkstyle and PMD for Java, and Scalastyle for Scala. We define a code snippet as "clean" only if it satisfies all applicable checks without any violations.

Main Results. Table 2 illustrates the proportion of "clean" code samples among the functionally-correct solutions generated by the studied LLMs. Overall, high code quality does not always correlate with model scaling. In several specific cases, open-source models outperform larger proprietary counterparts on code style: for instance, Devstral Small 1.1 achieves a higher clean code ratio in OCaml (50%) than DeepSeek-V3.2 (36%), and both Devstral Small 1.1 (63%) and Qwen3-Coder-Next (64%) maintain competitive clean code ratios in Scala relative to GPT-5 (57%). However, this advantage is not consistent across all language–model pairings, and proprietary models such as GPT-4o retain strong style scores in several settings.

Our broader analysis reveals an unexpected trade-off between functional correctness and code quality across almost all studied LLMs. When ranking models by their functional correctness, we observe a consistent inverse correlation with their clean code ratios. We interpret this as a form of reward hacking: during training, models are rewarded for passing test cases, creating an implicit incentive to produce code that satisfies the test oracle by any means, including falling back on imperative shortcuts, rather than generating idiomatic functional code. As a result, as models grow more capable of passing tests, their adherence to functional programming idioms and maintainability practices systematically degrades. For instance, in OCaml, as the pass rate increases from Devstral Small 1.1 (9.5%) to DeepSeek-V3.2 (46.3%), the proportion of clean code drops from 50% to 36%. A similar decline is observed in Scala, where smaller models maintain stronger clean code ratios while GPT-5, despite its higher pass rate, falls to 57%. This pattern suggests that test-passing and idiomatic functional code are, to some degree, in tension under current training objectives.

Most notably, Claude Opus 4.6 exemplifies this extreme trade-off: it records the lowest clean code ratio in OCaml at just 33%, sharply contradicting and undermining its exceptionally high pass rate. Furthermore, producing clean code in Haskell remains universally challenging; most models, including GPT-4o (46%) and GPT-5 (45%), fail to exceed 50%. Ultimately, this persistent trend implies that current LLMs prioritize passing functional test validations over code style. In practice, this can potentially harm software systems by introducing unnoticed technical debt, which increases the long-term cost of maintainability.

Common Issues. To shed the light to future improvements, we conducted a semi-automated analysis on the low quality code with respects to best practices in functional programming detected by static analysis tools. We first manually identified

Model	Language			
	Haskell	OCaml	Scala	Java
GPT-3.5	22%	63%	61%	58%
GPT-4o	46%	61%	77%	63%
GPT-5	45%	44%	57%	51%
DeepSeek-V3.2	33%	36%	65%	47%
Devstral Small 1.1	36%	50%	63%	64%
Qwen3-Coder-Next	51%	43%	64%	62%
Claude Opus 4.6	57%	33%	43%	43%

Table 2: Proportion of clean code in functionally correct solutions generated by studied LLMs

recurring code style and maintainability issues and mapped them to specific diagnostics from our static analysis tools; subsequently, we developed a script to automatically parse the tool logs across the entire dataset to quantify the prevalence of these issues.

In Haskell, the most common issues were non-idiomatic patterns such as unnecessary lambdas (37x), missed eta reductions (36x), and redundant brackets. GHC further warned about unused variables and non-exhaustive pattern matches. In OCaml, most issues were code readability, raised by ocamlformat, e.g. redundant semicolons and parentheses. We also observed several critical issues, including type errors and unintended mutation of immutable data structures, although these occurred less frequently. Overall, while these issues do not compromise the functional correctness of LLM-generated code, they violate established best practices in functional programming and may increase the risk of future maintenance issues.

The most interesting issues are observed in Scala. Specifically, we found that the most frequent issue was the use of `return` (236x), which is discouraged in idiomatic Scala, as it indicate the use of imperative paradigm. Other violations included magic numbers, long methods, and inconsistent procedure definitions. The consistent presence of imperative constructs suggests that LLMs may not fully leverage Scala's functional features, instead generating code that resembles Java-style imperative logic, which is allowed but not encouraged in Scala. These observations lead us to a broader question: *Do LLMs truly generate functional code, or they just "hack" the functional correctness by embedding imperative code?*

Imperative Bias in LLM-Generated Functional Code. Despite being explicitly prompted for functional programming, LLMs frequently produce code exhibiting an *imperative bias*, using constructs such as `return`, mutable variables, and explicit loops in Scala, and mutable references or IO effects in Haskell and OCaml, all of which betray the core FP principles of purity, immutability, and declarative control flow. Examples of these patterns are provided in Appendix E and F.

To quantify this, we detected imperative constructs via keyword-based pattern matching (e.g., `var`, mutable collections, explicit loops in Scala; mutable references and assignments in OCaml; excessive `if-then-else` in Haskell) across all generated code. As shown in Table 3, the bias is pervasive across all models and languages. Notably, the most capable frontier models consistently show the highest imperative bias rates: GPT-5 embeds imperative patterns in 88% of Haskell, 80% of OCaml, and 94% of Scala solutions, while Claude Opus 4.6 follows closely at 80%, 82%, and 79% respectively. These results suggest a widespread *reward hacking* pattern: as models scale and achieve higher pass rates, they increasingly exploit imperative shortcuts rather than generating idiomatic functional code.

Violations of Core Linguistic Properties. To further characterize the imperative bias, we quantify violations of three core functional programming principles across the studied models: (1) declarative control flow, i.e., reliance on explicit loops or `if-then-else` chains rather than pattern matching or higher-order functions; (2) purity, i.e., presence of IO effects or hidden side effects in nominally pure functions; and (3) immutability, i.e., use of mutable references or in-place assignments. Declarative control flow violations are the most pervasive, exceeding 68% in Haskell for the strongest models, confirming a systematic reversion to sequential constructs even in languages that strongly discourage them. Immutability violations are language-dependent, rare in Haskell due to compiler enforcement, but frequently exceeding 40% in OCaml and Scala. Purity violations occur at lower but consistent rates (5–25%) across all languages. Together, these results confirm that the imperative bias in Table 3 reflects multi-dimensional failures to adhere to functional programming principles, not merely superficial stylistic choices. Full per-model violation rates are reported in Appendix K.

3.3 RQ3: Self-repair

Prior work (Liu et al., 2024) suggests that LLMs can self-repair their mistakes with a feedback loop, this research question investigates whether LLM's *self-repair ability* can improve the functional correctness and code style and maintainability of LLM

Model	Haskell	OCaml	Scala
GPT-3.5	53%	57%	94%
GPT-4o	63%	42%	88%
GPT-5	88%	80%	94%
DeepSeek-V3.2	83%	64%	90%
Devstral Small 1.1	58%	38%	69%
Qwen3-Coder-Next	65%	57%	70%
Claude Opus 4.6	80%	82%	79%

Table 3: Percentage of LLM-generated code with imperative patterns

outputs in functional programming languages. To this end, we examine the ability of ChatGPT to refine its own code given explicit repair prompts.

Following (Liu et al., 2024), we consider two following repair approaches: (1) `Simple Repair`. We provide the model with a high-level system instruction to correct its previously generated solution (see Appendix G.1), without any structured guidance beyond the task description and a functional programming reminder. This serves as a minimal repair baseline. (2) `Instruction-guided Repair`. In this approach, we adopt a more structured repair strategy by combining feedback from static analysis and test validation with hand-crafted instructions targeting common quality issues. For each issue type, the model is provided with explicit instructions on how to fix them, tailored to each functional programming language and directly mapping common error categories to actionable repair guidance (see Tables 6, 7, 8) in G.

Results. Table 4 details the number of functionally-correct solutions across three settings: initial generation, simple repair, and instruction-guided repair. Overall, we can see that LLMs consistently improve the functional correctness of across all models and languages with self-repair capabilities using both studied repair approaches. A closer examination reveals a nuanced interaction between repair approaches and solution quality. While `Instruction-guided Repair` significantly boosts performance in certain contexts (e.g., DeepSeek-V3.2 in Haskell: 249 $\rightarrow$ 360), it does not always strictly outperform `Simple Repair` in terms of raw pass rates. For instance, in Haskell and OCaml with GPT-5, `Simple Repair` achieves slightly higher correctness (e.g., Haskell: 301 vs. 295). This phenomenon is explained by analyzing code style and maintainability of the generated code. As shown in Table 9 in Appendix H, in most of the cases, `Instruction-guided Repair` yields the highest rate of clean code, i.e., those follows best functional programming practices, substantially outperforming `Simple Repair`. This suggests that LLMs face a trade-off between code style and maintainability and functional correctness. `Simple Repair` allows the models to prioritize passing test cases by any necessary approaches, even retaining or introducing imperative shortcuts to fix errors. Meanwhile, `Instruction-guided Repair`, by enforcing strict FP constraints, increases the difficulty of the generation task. The models attempt to find a solution that is *both* correct and high quality with respects to code style and maintainability. In cases where the model cannot synthesize a high quality solution, it may fail to generate valid code, resulting in a slight drop in pass rates despite the better code style and maintainability of the successful solutions.

Fortunately the performance gap between `Instruction-guided Repair` and `Simple Repair` are minimal. For example, `Simple Repair` with GPT-5 only generates 6 more correct solutions than `Instruction-guided Repair` for Haskell. These results suggest that `Instruction-guided Repair` provides higher quality code with minimal trade-offs in `pass@1` rate. They also indicate that LLMs are increasingly capable of achieving functional correctness while simultaneously adhering to code style and maintainability considerations if best practices in functional programming are explicitly provided to these models. These findings suggest that these best practices should be more explicitly incorporated into LLM training and inference to enable the generation of code that is both functionally correct and of high quality in terms of code style and maintainability.

4 Related Works.

Code Generation Benchmarks. Numerous datasets have been proposed to evaluate the code generation capabilities of Large Language Models (LLMs). Early benchmarks primarily focused on Python code generation, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). More recently, the research community has pivoted toward multilingual evaluation through datasets such as MultiPL-E (Cassano et al., 2023) and HumanEval-XL (Peng et al., 2024), though these still largely prioritize imperative programming paradigms (e.g., C++, Java, and Go). Recent multilingual benchmarks such as McEval (Chai et al.,

Table 4: Number of correct solutions after applying different repair strategies. Ha. = Haskell, Oc. = OCaml, Sc. = Scala. **Bold** = best result per model per language. Claude Opus 4.6 is excluded due to budget constraints.

	GPT-3.5			GPT-4o			GPT-5			DeepSeek-V3.2			Devstral Small 1.1			Qwen3-Coder-Next		
Setting	Ha.	Oc.	Sc.	Ha.	Oc.	Sc.	Ha.	Oc.	Sc.	Ha.	Oc.	Sc.	Ha.	Oc.	Sc.	Ha.	Oc.	Sc.
Code Generated	93	68	118	196	261	280	287	376	420	249	334	320	78	68	180	92	230	240
Simple Repair	**102**	78	117	204	**272**	**297**	**301**	**396**	431	332	**365**	334	86	91	**247**	165	258	267
Instruction Repair	98	**85**	**139**	**209**	266	288	295	391	**433**	**360**	358	**342**	**93**	**99**	245	**184**	**269**	**288**

2024) (covering Scala, Haskell, Scheme, Racket, and F#, among 40 languages) and MDEval (Liu et al., 2025) (which includes F#) extend evaluation coverage across many programming languages. However, they treat functional languages as items in a broad multilingual inventory and assess only functional correctness: McEval via generation, explanation, and completion; MDEval via debugging. Other lines of work have examined specification-oriented languages, such as F* (Lahiri, 2024), Dafny (Loughridge et al., 2024), and JML (LeCong et al., 2025a). Different from these existing efforts, FPEval is purpose-built for the functional paradigm: it complements correctness with paradigm-aware quality metrics (purity, immutability, declarative control flow) and static-analysis–based style and maintainability assessment over Haskell, OCaml, and Scala. This holistic approach surfaces systematic behavioral patterns of modern LLMs that language-agnostic benchmarks structurally cannot capture, including an imperative bias where models generate imperative-style constructs even in functional contexts, and reward hacking behaviors, in which models achieve functional correctness while sacrificing code quality and adherence to functional programming principles.

Automation in Functional Programming. Several works in the literature have also investigated the automation of functional programming. Notably, Gissurarson et al. proposed PropR (Gissurarson et al., 2022), a property-based automated program repair (APR) framework designed to fix Haskell bugs by leveraging property-based testing and type-driven synthesis. Recently, Van Dam et al. (Van Dam et al., 2024) fine-tuned UniXCoder and CodeGPT specifically for Haskell code completion, demonstrating that language-specific models can significantly outperform general-purpose LLMs in this domain. Different from these studies, which are primarily restricted to the Haskell ecosystem, our work expands the scope to a diverse set of functional programming languages, including Haskell, Scala, and OCaml. Furthermore, rather than focusing on fine-tuning smaller architectures, we conduct a comprehensive evaluation of the latest frontier LLMs, including GPT-3.5, GPT-4o, GPT-5, DeepSeek-V3.2, Devstral Small 1.1 and Qwen3-Coder-Next. This allow us to assess the capabilities of these state-of-the-art models across multiple functional languages, providing a broader perspective on the current boundaries of AI-driven software engineering in functional programming.

5 Conclusion

In this paper, we presented FPEval, the first comprehensive evaluation framework and empirical study for assessing the code generation abilities of Large Language Models in the functional programming paradigm. FPEval contains an dual-assessment approach, containing both rigorous test suites and static analysis tools, allowed us to not only evaluate functional correctness but also code style and maintainability of LLM-generated code.

Our findings reveal a significant "paradigm gap" in current LLM performance: these models demonstrate substantially higher error rates in pure functional programming languages, such as Haskell and OCaml, compared to hybrid or imperative baselines. Furthermore, our analysis indicates that even when LLMs achieve functional correctness, they frequently generate low quality code that violate best practice in functional programming, thereby undermining the safety and maintainability benefits inherent to functional programming.

Despite these limitations, we also found that the code generation capabilities of LLMs in functional programming improves with model advancement. Additionally, our results demonstrated that LLMs hold potential on self-repairing their mistakes with guided by detailed feedbacks from static analysis and customized instructions. Overall, we hope that FPEval can serve as robust benchmark for the community and offer empirical insights to guide the development of LLMs in functional programming.

6 Limitations

The limitations of this work are as follows:

First, our evaluation is constrained by the limited scope of the subject models. Although we evaluate seven LLMs spanning both proprietary and open-source families,GPT-3.5, GPT-4o, and GPT-5 from OpenAI; Claude Opus 4.6 from Anthropic; DeepSeek-V3.2 from DeepSeek; and Qwen3-Coder-Next and Devstral Small 1.1 as open-source representatives, the landscape of available models is far broader. Some model families (e.g., Code Llama, StarCoder, Gemini) remain unevaluated, and our findings may not fully generalize to them. To minimize this risk, we selected models that span diverse providers, architectures, and capability tiers, including both the current industry-leading frontier models and competitive open-source alternatives, to ensure that our conclusions reflect a representative cross-section of the state of the art rather than the idiosyncrasies of a single vendor.

Second, this study mainly focuses on the code generation capabilities of LLMs in functional programming. While the generation of functionally-correct and high-quality code is a fundamental requirement for practical software engineering, it represents only a subset of the developer's workflow. Our evaluation does not account for other essential software development tasks, such as bug repair or code optimization. Consequently, our findings reflect the models' effectiveness in initial synthesis but may not generalize to their performance in iterative software evolution or debugging. However, we hope that our work can serve as a foundational benchmark for functional programming in the LLM era, providing a necessary baseline upon which future studies into broader software engineering automation can be built.

Finally, our benchmark primarily comprises problems sourced from educational and competitive programming platforms. While these tasks provide rigorous constraints for evaluating algorithmic correctness, they may not be fully representative of industrial-scale functional programming. Real-world software involves complex inter-module dependencies, legacy architectural constraints, and specific domain-driven design patterns that are often absent in isolated programming challenges. In the future work, we will consider the expansion of our dataset to include large-scale, open-source functional programming repositories.

References

Peter Achten. 2013. Why functional programming matters to me. In *The Beauty of Functional Code*.

Anthropic. 2025a. Claude Code: An agentic coding tool. Technical Release.

Anthropic. 2025b. Claude opus 4.6.

Anysphere. 2023. Cursor: The AI-first code editor. Accessed: 2025-11-19.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. 2026. Qwen3-coder-next technical report. *Preprint*, arXiv:2603.00729.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, Zekun Wang, Boyang Wang, Xianjie Wu, Bing Wang, Tongliang Li, Liqun Yang, Sufeng Duan, and Zhoujun Li. 2024. Mceval: Massively multilingual code evaluation. *Preprint*, arXiv:2406.07436.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53.

Matthew Farwell and Scalastyle Contributors. 2024. Scalastyle: Scala style checker. http://www.scalastyle.org. Accessed: 2024-05-20.

Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie Van Deursen, and David Sands. 2022. Propr: property-based automatic program repair. In *Proceedings of the 44th international conference on software engineering*, pages 1768–1780.

John Hughes. 1989. Why functional programming matters. *Comput. J.*, 32:98–107.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Majeed Kazemitabaar, Justin T.H. Chow, Carl Ka To Ma, Barb Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of ai code generators on supporting novice learners in introductory programming. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*.

Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Shuvendu K. Lahiri. 2024. Evaluating llm-driven user-intent formalization for verification-aware languages. *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pages 142–147.

Thanh Le-Cong, Bach Le, and Toby Murray. 2025a. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21991–22014, Vienna, Austria. Association for Computational Linguistics.

Thanh Le-Cong, Bach Le, and Toby Murray. 2025b. Memory-efficient large language models for program repair with semantic-guided patch generation. *Preprint*, arXiv:2410.16655.

LeetCode, Inc. 2025. Leetcode: The world's leading online programming learning platform. Accessed: 2025-12-15.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, Tao Sun, Jiaheng Liu, Yunlong Duan, Yu Hao, Liqun Yang, Guanglin Niu, Ge Zhang, and Zhoujun Li. 2025. Mdeval: Massively multilingual code debugging. *Preprint*, arXiv:2411.02310.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, 33(5).

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. Dafnybench: A benchmark for formal software verification. *Trans. Mach. Learn. Res.*, 2025.

Neil Mitchell. 2024. HLint: Haskell source code suggestions. https://github.com/ndmitchell/hlint. Accessed: 2024-05-20.

OCamlFormat Developers. 2024. OCamlFormat: Autoformatter for ocaml code. https://github.com/ocaml-ppx/ocamlformat. Accessed: 2024-05-20.

OpenAI. 2022. Introducing chatgpt. Accessed: 2025-12-29.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

OpenAI. 2025. Introducing gpt-5. Accessed: 2025-12-29.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8383–8394.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2785–2799. ACM.

James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett B. Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *ACM Transactions on Computer-Human Interaction*, 31:1 – 31.

Abhinav Rastogi, Adam Yang, Albert Q. Jiang, Alexander H. Liu, and 1 others. 2025. Devstral: Fine-tuning language models for coding agent applications. *Preprint*, arXiv:2509.25193.

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. 2014. A large scale study of programming languages and code quality in github. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

Tim Van Dam, Frank Van der Heijden, Philippe De Bekker, Berend Nieuwschepen, Marc Otten, and Maliheh Izadi. 2024. Investigating the performance of language models for completing code in functional programming languages: a haskell case study. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 91–102.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494.

Bowen Xu, Thanh-Dat Nguyen, Thanh Le-Cong, Thong Hoang, Jiakun Liu, Kisub Kim, Chen Gong, Changan Niu, Chenyu Wang, Bach Le, and David Lo. 2023. Are we ready to embrace generative ai for software q&a? *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1713–1717.

A Prompt for Synthesizing Test-Case Generators

To automatically construct high-quality and diverse test cases for each programming task, we employ GPT-4o to synthesize Python input-generation code directly from the natural-language task description. The model is prompted to behave as an expert competitive programmer and to strictly follow all problem constraints when constructing test data. Below is the exact prompt template used for generating test-case synthesis code. The assistant receives (i) the task description and (ii) a Python template scaffold, and is instructed to output a single Python function named construct_inputs() that returns 10 diverse inputs without printing them.

Prompt for input generation

```
    ("system",
     "You are an expert Python
         competitive programmer and your
          goal is to construct input
         generators for testing
         programming contest problems.
         You will write relevant
         generators and finally
         construct `construct_inputs`
         function that returns a list of
          diverse inputs sampled from
         the generator. Remember to
         strictly follow the
         instructions and constraints
         present in the problem
         statement.\n"
    ),
    ("human",
     "### INSTRUCTIONS:\n"
     "- Identify the constraints from
         the problem statement.\n"
     "- Construct a Python function `
         construct_inputs()` that
         generates diverse test cases.\n
         "
     "- Ensure all generated values
         strictly follow the constraints
         .\n\n"
     "### EXAMPLE REFERENCE:\n"
     "import numpy as np\n"
     "def random_input_generator(
         weight_min, weight_max,
         size_min, size_max):\n"
     "    weights_size = np.random.
         randint(size_min, size_max+1)\n
         "
     "    weights = np.random.randint(
         weight_min, weight_max, size=
         weights_size).tolist()\n"
     "    k = np.random.randint(1, len(
         weights)+1)\n"
     "    return weights, k\n\n"
     "def construct_inputs():\n"
     "    inputs_list = []\n"
     "    ## random inputs\n"
     "    for i in range(8):\n"
     "        inputs_list.append(
         random_input_generator(1, 50,
         1, 10))  # Adjust according to
         constraints\n"
     "    return inputs_list\n\n"
     "### TASK:\n"
     "{task_description}\n"
     "{python_template}\n"
     "Construct a random input generator
         . Use the format used in the
         above example by returning a
         single function that builds
         diverse inputs named `
         construct_inputs`. Do not print
          the inputs_list.\n"),
```

B Test Execution.

To ensure the execution of public and private test cases, we developed a specialized test infrastructure for functional languages that operates in four stages. First, FPEval generates per-problem unit tests using standard libraries (e.g., `OUnit2` for OCaml, `HUnit` for Haskell). Second, it constructs compilation-ready test templates, into which the LLM-generated solutions are injected. Third, the infrastructure validates both public and private test

cases by compiling and executing the resulting files in an isolated environment using Docker. Finally, it captures execution logs to rigorously distinguish between syntax errors, static type mismatches, runtime failures and functional errors. The infrastructure also automatically handles differences in data formats (e.g., booleans, lists, strings), supports docstring-based type extraction, and performs type-aware formatting of test inputs and expected outputs. This automation was essential for scaling evaluation across hundreds of problems and ensuring a high-quality assessment of LLM effectiveness in functional programming languages.

C Data Contamination.

We also investigate the potential impact of data contamination on the studied LLMs. We specifically select GPT-4o and DeepSeek-V3.2 for this analysis, as their clearly defined knowledge cutoffs provide a balanced distribution of tasks between the pre-cutoff and post-cutoff periods in our evaluation data. To quantify the impact of data contamination, we partition the programming tasks into distinct chronological periods based on their release dates. For GPT-4o, the tasks are partitioned into six periods: (1) before September 2022, (2) October 2022–March 2023, (3) April–September 2023, (4) October 2023–March 2024, (5) April–September 2024, and (6) after October 2024. Figure 2 illustrates the pass rates of GPT-4o in the four studied programming languages. We can see that GPT-4o shows a substantial post-cutoff performance drop in Java and Scala, i.e., from 52.0% to 32.8% in Java and from 35.8% to 21.6% in Scala. These results suggest possible data contamination in the pre-training data for mainstream languages. In contrast, the model maintains a low and stable accuracy in OCaml and Haskell throughout, consistent with their limited presence in pre-training corpora. To analyze DeepSeek-V3.2, we align the timeline with its official knowledge cutoff in July 2024, grouping tasks into: (1) before December 2022, (2) January–July 2023, (3) August–December 2023, (4) January–June 2024, (5) July–December 2024, and (6) after January 2025. Figure 3 illustrates its Pass@1 performance. Notably, DeepSeek-V3.2 exhibits a sharp and immediate post-cutoff performance decline beginning in the July–December 2024 period. In mainstream languages, accuracy drops from 55.4% to 38.0% in Java, and from 43.4% to 32.1% in Scala. Interestingly, unlike

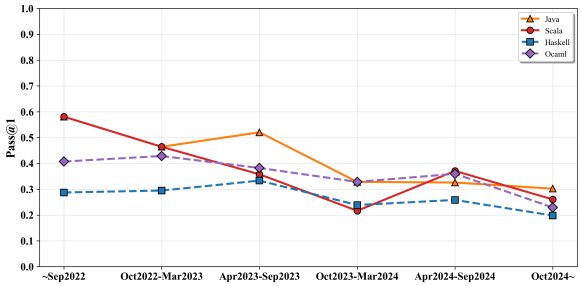

Figure 2: Pass@1 performance of GPT-4o

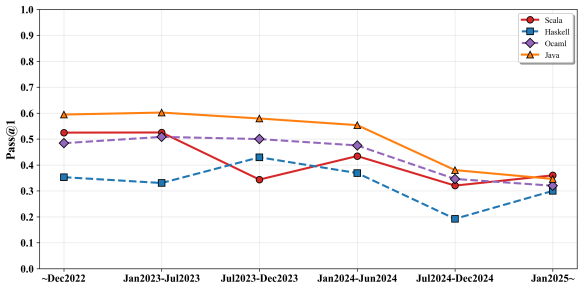

Figure 3: Pass@1 performance of Deepseek-V3.2

GPT-4o, DeepSeek-V3.2 also experiences a severe degradation in functional languages post-cutoff, with Haskell plummeting from 36.9% to 19.2% and OCaml dropping from 47.5% to 34.6%. This uniform drop across all studied languages strongly suggests extensive data contamination in its pre-training corpora, impacting both mainstream and niche programming languages prior to its July 2024 cutoff.

D Static Analysis Rule Categorization

To ensure the reproducibility of our code cleanliness metrics, we provide a categorization of the static analysis rules employed by `FPEval`. Table 5 details the specific configurations used for each language. These rules are selected to detect two primary types of violations:

- **Imperative Bias:** Constructs that are syntactically valid but violate functional programming principles (e.g., use of mutable variables, explicit loops, or procedural control flow).
- **Non-Idiomatic Usage:** Code that fails to leverage the expressive power of the language.

For **Haskell**, we rely on `HLint` to detect redundant constructs and suggest higher-order replacements. For **OCaml**, we invoke the compiler directly using the `-warn-error +a` flag to enforce strict static semantics. For **Scala**, we employ the

default `Scalastyle` configuration, which reflects the community's standard for idiomatic code. However, to prioritize semantic correctness over formatting minutiae, we explicitly disabled all whitespace-related rules (e.g., tab checking, spacing alignment), focusing strictly on logic and paradigm violations.

E Representative Examples of Imperative Code Patterns

1. Mutable Variables

- **Imperative-style (incorrect):**

```
var count = 0
count = count + 1


// Functional-style (preferred)

val count = 0
val newCount = count + 1
```

In FP, variables are ideally immutable (`val` in Scala, `let` without reassignment in OCaml). However, GPT often generates code using mutable bindings (`var`) or reassignment operators.

2. Imperative Loops

- **Imperative-style (incorrect):**

```
for (i <- 0 until n) {
  println(i)
}
```

- **Functional-style (preferred):**

```
(0 until n).foreach(println)
```

- **Or recursion (in OCaml):**

```
let rec print_numbers i n =
  if i < n then (
    print_int i;
    print_newline ();
    print_numbers (i + 1) n
  )
```

GPT frequently defaults to `for`, `while`, or `do while` loops common in imperative languages but less idiomatic in FP. A functional solution would use higher-order functions or recursion.

3. In-place Mutation of Data Structures

- **Imperative-style (incorrect):**

```
val arr = Array(1, 2, 3)
arr(0) = 42
```

- **Functional-style (preferred):**

```
val arr = Vector(1, 2, 3)
val newArr = arr.updated(0, 42)
```

Instead of treating data structures as immutable, GPT often produces code that mutates arrays or lists directly, especially in performance-related contexts. This contradicts the functional principle of constructing new values rather than mutating existing ones.

4. Use of `null` Instead of Option Types

- **Imperative-style (incorrect):**

```
def findUser(name: String): User = {
  if (db.contains(name)) db(name)
      else null
}
```

- **Functional-style (preferred):**

```
def findUser(name: String): Option[
    User] = db.get(name)
```

FP discourages the use of `null`, favoring safer alternatives like `Option`, `Maybe`, or pattern matching.

F Imperative Bias in LLM-Generated Functional Code

Although LLMs are prompted to generate code in functional programming languages such as Haskell, OCaml, and Scala, their outputs often reflect an *imperative programming bias*. This section provides representative examples and quantitative evidence of this phenomenon.

Example 1: Scala

- **Imperative-style (incorrect):**

```
import scala.collection.mutable

def countAlternatingSubarrays(nums:
    List[Int]): Int = {
    val n = nums.length
    // Mutable variable 'n'
    var count = 0
    //Mutable variable (var)
    var left = 0
    //Mutable variable (var)
    var right = 0
    // Mutable variable (var)
    val set = mutable.HashSet[Int]()
    // Uses mutable collection (
        HashSet)
    while (right < n) {
    //Imperative loop
        set.add(nums(right))
        // Side-effect: mutating set
```

Table 5: Categorization of Static Analysis Rules employed in FPEval

Language	Tool	Rule / ID	Description & Rationale
Haskell	HLint	*Use map / fmap*	Suggests replacing explicit recursion or list comprehensions with higher-order functions (e.g., `map`), enforcing idiomatic FP style.
		Redundant do	Flags `do` blocks containing a single expression, encouraging pure functional expressions over monadic sequencing.
		Use let	Recommends `let` bindings over monadic generators (`<-`) for pure values.
		Eta reduce	Encourages point-free style by removing redundant function arguments.
	GHC	*-Wall*	Enables all warnings, strictly flagging shadowed variables and unused matches common in imperative logic porting.
Scala	Scalastyle (Default Profile)	*VarChecker*	*(Standard)* Detects mutable variables (`var`), promoting the use of immutable values (`val`).
		ReturnChecker	*(Standard)* Detects the explicit `return` keyword, ensuring expression-oriented control flow.
		NullChecker	*(Standard)* Disallows `null`, enforcing type-safe error handling via `Option`.
		CyclomaticComplexity	*(Standard)* Penalizes methods with high complexity (> 10), often caused by nested imperative `if-else` blocks.
		PublicMethodsHaveType	*(Standard)* Requires explicit return types for public methods, ensuring type clarity in functional APIs.
OCaml	Compiler (`ocaml -c`)	*-warn-error +a*	**Strict Mode:** Treats all warnings as fatal errors. This ensures generated code adheres strictly to OCaml static semantics.
		Warning 8	*Partial match:* Enforced via strict mode. Rejects pattern matching that fails to cover all cases.
		Warning 26	*Unused variable:* Enforced via strict mode. Rejects unused bindings often left behind by procedural logic generation.
	OCamlFormat	*Formatting*	Enforces community-standard indentation, discouraging deep nesting structures typical of imperative code.

```
        while (set.size > 2) {
            set.remove(nums(left))
            //Side-effect: mutating
                set
            left += 1
            //Mutation
        }
        count += right - left + 1
        //Mutation
        right += 1
        //Mutation
    }

    count
}
```

Problem: Count alternating subarrays

- **Functional-style (preferred):**

```
def countAlternatingSubarrays(nums:
    List[Int]): Int = {
  nums.indices.foldLeft((0, 0, Map.
      empty[Int, Int], 0)) {
    case ((left, right, freqMap,
        count), r) =>
      val num = nums(r)
      val updatedFreq = freqMap.
          updated(num, freqMap.
          getOrElse(num, 0) + 1)

      // Slide left pointer while
          more than 2 distinct
          elements
      val (newLeft, newFreq) =
          Iterator.iterate((left,
          updatedFreq)) {
```

```
        case (l, fmap) =>
          val leftNum = nums(l)
          val newCount = fmap(
              leftNum) - 1
          val newFmap = if (newCount
               == 0) fmap - leftNum
              else fmap.updated(
              leftNum, newCount)
          (l + 1, newFmap)
      }.dropWhile { case (_, fmap) =
          > fmap.size > 2 }.next()

      val newCount = count + (r -
          newLeft + 1)
      (newLeft, r + 1, newFreq,
          newCount)
  }._4
}
```

The functional version uses an immutable ‘Map‘ instead of a mutable ‘HashSet‘, eliminates all ‘var‘ declarations, and replaces reassignment with state threading via ‘foldLeft‘. It also employs ‘Iterator.iterate‘ to express the sliding window logic in a recursion-inspired style.

Example 2: OCaml

- **Imperative-style (incorrect):**

```
let minTime (skill: int list) (mana:
     int list) : int =
  let n = List.length skill in
  let m = List.length mana in
  let dp = Array.make_matrix n m 0
      in
  (*uses mutable array*)
  dp.(0).(0) <- skill.(0) * mana.(0)
      ;
  (*mutation using "<-"*)
  for j = 1 to m - 1 do
  (*Imperative loop *)
    dp.(0).(j) <- max dp.(0).(j-1) (
        skill.(0) * mana.(j));
    (*Mutation inside loop*)
  done;
  for i = 1 to n - 1 do
  (*Imperative loop*)
    dp.(i).(0) <- dp.(i-1).(0) +
        skill.(i) * mana.(0);
    (*Mutation*)
    for j = 1 to m - 1 do
    (*Nested loop*)
      dp.(i).(j) <- max dp.(i).(j-1)
           dp.(i-1).(j) + skill.(i)
          * mana.(j);
      (*Mutation*)
    done;
  done;
  dp.(n-1).(m-1)
```

Problem: Find the minimum amount of time to brew potions

- **Functional-style (preferred):**

```
let minTime (skill: int list) (mana:
     int list) : int =
  let rec build_dp prev_row i =
    match List.nth_opt skill i with
    | None -> prev_row
    | Some s ->
      let row =
        List.mapi (fun j m ->
          let from_left = if j = 0
              then None else Some (
              List.nth row (j - 1))
              in
          let from_top = List.nth
              prev_row j in
          let cell_value =
            match from_left with
            | None -> from_top + s *
                m
            | Some l -> max l
                from_top + s * m
          in
          cell_value
        ) mana
      in
      build_dp row (i + 1)
  in
  match skill with
  | [] -> 0
  | s0 :: rest ->
    let first_row =
      snd (List.fold_left (fun (
          max_val, acc) m ->
        let curr = max max_val (s0 *
            m) in
        (curr, acc @ [curr])
      ) (min_int, []) mana)
    in
    let final_dp = build_dp
        first_row 1 in
    List.hd (List.rev final_dp)
```

In this version:

- The dp matrix is reconstructed immutably using List.fold_left and pure functions.
- There are no imperative loops or mutation (<- is gone).
- Recursion and list transformations maintain purity and immutability.

Example 3: Haskell

- **Imperative-style (incorrect):**

```
import Data.List
maxOperations :: [Int] → Int
maxOperations nums = helper nums 0
  where
    helper nums count
      | length nums < 2 = count
      | otherwise = let score = sum
          $ take 2 nums
            in helper (delete (head
                nums) $ delete (head
                 $ reverse nums)
                nums) (count + 1)
```

Problem: Maximum number of operations with the same score ii

- **Functional-style (preferred):**

```
maxOperations = go 0
  where
    go count (x:xs@(_:_)) =
      let mid = init xs
      in go (count + 1) mid
    go count _ = count
```

G Prompt Design and Language-specific Self-repair Instructions

This appendix details the prompting strategies and language-specific repair mappings used in our experiments. It consists of two parts: (1) the prompt templates for code generation and self-repair, and (2) the mapping between common compiler warnings or errors and the corresponding self-repair instructions for each functional language (Haskell, OCaml, and Scala).

G.1 Prompt Templates

The first part presents the exact prompts used for the baseline code generation and self-repair settings.

Prompt for generating code

```
    ("system",
     "You are an expert {lang}
         programmer. You will be given a
          question (problem
         specification) and will
         generate a correct {lang}
         program that matches the
         specification and passes all
         tests. You will NOT return
         anything except for the program
          AND necessary imports.\n",
    ),
    ("human",
     "### QUESTION:\n{description}\n"
     "### FORMAT: You will use the
         following starter code to write
          the solution to the problem
         and enclose your code within
         delimiters.\n{template}\n"
     "### ANSWER: (use the provided
         format with backticks)\n"
     )
```

Prompt Baseline for self-repair

```
    ("system",
    "You are a helpful programming
        assistant and an expert {lang}
        programmer. The previously
        generated code has quality
        issues, does not pass the tests,
         and is not idiomatic functional
         programming. Please provide a
        better code implementation as
        expected by the task description
         and the function using
        idiomatic functional programming
         style. You must put the entire
        fixed program within code
        delimiters only once.",
    ),
    ("human",
    "### CODE: \n{pre_code}\n"
    )
```

G.2 Language-specific Self-repair Mappings

The second part summarizes detailed mappings between compiler feedback and the corresponding self-repair instructions. These mappings serve as structured guidance for the model to correct errors and improve code style and maintainability during the self-repair process.

H Supplementary Experimental Results

I Preliminary Pass@k Analysis

To investigate whether increasing the number of sampled candidates improves performance, we conducted a preliminary evaluation of pass@10 using DeepSeek-V3.2 on a random subset of 300 tasks from FPBench. For each task we generated ten independent candidate solutions and marked the task as successful if at least one solution passed all test cases.

As shown in Table 10, pass@10 yields consistent improvements over pass@1 across all three languages (e.g., +15 pp in Haskell, +17.5 pp in OCaml, +15 pp in Scala), confirming that wider sampling raises the ceiling of solvable tasks. However, these correctness gains come at a cost to code quality. Among solutions that first succeeded within the pass@10 range (i.e., failed at pass@1 but passed within ten attempts), the vast majority still exhibit style or maintainability violations: 78.95% in Haskell, 65.43% in OCaml, and 83.97% in Scala. This suggests that the additional correct solutions found through broader sampling tend to rely on imperative shortcuts rather than idiomatic functional code, consistent with the "reward-hacking" pattern documented in Section 3.2. We report pass@1 as the primary metric throughout the paper because it reflects realistic single-query deployment settings; extending this analysis to additional models and the full benchmark is left for future work.

Table 6: Instruction mapping for Haskell self-repair

Warning / Error	Instruction for Self-repair
Avoid lambda	Avoid using lambda expressions when a function or operator already exists.
Eta reduce	Remove unnecessary parameters by applying eta-reduction.
Redundant bracket	Eliminate redundant brackets to simplify the expression.
Avoid lambda using infix	Replace lambda expressions with infix operators when possible.
Use zipWith	Use `zipWith` for parallel list operations instead of combining `map` and `zip`.
Avoid reverse	Avoid using `reverse` before `head`, `last`, or indexing, as it is inefficient.
Move brackets to avoid $	Prefer parentheses over excessive use of the `$` operator for readability.
Move filter	Apply `filter` closer to the data source to reduce unnecessary computation.
Use uncurry	Use `uncurry` when applying a function to a tuple.
Use infix	Use infix notation for better readability when working with operators.

Table 7: Instruction mapping for OCaml self-repair

Warning / Error	Instruction for Self-repair
Type error	Check type annotations and ensure expressions match expected types. For example, if a function expects an `int` but receives a `string`, convert or adjust the type.
Unbound identifier	Declare the identifier before using it, or fix the spelling. Make sure the correct module is opened/imported.
Parse error	Fix OCaml syntax: check missing `in`, `->`, `;;`, or misplaced parentheses/brackets.
Unused	Remove unused variables, or prefix them with _ if they are intentionally unused.
Exhaustiveness error	Add missing cases in pattern matching to cover all constructors, or use _ as a catch-all case.
Incorrect arity	Check the number of arguments when calling a function. Supply missing arguments or remove extra ones.
Missing labeled argument	Add the required labeled argument when calling the function (e.g., `f ~x:value` if the function expects `~x`).
Mutation of immutable	OCaml values are immutable by default. Rewrite code to avoid mutation, and use new bindings (`let x = ...`) instead of trying to update old ones.

J Transpiler Verification Details

To ensure the semantic integrity of the transpiled templates, we implemented a two-step verification process. All generated templates underwent automated syntax and type checking (via `ghc`, `ocamlopt`, and `scalac`), followed by a manual audit of full the dataset to confirm the strict preservation of problem constraints and I/O mappings. The automated stage rejected 4.3% of templates overall (6.1% for Haskell, 2.9% for OCaml), primarily due to unsupported type constructs (e.g., nested generics, callback types) and ambiguous overloaded operators that could not be resolved without runtime context; these tasks were excluded from the benchmark. In the manual audit, we observed some minor differences in I/O formatting, all of which were corrected before finalizing the dataset. This rigorous verification process ensures that the templates used for evaluation are both syntactically valid and

Table 8: Instruction mapping for Scala self-repair

Warning / Error	Instruction for Self-repair
If block needs braces	Always wrap `if` blocks with braces `{}` for clarity and to avoid ambiguity.
Avoid using return	Do not use `return`; rely on expression values instead.
Magic Number	Replace magic numbers with named constants or enums for readability.
Cyclomatic complexity of	Refactor the function into smaller ones or use functional constructs (`map`, `fold`, recursion) to reduce complexity.
There should be a space before the plus (+) sign	Add proper spacing around operators (e.g., `a + b` not `a+b`).
File line length exceeds	Split long lines into multiple shorter lines for readability.
File must end with newline character	Ensure the file ends with a newline character.

Table 9: Proportion of clean code correct outputs generated by studied LLMs. Origin, Sim. Rep., and Ins. Rep. denote results obtained from the original generated code, code repaired using `Simple Repair`, and code repaired using `Instruction-guided Repair`, respectively.

Model	Method	Haskell	OCaml	Scala
GPT-3.5	Origin	22%	63%	61%
	Sim. Rep.	46%	78%	75%
	Ins. Rep.	**61%**	**87%**	**79%**
GPT-4o	Origin	46%	61%	**77%**
	Sim. Rep.	48%	**92%**	61%
	Ins. Rep.	**56%**	78%	66%
GPT-5	Origin	45%	44%	57%
	Sim. Rep.	58%	87%	**62%**
	Ins. Rep.	**63%**	**91%**	60%
DeepSeek-V3.2	Origin	33%	36%	65%
	Sim. Rep.	46%	**49%**	67%
	Ins. Rep.	**47%**	43%	**71%**
Devstrall Small 1.1	Origin	36%	50%	**63%**
	Sim. Rep.	38%	53%	61%
	Ins. Rep.	**48%**	**59%**	62%
Qwen3-Coder-Next	Origin	51%	43%	64%
	Sim. Rep.	54%	48%	**68%**
	Ins. Rep.	**55%**	**56%**	67%

semantically faithful to the original problem specifications, providing a reliable foundation for assessing LLM performance in functional programming.

K Violations of Core Linguistic Properties.

To provide a deeper investigation into the nature of the imperative bias, we extend our analysis to explicitly quantify violations of three core linguistic principles of functional programming: (1) *declarative control flow* (reliance on explicit loops or `if-then-else` chains instead of pattern matching or higher-order functions); (2) *purity* (presence of IO effects, monadic misuse, or hidden side effects in nominally pure functions); and (3) *immutability* (use of mutable references or in-place assignments). Table 11 reports the violation rates across DeepSeek-V3.2, Devstral Small 1.1, Qwen3-Coder-Next, and Claude Opus 4.6.

Table 10: Pass@1 vs. Pass@10 for DeepSeek-V3.2 on 300 FPBench tasks

Metric	Haskell	OCaml	Scala
Pass@1	34.0%	44.5%	48.0%
Pass@10	49.0%	62.0%	63.0%

Violations of declarative control flow are the most prevalent across all models and languages, exceeding 68% in Haskell for the strongest models and remaining between 31% and 47% in OCaml and Scala. This indicates a systematic reversion to sequential, imperative constructs even in languages that strongly discourage them. Immutability violations are strongly language-dependent: rare in Haskell (under 5%), reflecting stricter compiler enforcement, but common in OCaml and Scala (often exceeding 40%), where the language permits but discourages mutable state. Purity violations occur at lower rates (approximately 5–25%) but are present across all languages, suggesting recurring difficulty in preserving referential transparency and correct monadic structure. Together, these findings confirm that the imperative bias documented in Table 3 reflects systematic, multi-dimensional failures to adhere to functional programming principles, not merely superficial stylistic choices.

Table 11: Percentage of LLM-generated code with violations of core functional programming properties. Decl. Ctrl. Flow = Declarative Control Flow.

Model	**Property**	**Haskell**	**OCaml**	**Scala**
DeepSeek-V3.2	Decl. Ctrl. Flow	75.92%	37.53%	36.50%
	Purity	5.78%	13.21%	16.40%
	Immutability	1.04%	41.08%	47.10%
Devstral Small 1.1	Decl. Ctrl. Flow	47.01%	32.34%	42.20%
	Purity	25.41%	7.48%	11.00%
	Immutability	0.76%	41.45%	46.80%
Qwen3-Coder-Next	Decl. Ctrl. Flow	72.50%	39.33%	42.96%
	Purity	11.02%	18.00%	12.81%
	Immutability	0.40%	36.78%	44.09%
Claude Opus 4.6	Decl. Ctrl. Flow	68.33%	31.69%	46.39%
	Purity	11.65%	17.31%	11.35%
	Immutability	4.69%	49.25%	41.88%