Summary-Mediated Repair: Can LLMs use code summarisation as a tool for program repair?

Lukas Twist
King's College London
lukas.twist@kcl.ac.uk

Abstract

Large Language Models (LLMs) often produce code with subtle implementation-level bugs despite strong benchmark performance. These errors are hard for LLMs to spot and can have large behavioural effects; yet when asked to summarise code, LLMs can frequently surface high-level intent and sometimes overlook this low-level noise. Motivated by this, we propose *summary-mediated repair*, a prompt-only pipeline for program repair that leverages natural-language code summarisation as an explicit intermediate step, extending previous work that has already shown code summarisation to be a useful intermediary for downstream tasks. We evaluate our method across eight production-grade LLMs on two function level benchmarks (`HumanEvalPack` and `MBPP`), comparing several summary styles against a direct repair baseline. Error-aware diagnostic summaries consistently yield the largest gains–repairing up to 65% of unseen errors, on average of 5% more than the baseline–though overall improvements are modest and LLM-dependent. Our results position summaries as a cheap, human-interpretable diagnostic artefact that can be integrated into program-repair pipelines rather than a stand-alone fix-all.

1 Introduction

Large language models (LLMs) have recently made great strides in their code generation abilities, achieving impressive results on a variety of benchmark tasks [3, 10]. However, even the most state-of-the-art LLMs frequently produce incorrect code: in particular, short implementation-level errors (off-by-one errors, incorrect boundary checks, small interface misuses) commonly survive generation [14]. These subtle failures are important in practice because they can be difficult to diagnose while having a significant semantic effect, and they can often avoid follow-up repair prompts [29].

One reason why these errors are hard for LLMs is that they can hinge on single-character distinctions (for example a "<" instead of "<=") which are sometimes obscured by the LLMs' token encodings and internal representations [2]. In contrast, when asked to describe code in natural language, LLMs can often focus their attention on the bigger picture and understand the expected behaviour of the code [26], potentially overlooking these subtle errors. Previous work has already shown that natural language summaries can serve as useful intermediate artefacts for downstream tasks such as code style normalisation [13] and code modification [27], suggesting that summaries may also be a viable bridge between programme text and intended behaviour.

Motivated by this observation, we propose a simple, prompt-only pipeline for program repair that leverages natural-language code summarisation as an explicit intermediate step, and does not require a formal program specification. We call our method ***summary-mediated repair***: first, we have the LLM summarise the code, then we have the LLM generate new code conditioned on that summary. The idea is intentionally minimal, so that any improvement can be attributed to the summary artefact itself rather than to complex orchestration or heavy tooling.

We evaluate *summary-mediated repair* across eight production-grade LLMs on two widely used function-level benchmarks: using existing bugs from the `HumanEvalPack` [20] dataset and the LLMs' own failed generations of the `MBPP` [1] dataset. In both cases, we compare the results of varying summary styles (for example concise, intent-focussed, or error-aware) against a *direct repair* baseline, to understand whether the use of summaries can help the LLM to improve its inherent abilities to infer expected functionality and repair code.

We observe that *summary-mediated repair* is capable of helping LLMs to repair code, but the gains are modest and summary-dependent. Error-aware summaries–those that explicitly tell the LLM to expect faults and be suspicious of the code–consistently yield the best results, repairing up to 65% of unseen errors (on average 5% more than the baseline). In contrast, when asking an LLM to repair its own initially failing generations, both the *direct repair* baseline and *summary-mediated repair* struggled, with only 11% of errors repaired at most; this suggests that more sophisticated techniques are likely needed for reliable self-repair via prompt-only methods.

Overall, our results position summaries as a diagnostic artefact that can be integrated into program repair pipelines rather than a stand-alone fix-all. They are low-cost to run, easy for humans to inspect, and particularly helpful for surfacing subtle, implementation-level errors that other techniques may miss, especially in cases where a formal specification may be unavailable. Although not state-of-the-art on their own, our results build on prior work to further advocate for the use of code summaries as natural language intermediate artefacts for downstream tasks.

Our contributions are as follows:

(1) We introduce *summary-mediated repair*, a novel, prompt-only technique that uses natural language code summaries as an intermediate artefact for program repair.
(2) We evaluate our *summary-mediated repair* across eight LLMs, exploring multiple summary styles, and showing that diagnostic summaries can improve the repair of subtle bugs.
(3) We release our code, results, and evaluation harness publicly via our GitHub repository, to encourage further investigation in this area: *https://github.com/itsluketwist/summary-mediated-repair/*

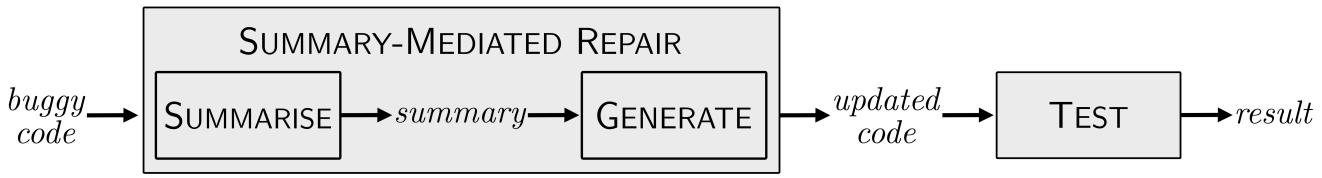

Figure 1: *Summary-Mediated Repair.* Our proposed pipeline for a prompt-only APR method, where a code summary is generated as an intermediate artefact. Full details in Section 3.

2 Related Work

Code Generation. LLMs trained on massive code corpora have been shown to excel at code generation [10], achieving impressive pass rates on functional benchmarks such as `HumanEval` and powering tools such as GitHub Copilot [4]. However, further evaluations reveal correctness gaps: LLMs frequently fail on corner cases, produce subtle semantic errors, and replicate bugs from training data [5]. Benchmarks that emphasise real-world scenarios–such as `HumanEvalPack` [20] and `SWE-Bench` [11]–highlight these limitations and motivate research into methods that improve robustness of the generated code.

Code Summarisation. Source code summarisation has long been studied as a way to automatically document and explain code [25]. Recent work shows that modern LLMs can produce accurate intent-capturing summaries of code [26]. LLM-produced summaries have already been used as intermediate artefacts for downstream tasks such as style normalisation [13] and developer-mediated code modification [28]. These findings motivate our investigation of whether code summaries can also serve as semantically rich intermediate artefacts for the task of program repair.

Code Repair. Automated program repair (APR) has been extensively studied, evolving from search- and genetic-based methods to template- and constraint-based [8, 12], and more recently to LLM-driven approaches [29]. Despite these advances, persistent problems–notably "plausible-but-incorrect" patches and weak fault localization–continue to make small-scale semantic fixes unreliable [27]. In addition, many of the techniques can struggle when there only exists a weak oracle to check for correctness [30]. This motivates exploration of techniques that can specifically be used to fix subtle errors, and are robust to the lack of a formal specification of the intended behaviour.

3 Summary-Mediated Repair

Our method is deliberately simple: we use code summarisation as an intermediate stage in a prompt-only APR process that does not require a formal specification. A typical APR process takes the original *code* as input, along with some *error* description, produces a candidate $code' = \textsc{Repair}(code, error)$, and validates it on some test harness to get $result = \textsc{Test}(code')$. Our Repair method has two stages and only requires the original *code*, as shown in Figure 1. We first prompt an LLM to generate a summary, $summary = \textsc{Summarise}(code)$, describing the intended behaviour of *code*. We then prompt the LLM to generate a repaired program using the *summary*, $code' = \textsc{Generate}(summary)$, and validate on the same test harness to get $result = \textsc{Test}(code')$.

The simplicity of this pattern means that any observed benefit can be attributed to the summary itself rather than any complex orchestration or heavy tooling. Additionally, there is a lot of flexibility with the exact prompts used in both stages: Summarise could request a concise one-line summary, a detailed list of behaviour, or even an error-aware summary; Generate could be a basic prompt, or provide more detailed instructions on how to interpret the summary.

Although we do not expect this method to universally fix all types of bug, it is possible that it can help LLMs overcome subtle semantic errors as part of a larger APR pipeline, particularly when there is a lack of a formal specification. An "off-by-one" bug, for example, may be caused by a single < instead of <=, or a missing +1 – a tiny textual change with potentially large semantic effect. Depending on its token encodings and internal representations, an LLM can be insensitive to single-character changes [2], and can therefore miss such character-level mistakes when asked to repair code directly. Yet, the same LLM may still correctly describe the overall intended behaviour in a natural-language summary, such that when prompted to generate code from the summary, the error is resolved. We therefore investigate whether code summaries can externalise behaviour-level information that helps LLMs avoid repeating subtle implementation-level mistakes.

4 Experimental Design

This section states the research questions (RQs) and describes the experiments we run to evaluate the effectiveness of *summary-mediated repair* for repairing function-level errors.

4.1 Research Questions

RQ1: **Bug Repair**: *How effective is summary-mediated repair on existing buggy code when using different styles of summaries?* To answer this RQ, we use *summary-mediated repair* to fix buggy code from the `HumanEvalPack` dataset.

RQ2: **Self Repair**: *How effective is summary-mediated repair at helping an LLM to fix errors in its own code when using different styles of summaries?* To answer this RQ, we ask different LLMs to solve tasks from `MBPP`, and use *summary-mediated repair* to try and fix any incorrect solutions.

4.2 LLM Selection

We use a range of LLMs to assess *summary-mediated repair* and enable a broad understanding of its effectiveness. In particular, we

want to experiment on a range of LLM sizes (number of parameters), different use-cases (general or code-specific), varying availabilities (open or closed source), and built by different organisations. Therefore, we choose eight different models from four different providers (OpenAI, Llama, Mistral and Qwen). *Full details given in Table 1.*

LLM Configuration. As is standard in code generation evaluation, we focus on low-variance single sample metrics (such as *pass@1* [23]) and therefore use conservative decoding settings (*temperature* = 0.2 and *top_p* = 1.0) to reduce sampling noise and make LLM comparisons fair. A low, non-zero *temperature* value reduces randomness in responses and is a suitable setting when comparing code generation performance for single-sample runs [4]. Similarly, the *top_p* parameter controls nucleus sampling and is typically used to improve diversity in responses; therefore, we choose a value of 1.0 to further reduce randomness [7]. Furthermore, we conduct each LLM interaction in a fresh API session to avoid bias from prompt caching or leakage [6]; and we do not use a system prompt to ensure that each LLM has its base functionality considered [19].

Table 1: *LLMs Configuration.* Details of all LLMs used in this study. Entries marked with "–" indicate that the information was not available at the time of the study (September 2025).

Model	Version	Release	Knowledge cut-off	Size	Open-source?	Code model?
GPT-4o-mini [22]	gpt-4o-mini-2024-07-18	July '24	Oct. '23	-	✗	✗
GPT-4.1-mini [21]	gpt-4.1-mini-2025-04-14	Apr. '25	Jun. '24	-	✗	✗
Llama-3.3 [15]	llama-3.3-70b-instruct-turbo	Dec. '24	Dec. '23	70B	✓	✗
Llama-4 [16]	llama-4-scout-17b-16e-instruct	Apr. '25	Aug. '24	109B	✓	✗
Qwen-2.5-Coder [9]	qwen2.5-coder-32b-instruct	Sep. '24	Mar. '24	32B	✓	✓
Qwen-2.5-Turbo [24]	qwen2.5-72b-instruct-turbo	Sep. '24	Mar. '24	72B	✓	✗
Codestral [17]	codestral-2501	Jan. '25	-	-	✗	✓
Mistral-Medium [18]	mistral-medium-2505	May '25	-	-	✗	✗

4.3 Prompt Strategy

Our *summary-mediated repair* method (as defined in Section 3) intentionally separates the Summarise and Generate steps, and allows for a lot of flexibility in the prompts used in both. We keep the Generate prompt consistent in all experiments and vary only the Summarise prompt. This isolates the *summary* artefact and reduces the number of variables, making it easier to attribute any observed differences to how the code is summarised, rather than the generation instructions. We also explicitly choose function-level tasks for our experiments because they provide a controlled, reproducible setting that is appropriate for an initial, focused assessment of our method.

Summarisation prompts. We compare the effectiveness of five prompts for the Summarise step. We do not aim to find the absolute best prompt; we sample sensible, interpretable prompt styles to get an initial assessment of how effective our method is and how impactful summary style is. The summary prompts are as follows:

Base – A neutral prompt with no extra guidance:

*"Summarise the following python code: **{code}**"*

Short – Ask for a concise, one-sentence summary:

*"Summarise the following python code in one sentence: **{code}**"*

Intent – Instruct the LLM to infer the original intent of the code:

*"Summarise the following python code, be smart and infer expected functionality: **{code}**"*

Error – Explicitly tell the LLM that the code contains an error:

*"Summarise the following python code, it contains at least one bug that is making a test fail: **{code}**"*

Warn – An error-aware prompt that warns about potential bugs:

*"Summarise the following python code, it is not very well written, and might have some bugs, so please work around this when writing the summary: **{code}**"*

Generation prompt. After summarising the failing code, we need to generate an updated version. To ensure that the final output is directly executable by the unit-test harnesses used in our experiments, we use a Generate prompt that requests a function definition:

*"Write a python function '**{function}**', satisfying the following code summary: **{summary}**"*

Baseline prompt. We also need a baseline against which to compare our results. Our method does not require the initial specification in order to attempt a repair; therefore, we do not include the initial specification for the baseline prompt either. We use a simple prompt that directly requests the LLM to repair a given piece of code:

*"Repair the following python code: **{code}**"*

4.4 Experimental Setup for RQ1: Bug Repair

For this RQ we want to investigate whether *summary-mediated repair* is effective on existing code that contains bugs. To do this we use the `HumanEvalPack` dataset [20] – an extension of OpenAI's `HumanEval` [4] dataset that covers three distinct tasks, one of which is function-level APR. `HumanEvalPack` provides a candidate solution (with a simple, manually inserted bug) and test cases for each of the 164 tasks from the original dataset. For each task, we run the incorrect solution through our *summary-mediated repair* pipeline (defined in Section 3), and validate the returned code using the provided tests. We consider a repair successful if all tests now pass.

We report single-sample results, defining the metric *fix@1*: the percentage of bugs fixed after one execution of the *summary-mediated repair* pipeline.

4.5 Experimental Setup for RQ2: Self Repair

While RQ1 evaluates *summary-mediated repair* on existing buggy programs; RQ2 examines whether our approach can help LLMs to repair their own incorrect code generations. We choose the `MBPP` [1] dataset–which contains 974 function-level `Python` programming tasks, each with a natural language task description and corresponding test cases–and use the 500 tasks from its test split. We first prompt the LLM to produce an initial solution and check correctness against the provided test cases. If the solution fails, we

Table 2: *RQ1: Bug Repair.* Results for LLMs fixing buggy code from the HumanEvalPack dataset using *summary-mediated repair* with different styles of summary. For each LLM and repair method we give the *fix@1* rate, the percentage of bugs fixed after one attempt of the repair method.

Repair Method		GPT-4o-mini	GPT-4.1-mini	Llama-3.3-70B	Llama-4-Scout	Codestral	Mistral-Medium	Qwen-Coder	Qwen-Turbo
		fix@1	*fix@1*	*fix@1*	*fix@1*	*fix@1*	*fix@1*	*fix@1*	*fix@1*
Direct repair (baseline)		54.88%	57.93%	42.07%	43.90%	51.83%	54.88%	52.44%	60.98%
Summary-mediated repair	Base	37.20%	26.22%	37.20%	39.02%	37.20%	37.20%	43.90%	45.12%
	Short	31.71%	22.56%	29.88%	39.02%	37.20%	35.37%	34.76%	40.85%
	Intent	33.54%	31.10%	37.20%	36.59%	39.63%	36.59%	47.56%	44.51%
	Error	57.32%	64.63%	52.44%	51.83%	57.32%	56.10%	58.54%	60.98%
	Warn	46.34%	33.54%	39.63%	37.20%	45.73%	41.46%	50.00%	56.71%

apply the *summary-mediated repair* pipeline (defined in Section 3), using the failing code as input to generate a new version. We again evaluate this new code by using the corresponding test cases to check for correctness.

We report two metrics: *fix@1*, defined as the proportion of initially incorrect solutions that are successfully repaired after one execution of the *summary-mediated repair* pipeline; and an adjusted *pass@1*, representing the overall proportion of tasks correctly solved after the repair process has been applied.

5 Results

This section presents the results of our experiments into the effectiveness of *summary-mediated repair* for varying styles of summaries.

5.1 Results for RQ1: Bug Repair

This RQ investigates whether *summary-mediated repair* is effective on existing buggy code. Table 2 reports the *fix@1* rates for each LLM and repair method on the HumanEvalPack dataset.

Summary-mediated repair shows potential, all summary prompts were able to repair at least 22.6% of errors, but its effect is inconsistent. The error-aware prompts (*error* and *warn*), where the LLM is told to expect errors in the code, provide the best results, with *fix@1* rates of 57.4% and 43.8% respectively. Yet, only the *error* summary outperformed the *direct repair* baseline, doing so by 5.0% on average. This indicates that it is possible to use *summary-mediated repair* to infer functionality and fix existing buggy code; and asking the LLM to treat the input as potentially incorrect produces more useful summaries for this purpose.

The effectiveness of *summary-mediated repair* is also LLM dependent and does not appear to be linked to the baseline *fix@1* rate for *direct repair*. Qwen LLMs have the highest average *fix@1* rates across summary styles (47.0% for Qwen-Coder and 49.6% for Qwen-Turbo), but they also have high baselines. LLama LLMs on the other hand have some of the lowest average *fix@1* rates (39.3% for LLama-3.3-70B and 40.7% for LLama-4-Scout), but they are the most comparable to their baselines. In general, we observe no correlation between an LLMs ability to directly repair the code, and the effectiveness of our method.

Case analysis. To understand when summary-mediated repair is most effective, we examine HumanEvalPack records that every summary variant successfully fixes but the *direct repair* baseline fails to repair. There are 24 such cases. The most common underlying bug types are "value misuse" (9 cases, typically involving subtle mistakes in how variables are interpreted) and "excess logic" (6 cases, involving unnecessary or misleading logical steps). These instances highlight scenarios where summaries help the model recover the intended functionality at a higher level. The summaries steer the LLM toward the core behaviour, whereas the baseline prompt often misidentifies the bug, for example, focussing on output formatting rather than a subtle logical flaw.

> **Answer to RQ1**: *Summary-mediated repair* can fix existing buggy code, but is inconsistent and summary-dependent. Explicitly telling the LLM that the code has an error when generating the summary produces the best results, fixing **57.4%** of code on average across LLMs, an increase of **5.0%** on the baseline.

5.2 Results for RQ2: Self Repair

This RQ investigates whether *summary-mediated repair* is effective when helping an LLM to fix errors in its own generated code. Table 3 reports the *fix@1* and adjusted *pass@1* rates for each LLM and repair method on the MBPP dataset.

Generally, LLMs struggled to repair their own code with both *direct repair* prompts and our *summary-mediated repair* method. We observed much lower *fix@1* rates for all summaries and LLMs (all less than 10%, most less than 5%). The *error* summary still performs best, fixing an average of 5.2% of code errors across LLMs. Perhaps surprisingly, the *short* summary performs second best, indicating that a more detailed summary may not be best when an LLM needs to fix its own code, and a more concise description may help the LLM to take another attempt at a solution. Although small, these improvements are non-trivial and are much more similar to the *direct repair* baseline than for RQ1, indicating that many MBPP tasks are either already solved by the LLMS in our study, or not fixable by simple prompt-engineering strategies.

Again, the effects are highly dependent on the LLM and summary prompt in question. Qwen-Coder has by far the highest success rate at fixing its own bugs, with an average *fix@1* rate of 7.2% across summaries – more than double any other LLM. Some LLM/prompt combinations do show meaningful improvements though: Codestral has a *fix@1* rate of 5.4% for the *short* summary, up from a 3.0%

Table 3: *RQ2: Self Repair.* Results for LLMs solving tasks from the MBPP dataset, then fixing any errors using *summary-mediated repair* with different styles of summary. For each LLM and repair method we give the *fix@1* rate, the percentage of initially incorrect solutions fixed with one attempt of the repair method; and an adjusted *pass@1* rate, the overall proportion of tasks correctly solved after repair.

Repair Method		GPT-4o-mini		GPT-4.1-mini		Llama-3.3-70B		Llama-4-Scout		Codestral		Mistral-Medium		Qwen-Coder		Qwen-Turbo	
		fix@1	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*	*fix@1*	*pass@1*
Solve rate (no repair)		–	66.60%	–	75.20%	–	67.00%	–	65.40%	–	66.60%	–	70.80%	–	77.80%	–	75.20%
Direct repair (baseline)		3.59%	67.80%	3.23%	76.00%	4.85%	68.60%	2.31%	66.20%	2.99%	67.60%	4.79%	72.20%	10.81%	80.20%	4.03%	76.20%
Summary-mediated repair	Base	1.20%	67.00%	1.61%	75.60%	2.42%	67.80%	1.73%	66.00%	1.20%	67.00%	1.37%	71.20%	6.31%	79.20%	0.81%	75.40%
	Short	3.59%	67.80%	1.61%	75.60%	3.64%	68.20%	4.05%	66.80%	5.39%	68.40%	4.11%	72.00%	8.11%	79.60%	4.03%	76.20%
	Intent	1.80%	67.20%	1.61%	75.60%	1.82%	67.60%	2.31%	66.20%	1.20%	67.00%	2.74%	71.60%	3.60%	78.60%	0.00%	75.20%
	Error	4.79%	68.20%	4.84%	76.40%	3.64%	68.20%	3.47%	66.60%	1.80%	67.20%	5.48%	72.40%	9.91%	80.00%	8.06%	77.20%
	Warn	3.59%	67.80%	2.42%	75.80%	3.03%	68.00%	4.05%	66.80%	1.20%	67.00%	2.05%	71.40%	8.11%	79.60%	3.23%	76.00%

baseline and 1.4% average across the other summaries; and Qwen-Turbo has a *fix@1* rate of 8.1% for the *error* summary, when other summaries show little to no ability to fix bugs.

Case analysis. For MBPP, we found no cases in which a failed task was repaired by all summary variants but not by the *direct repair* baseline, preventing a meaningful case analysis. In general, few methods performed well on any given task, and there was no consistent pattern indicating when *summary-mediated repair* outperformed the baseline. In most cases, the LLM either recognised its own error or it did not, and the specific prompting strategy had minimal influence on this outcome.

> **Answer to RQ2**: *Summary-mediated repair* can help LLMs to repair a small but non-trivial fraction of their failing generations. Error-aware prompts performed the best, fixing an average of **5.2%** of an LLMs errors. Less detailed summaries showed promise for allowing an LLM to retry code generation.

6 Discussion

Practical Implications. Our experiments show that natural language code summaries have potential as low-cost, intermediate artefacts in APR pipelines. They do not achieve state-of-the-art APR on their own, but they offer a novel, practical way to help LLMs fix code, particularly when a formal specification is unavailable. Error-aware, diagnostic styles of summary–those that ask the LLM to expect errors or flag suspicious code–have the best results; naive summaries, by contrast, can often strip away useful implementation detail and hurt repair. When an LLM fails to solve a task initially, however, *summary-mediated repair* usually yields only marginal gains: this indicates that our method can assist with surfacing an LLMs ability to fix newly seen bugs, but is unlikely to enhance its general ability to solve coding problems. Practically, summaries should be seen as a lightweight addition to an APR workflow – inexpensive to compute, easy for humans to reason about, and most useful when integrated alongside complementary techniques such as fault localisation, basic behavioural checks, or human-in-the-loop adjustments.

Limitations. There are several important limitations to be aware of. First, we intentionally focus our evaluation on function-level tasks (HumanEvalPack, MBPP) because these problems are small and self-contained, allowing controlled, isolated evaluation; the results may not generalise to more complex, project-level bugs. Second, our reported metrics are single-sample (*fix@1*, *pass@1*) and therefore sensitive to decoding and sampling choices; we mitigate this by using conservative decoding settings, but more extensive experiments (for example, *pass@k*) could change effect sizes. Finally, passing unit tests does not guarantee semantic correctness: plausible-but-incorrect patches remain a concern. A more detailed analysis of specific fixes (and larger manual audits) would help clarify exactly where this process can assist LLMs.

Future work. Future work should explore improved summary designs and broader evaluation settings. Our experiments suggest both error-aware and concise summaries can help in different ways; combining these styles (for example, short but diagnostic summaries) or automatically tuning summary length and focus per task may hit a useful sweet spot. We also need to evaluate *summary-mediated repair* in more realistic settings, applying the technique to industrial function-level bugs and project-level errors.

More broadly, code summarisation as an intermediate has already shown promise for style normalisation [13] and developer-facing modification [28]. It would be worthwhile to investigate its usefulness for other related tasks such as code translation (for example, summarise Python, then generate Java) or code refactoring (for example, replacing a sqlite database implementation with sqlalchemy).

7 Conclusion

We introduce *summary-mediated repair*, a simple prompt-only pipeline that asks an LLM to produce a natural-language summary of buggy code and then regenerates the code conditioned on that summary, with the aim of identifying errors and repairing the original code. We evaluate the technique across eight LLMs on function-level coding tasks from HumanEvalPack and MBPP. *Summary-mediated repair* shows potential, being able to fix subtle implementation-level bugs, but gains are modest and LLM-dependent; only error-aware summaries outperformed directly prompting the LLMs to repair the code.

Although not state-of-the-art, *summary-mediated repair* highlights the value of code summaries as a lightweight and interpretable intermediate artefact for downstream coding tasks. We

release the full code and results on our GitHub repository[1], and we hope this work sparks follow-up studies that leverage code summarisation in downstream coding tasks and more broadly in natural language programming.

References

[1] Jacob Austin et al. 2021. Program Synthesis with Large Language Models. (Aug. 2021). arXiv: 2108.07732. doi:10.48550/arXiv.2108.07732.

[2] Yekun Chai et al. 2024. Tokenization Falling Short: On Subword Robustness in Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. Yaser Al-Onaizan et al., (Eds.) Association for Computational Linguistics, Miami, Florida, USA, (Nov. 2024), 1582–1599. doi:10.18653/v1/2024.findings-emnlp.86.

[3] Liguo Chen et al. 2024. A Survey on Evaluating Large Language Models in Code Generation Tasks. *Journal of computer science and technology*. doi:10.48550/ARXIV.2408.16498.

[4] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. (July 2021). arXiv: 2107.03374. doi:10.48550/arXiv.2107.03374.

[5] QiHong Chen et al. 2025. A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why? (Mar. 2025). arXiv: 2411.01414 [cs]. doi:10.48550/arXiv.2411.01414.

[6] Chenchen Gu et al. 2025. Auditing Prompt Caching in Language Model APIs. In *Forty-Second International Conference on Machine Learning*. (Feb. 2025). arXiv: 2502.07776 [cs].

[7] Ari Holtzman et al. 2019. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*. (Sept. 2019).

[8] Kai Huang et al. 2023. A Survey on Automated Program Repair Techniques. (May 2023). arXiv: 2303.18184 [cs]. doi:10.48550/arXiv.2303.18184.

[9] Binyuan Hui et al. 2024. Qwen2.5-Coder Technical Report. (Nov. 2024). arXiv: 2409.12186 [cs]. doi:10.48550/arXiv.2409.12186.

[10] Juyong Jiang et al. 2024. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology*, (June 2024). arXiv: 2406.00515.

[11] Carlos E. Jimenez et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *The Twelfth International Conference on Learning Representations*. arXiv. arXiv: 2310.06770 [cs]. doi:10.48550/arXiv.2310.06770.

[12] Claire Le Goues et al. 2019. Automated program repair. *Commun. ACM*, 62, 12, (Nov. 2019), 56–65. doi:10.1145/3318162.

[13] Haochen Li et al. 2024. Rewriting the Code: A Simple Method for Large Language Model Augmented Code Search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Lun-Wei Ku et al., (Eds.) Association for Computational Linguistics, Bangkok, Thailand, (Aug. 2024), 1371–1389. doi:10.18653/v1/2024.acl-long.75.

[14] Jiawei Liu et al. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (NIPS '23). Curran Associates Inc., Red Hook, NY, USA, (Dec. 2023), 21558–21572. Retrieved Oct. 20, 2025 from.

[15] Meta. 2025. Llama 3.3 | Model Cards and Prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/. (2025).

[16] Meta. 2025. Llama 4 | Model Cards and Prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/. (2025).

[17] Mistral AI. 2025. Codestral 25.01 | Mistral AI. https://mistral.ai/news/codestral-2501. (2025).

[18] Mistral AI. 2025. Medium is the new large. | Mistral AI. https://mistral.ai/news/mistral-medium-3. (2025).

[19] Norman Mu et al. 2025. A Closer Look at System Prompt Robustness. In *Neurips Safe Generative AI Workshop 2024*. (Feb. 2025). arXiv: 2502.12197 [cs].

[20] Niklas Muennighoff et al. 2024. OctoPack: Instruction Tuning Code Large Language Models. In *The Twelfth International Conference on Learning Representations*. (Feb. 2024). arXiv: 2308.07124 [cs]. doi:10.48550/arXiv.2308.07124.

[21] OpenAI. 2025. GPT-4.1 mini - API. https://platform.openai.com/docs/models/gpt-4.1-mini. (2025).

[22] OpenAI. 2025. GPT-4o mini - API. https://platform.openai.com/docs/models/gpt-4o-mini. (2025).

[23] Debalina Ghosh Paul et al. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE Computer Society, (July 2024), 87–94. ISBN: 979-8-3503-6505-4. doi:10.1109/AITest62860.2024.00019.

[24] Qwen et al. 2025. Qwen2.5 Technical Report. (Jan. 2025). arXiv: 2412.15115 [cs]. doi:10.48550/arXiv.2412.15115.

[25] Sean Stapleton et al. 2020. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension* (ICPC '20). Association for Computing Machinery, New York, NY, USA, (Sept. 2020), 2–13. ISBN: 978-1-4503-7958-8. doi:10.1145/3387904.3389258.

[26] Weisong Sun et al. 2025. Source Code Summarization in the Era of Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, (Sept. 2025), 1882–1894. ISBN: 979-8-3315-0569-1.

[27] Shin Hwei Tan et al. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2016). Association for Computing Machinery, New York, NY, USA, (Nov. 2016), 727–738. ISBN: 978-1-4503-4218-6. doi:10.1145/2950290.2950295.

[28] Ningzhi Tang et al. 2025. Exploring Direct Instruction and Summary-Mediated Prompting in LLM-Assisted Code Modification. (Aug. 2025). arXiv: 2508.01523 [cs]. doi:10.48550/arXiv.2508.01523.

[29] Boyang Yang et al. 2025. A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications. (June 2025). arXiv: 2506.23749 [cs]. doi:10.48550/arXiv.2506.23749.

[30] He Ye et al. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26, 2, (Feb. 2021), 20. doi:10.1007/s10664-020-09920-w.

[1] Project GitHub repository: *https://github.com/itsluketwist/summary-mediated-repair/*