World Journal of Information and Knowledge Management
Print ISSN: 2960-0111
Online ISSN: 2960-012X
DOI: https://doi.org/10.61784/wjikm4029

EXECUTION-FEEDBACK-GUIDED CONTEXT SELECTION FOR LARGE LANGUAGE MODEL-BASED SOFTWARE TESTING AND DEBUGGING

TianJun Mo[1*], GaiKe Wang[2], EnMiao Feng[1], PengFei Li[1]
[1]*Duke University, Durham, NC 27708, USA.*
[2]*New York University, New York, NY 10012, USA.*
[*]*Corresponding Author: TianJun Mo*

Abstract: Large language models (LLMs) are now routinely used to generate unit tests, localize faults and repair programs, yet their effectiveness on real projects hinges on a step that is rarely engineered explicitly: deciding which code enters the prompt. Whole files exceed context budgets, lexical retrieval misses the helper functions in which faults hide, and stack traces are silent for assertion failures. This paper presents EFCS, an execution-feedback-guided context selection approach that turns the by-products of test execution—test outcomes, exception traces, statement-level spectra and dynamic call edges—into a relevance model over function-level code units. EFCS fuses spectrum and traceback evidence, assembles a token-budgeted context from execution-trimmed function bodies and interface skeletons of runtime callees, and re-selects the context across debugging iterations through an evidence memory and a cost-aware feedback-escalation policy. We evaluate EFCS on 897 single-fault and 210 two-fault instances seeded into seven real-world Python projects with 4,182 developer-written tests, counting tokens with the cl100k tokenizer of GPT-3.5 and GPT-4. On held-out faults, EFCS places the faulty statement inside a 1K-token context for 94.6% of the faults, compared with 38.2% for BM25 retrieval, 58.7% for stack-trace selection and 9.6% for focal-file prompting; under a 2K budget it delivers interface-complete contexts for 95.3% of the faults while using 10.9% fewer tokens than spectrum-based selection. In a replay of iterative debugging, EFCS resolves 97.2% of two-fault instances within five model calls, lowers the share of wasted calls to 8.6%, and its escalation policy reduces traced test executions by 64.8% at a comparable resolution rate.
Keywords: Large language models; Automated debugging; Program repair; Fault localization; Context selection; Software testing

1 INTRODUCTION

Large language models (LLMs) trained on code have changed how software is tested and debugged [1]. Within a single year, LLM-based techniques were reported to repair real bugs through a conversation with the model [2], to debug their own programs from execution results [3], to outperform earlier learning-based repair tools [4-5], to fix programs generated by other models [6], to escape coverage plateaus in search-based test generation [7], to write unit tests from documentation and usage examples [8-9], to reproduce bugs from reports [10], and to localize faults by navigating a code base [11]. Recent surveys document the breadth of this movement in software engineering and in software testing in particular [12-13].

A common thread runs through these techniques: the model can only reason about the code that the prompt contains. Program-repair studies usually supply the buggy function directly, which presumes perfect fault localization—an assumption known to distort the comparison of repair tools [14]. Repository-level benchmarks expose the cost of dropping that assumption: resolving real GitHub issues requires locating the relevant code first [15], and code completion depends on retrieving the right cross-file context [16-17]. The context window is finite and must also hold instructions, failure reports and conversation history, and models exploit long inputs unevenly, using information at the beginning and the end of the context more reliably than information in the middle [18]. Selecting a small context that still contains the fault is therefore a first-order design problem for LLM-based debugging.

In practice, LLM debugging pipelines choose context in one of three ways: they paste the source file associated with the failing test, they retrieve code by lexical similarity to the failing test or issue text, as in the sparse-retrieval setting of SWE-bench that builds on BM25 [15, 19], or they include the functions that appear on the stack trace. Each choice has a structural blind spot. Library modules routinely span thousands of lines, so a file truncated to the budget seldom reaches the fault. Lexical similarity is high for the function that a test names but low for the helper in which the fault actually resides. Stack traces point at the crash site, but an assertion failure produces a trace that ends inside the test rather than inside the faulty code.

Our key observation is that a debugging loop already executes the tests. The resulting execution feedback—which tests fail, where exceptions originate, which statements each test executes, and which functions call which at run time—is precisely the evidence that spectrum-based fault localization has exploited for years [20]. Current LLM pipelines, however, forward this feedback to the model as text, for example as error messages and test results within a conversation or as verbal self-feedback [2-3, 21-22], instead of using it to decide which code the model should read.

© By the Author(s) 2023, under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0).

This paper presents EFCS (Execution-Feedback-guided Context Selection), an approach that places execution feedback at the center of context construction for LLM-based testing and debugging. EFCS (i) collects execution evidence with a lightweight test-runner tracer, (ii) fuses statement spectra, traceback proximity and a lexical prior into a relevance score over function-level code units, (iii) assembles a token-budgeted context from execution-trimmed function bodies and interface skeletons of the callees observed at run time, and (iv) re-selects the context whenever the validation of a model proposal yields new feedback, using an evidence memory that rotates unproductive code out of the prompt and an escalation policy that collects expensive feedback only when cheap feedback proves insufficient.

We evaluate EFCS on seven real-world Python projects with 4,182 developer-written tests, into which we seed 897 single-statement faults and 210 two-fault combinations. To isolate context selection from the generative variance of a particular model and from training-data contamination [23], the evaluation measures context quality directly and replays iterative debugging under an explicit repair model. The contributions of this paper are as follows:

An execution-feedback-guided formulation of context selection for LLM-based testing and debugging, and the EFCS approach that combines evidence fusion, budget-aware assembly and feedback-driven re-selection.

Two context-compression operators—execution trimming and runtime interface skeletons—that keep the faulty statement in 99.9% of the cases while removing 27.4% of the tokens of faulty functions on average, and that complete the callee interfaces of the selected code.

A model-agnostic evaluation protocol, consisting of context-level metrics and a replay of iterative debugging, instantiated on 1,107 fault instances in seven projects.

Empirical evidence that EFCS dominates file-, retrieval- and trace-based context selection, improves on spectrum-based selection at tight budgets and in interface completeness, and resolves more multi-fault instances with fewer wasted model calls.

The remainder of the paper is organized as follows. Section 2 reviews related work, Section 3 describes EFCS, Sections 4 and 5 present the experimental setup and the results, Section 6 discusses implications, and Section 7 concludes.

2 BACKGROUND AND RELATED WORK

2.1 LLMs for Testing and Debugging

Codex demonstrated that LLMs trained on public code synthesize functionally correct programs from docstrings [1]. For program repair, Xia et al. showed that pre-trained LLMs outperform dedicated repair tools when prompted with the buggy code [4], Jiang et al. found that fine-tuning substantially improves code language models for repair [5], and Fan et al. studied the repair of incorrect programs produced by LLMs [6]. ChatRepair interleaves patch generation with test feedback in a conversation [2], and Self-Debug teaches models to explain and correct their code using execution results [3]. General feedback loops such as Self-Refine and Reflexion iterate on verbal feedback [21-22], and Olausson et al. found that the gains of self-repair are bounded by the quality of the feedback the model receives [24]. For testing, CodaMosa invokes an LLM when search-based test generation stalls [7], TestPilot generates unit tests from signatures, documentation and usage snippets [8], ChatTester refines generated tests using compilation and execution errors [9], and LIBRO reproduces bugs from reports [10]. AutoFL lets an LLM navigate a repository through function-call tools to localize and explain faults [11]. These techniques depend on the code placed in the prompt, yet they either assume a given focal function or delegate exploration to the model itself.

2.2 Context Retrieval for Code Models

Repository context is a recognized bottleneck for code models. RepoCoder alternates retrieval and generation so that a draft completion guides the retrieval of further repository code [16], and CrossCodeEval shows that code completion improves substantially when relevant cross-file context is provided [17]. SWE-bench pairs real GitHub issues with their repositories and evaluates models with code retrieved by BM25 [19] using the issue text as the query, as well as with an oracle that supplies the files edited by the reference fix [15]. Liu et al. show that models make uneven use of long inputs [18], which further rewards compact and well-ordered contexts. These approaches retrieve by textual or semantic similarity. EFCS retrieves by execution evidence, which is available precisely in the testing and debugging scenarios where a test has failed.

2.3 Spectrum-Based Fault Localization and Fault Benchmarks

Spectrum-based fault localization (SBFL) ranks program elements by correlating their execution with test outcomes; the Ochiai coefficient is among the most accurate similarity measures for this purpose [20]. Liu et al. demonstrated that the localization step strongly affects which bugs repair systems fix and how the systems compare [14]. Real-fault benchmarks such as Defects4J [25] have driven this research, and mutation-based artificial faults are a valid substitute for real faults in testing experiments [26]. For LLM-based techniques, faults from public benchmarks may already appear in pre-training corpora, which motivates evaluation data that cannot have been memorized [23]. EFCS uses SBFL as one of its evidence sources, but its output differs in kind: instead of a ranked list for a developer, it produces a token-budgeted, compressed and interface-complete context for a model, and it updates that context as the debugging loop produces new feedback.

3 THE EFCS APPROACH

3.1 Overview and Problem Formulation

Figure 1 summarizes EFCS. Let P be a program decomposed into code units U, namely module-level functions and class methods, where nested functions are folded into their enclosing unit. Let T be the test suite and $F \subseteq T$ the failing tests. A context C is an ordered sequence of unit views, where a view is the full body of a unit, its execution-trimmed body, or its skeleton (signature and first docstring line). Each view is preceded by a one-line header with the file path and the line span, and the cost $\tau(C)$ is the number of tokens of all views and headers. Given a budget B, context selection chooses C with $\tau(C) \leq B$ such that C contains the statements responsible for the failure and the interfaces needed to change them. In the iterative setting, each model proposal is validated by re-executing the tests, and the resulting feedback determines the next context.

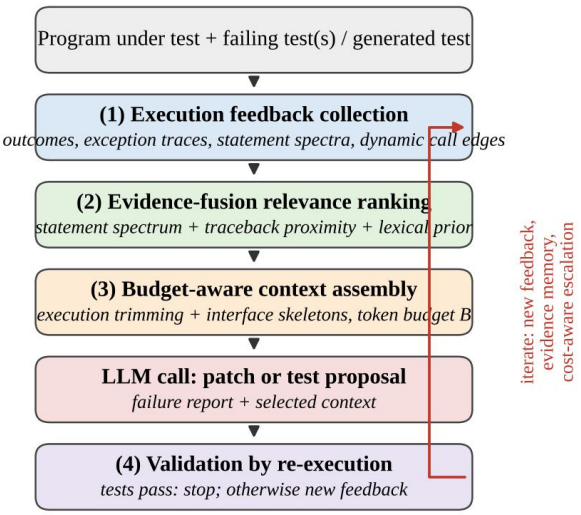


Figure 1 Overview of EFCS
Execution feedback drives ranking and assembly; every validation run produces new feedback for the next model call.

3.2 Execution Feedback Collection

EFCS instruments the test runner with a tracer that is restricted to project source files. For every test t, it records the outcome, the set of executed statements, and the dynamic call edges between call sites and callee units. For every failing test, it additionally records the exception type, the message and the traceback frames; assertion messages produced by the test framework carry the expected and the actual values and are kept verbatim for the prompt. The tracer supports two modes: a cheap mode that traces only the failing tests, and a full mode that also traces the passing tests and thereby provides complete spectra. One traced run is needed per program version.

3.3 Evidence-Fusion Ranking

Statement spectra. For a statement s, let $e_f(s)$ and $e_p(s)$ denote the numbers of failing and passing tests that execute s. EFCS uses the Ochiai coefficient [20].

$$\mathrm{Och}(s) = e_f(s) / \sqrt{(|F| \cdot (e_f(s) + e_p(s)))} \quad (1)$$

and lifts it to units by maximum aggregation, $S(u) = \max_{s \in u} \mathrm{Och}(s)$. Only units executed by at least one failing test can contain a statement that triggers the failure, so the candidate set is $U_F = \{u \in U : \exists\ s \in u, e_f(s) > 0\}$. On our subjects, U_F contains 43.7 units on average, i.e., 17.7% of all units, before any ranking takes place.

Traceback proximity. For a failing test t, let $(u_1, \ldots, u_k)$ be the distinct project units on its traceback, ordered from the innermost frame outwards, and let $r(t, u)$ be the position of u in this sequence. EFCS computes

$$T(u) = (1/|F|) \cdot \Sigma_{t \in F}\ \gamma^{r(t,u)-1} \quad (2)$$

where terms are zero for units that do not appear on the traceback of t and $\gamma = 0.5$. Crash failures thus emphasize the raising function and its immediate callers, whereas assertion failures, whose traces end in test code, contribute nothing and leave the decision to the spectrum.

Lexical prior. Let q_F be the identifiers in the source code of the failing tests and in their exception messages, split at case and underscore boundaries. The prior $L(u)$ is the BM25 score of the text of unit u for the query q_F, normalized by its maximum over U_F [19].

Fusion. The relevance of a candidate unit is

$$R(u) = S(u) + \alpha \cdot T(u) + \delta \cdot L(u), \qquad u \in U_F \quad (3)$$

with $\alpha = 0.1$ and $\delta = 0.05$ selected on a development split (Section 4). The spectrum dominates the score; the remaining terms reorder units whose spectra are equal or close, a frequent situation because callers and the helpers they invoke tend to be executed by the same tests. In cheap mode, $S(u)$ is replaced by the fraction of failing tests that execute u.

3.4 Budget-Aware Context Assembly

Execution trimming. The trimmed view of a unit keeps its signature, every statement executed by at least one failing test, and the headers of the compound statements that enclose them (if/elif/else, loops, try/except/finally and with). Each maximal run of unexecuted statements collapses into a single "..." line, and docstrings are dropped. Because a fault has to be executed to cause a failure, the statement that triggers the failure survives trimming. On our subjects, the faulty statement is retained in 99.9% of the cases, while the faulty unit shrinks by 27.4% of its tokens on average.
Interface skeletons. A model that edits a function needs to know the contracts of the functions it calls. EFCS reads the callees of the highest-ranked units from the dynamic call edges of the failing runs and, for callees that are not included in full, inserts a skeleton consisting of the signature and the first docstring line. Skeletons cost a few tokens each, and they expose exactly the helpers that the failing executions used rather than all statically possible callees.
Packing. EFCS fills a fraction $(1 - \sigma)B$ of the budget with trimmed views in descending order of relevance, skipping views that do not fit (first fit), adds skeletons for the callees of the three highest-ranked units, and finally uses the remaining budget for further trimmed views ($\sigma = 0.15$). Views are emitted in relevance order, so that the most suspicious code appears at the beginning of the prompt, where models use context most reliably [18]. Algorithm 1 shows the complete debugging loop.

Algorithm 1 EFCS Debugging Loop

Input: program P, tests T, budget B, call limit K, model M
$E \leftarrow$ Collect(P, cheap); $m \leftarrow 0$; *full* $\leftarrow$ false
for $k = 1$ **to** K **do**
$R'(u) \leftarrow S(u) + \alpha T(u) + \delta L(u) - \mu \cdot m(u), \quad u \in U_F$
$C \leftarrow$ Pack(U_F ordered by R', B) // trimming, skeletons
$\pi \leftarrow M$(FailureReport(E), C); $F' \leftarrow$ Fail(apply(P, π))
if $F' = \varnothing$ **then return** apply(P, π)
if $F' \subset$ Fail(P) **then** // progress: keep the patch
$P \leftarrow$ apply(P, π); $E \leftarrow$ Collect(P, cheap); *full* $\leftarrow$ false
else $m(u) \leftarrow m(u) + 1$ for all u in C
if not *full* **then** $E \leftarrow$ Collect(P, full); *full* $\leftarrow$ true
end for

3.5 Feedback-Driven Iterative Re-Selection

Re-selection on new feedback. Each model proposal is validated by re-running the tests. A proposal that passes all tests ends the loop. A proposal that removes some failures without introducing new ones is kept, which changes the program and therefore its execution evidence: faults that were masked or ranked low under the old failure set now surface, and EFCS rebuilds the context from the new feedback.
Evidence memory. A call that makes no progress indicates that its context did not enable a fix. EFCS records for each unit the number $m(u)$ of unproductive calls whose context contained it and ranks by

$$R'(u) = R(u) - \mu \cdot m(u) \tag{4}$$

The penalty rotates unproductive code out of the context so that the next-ranked candidates become visible. Because it is additive on the evidence scale, a unit whose evidence clearly dominates stays in the context after an unsuccessful call—the model may simply have produced an incorrect patch—whereas near-ties are resolved in favor of code that the model has not yet seen. The penalty $\mu = 0.05$ was selected on the development split.
Cost-aware escalation. For each new program version, EFCS first builds the context from cheap feedback, i.e., from the failing tests only. If the resulting call makes no progress, it escalates to full feedback by tracing the passing tests once, so the cost of complete spectra is paid only when cheap evidence proves insufficient.
Prompt construction and test generation. The prompt combines a failure report—the source of the failing tests and the exception types and messages, including the expected and actual values of failed assertions—with the selected context. The same loop serves LLM-based test generation: when a generated test fails to import, raises an attribute or type error, or fails an assertion, the generated test takes the role of the failing test, and its traceback and the callees it exercised select the definitions and interface skeletons that the model needs to correct the test.

4 EXPERIMENTAL SETUP

4.1 Research Questions

RQ1 (Effectiveness): How accurately and compactly does EFCS select fault-relevant context compared with the context-selection strategies used in LLM debugging pipelines?
RQ2 (Ablation): How much do traceback evidence, the lexical prior, execution trimming, interface skeletons and full spectra contribute?

RQ3 (Iterative debugging): Does feedback-driven re-selection increase the number of faults resolved within a fixed number of model calls, and at what cost in tokens and test executions?

4.2 Subject Projects

We selected seven open-source Python libraries with comprehensive developer test suites and without native extensions (Table 1): cachetools (caching), toolz (functional utilities), more-itertools (iterator recipes), sqlparse (SQL parsing), click (command-line interfaces), marshmallow (object serialization) and Python-Markdown (text rendering). We use the releases listed in Table 1, all published in 2023 or earlier. The subjects comprise 89 source files, 26.8K non-comment source lines, 1,716 function-level units and 4,182 developer tests that pass on the original versions under Python 3.12 and pytest 7.4.

Table 1 Subject Projects and Fault Instances

Project	Version	Files	SLOC	Units	Tests	1-fault	2-fault
cachetools	5.3.1	3	797	101	214	121	30
toolz	0.12.0	12	3,302	169	179	168	30
more-itertools	10.1.0	2	4,161	206	598	162	30
sqlparse	0.4.4	16	1,957	164	427	102	30
click	8.1.7	15	7,611	477	589	111	30
marshmallow	3.20.1	10	3,918	255	1,218	119	30
Python-Markdown	3.4.4	31	5,050	344	957	114	30
Total	–	**89**	**26,796**	**1,716**	**4,182**	**897**	**210**

4.3 Fault Instances

We seeded single-statement faults with eight operators that mirror common bug patterns: relational operator replacement (ROR), arithmetic operator replacement (AOR), logical connector replacement (LCR), removal of a negation (UOD), negation of a branch condition (NEG), constant perturbation (CRP), swap of call arguments (ASW) and statement deletion (SDL). Candidates were sampled across the units of each project (at most two per unit and balanced over operators), and a candidate was kept if it made between one and thirty developer tests fail. Mutants of this kind are coupled to the majority of real faults and are an accepted substitute for them in testing research [26]. In addition, seeded faults create program versions that do not exist in public repositories, which avoids the data-leakage concern raised for public bug benchmarks [23].

The data set contains 897 single-fault instances (SDL 201, ASW 158, NEG 143, ROR 121, CRP 116, AOR 69, LCR 57, UOD 32). Of these, 385 manifest as crashes, i.e., as exceptions raised inside project code, and 512 as assertion failures; a fault makes 6.5 tests fail on average (median 3). For RQ3, we combined two faults located in different units of the same project and re-executed the test suite on the combined version, which yields 210 two-fault instances. We split all instances by a hash of the fault identity: one third forms the development split used for tuning (284 single-fault and 69 two-fault instances), and all reported results are computed on the remaining held-out split (613 single-fault and 141 two-fault instances).

4.4 Baselines

We compare EFCS with four context-selection strategies that reflect current practice and the strongest localization-based alternative. *FocalFile* (FF) pastes the source module matched to the failing test by naming convention (test_x.py → x.py), or the module with the highest BM25 score if no name matches, truncated to the budget. *BM25* retrieves units by lexical similarity, using the failing test code and the exception messages as the query, following the sparse-retrieval setting of SWE-bench [15, 19]. *StackTrace* (ST) includes the project units on the failing tracebacks from the innermost frame outwards and fills the remaining budget by BM25. *SBFL* ranks the units executed by failing tests by maximum statement-level Ochiai [20] and includes their full bodies; it uses the same full-mode feedback as EFCS. All strategies share the same packing procedure, headers and tokenizer.

4.5 Metrics and Replay Protocol

For RQ1 and RQ2, we report Top-k, the share of faults whose faulty unit is ranked among the first k units, and the mean reciprocal rank (MRR). Hit@B is the share of faults for which the context under budget B contains the faulty statement, either in a full body or in a trimmed body that retains it. Suff@B additionally requires every callee that the faulty unit invoked during the failing runs to be present in full or as a skeleton, i.e., it measures whether the context is interface-complete. Tok@B is the number of tokens actually used. Budgets range from 256 to 4,096 tokens and are counted with cl100k_base, the tokenizer of GPT-3.5 and GPT-4.

For RQ3, we replay debugging episodes under an abstract repair model. A model call succeeds with probability p if its context contains a remaining faulty statement, in which case the highest-ranked such fault is fixed; unsuccessful proposals are discarded, leaving the program unchanged. We use $p = 1.0$, a model that always fixes a visible fault, and $p = 0.5$, a model that fails on half of its attempts, with ten random seeds and common random numbers across strategies.

Each episode allows $K = 5$ calls with a budget of 1K tokens for two-fault and 512 tokens for single-fault instances. We report the resolution rate, the mean number of calls, the share of wasted calls—calls whose context contains none of the remaining faults and that therefore cannot succeed with any model—the cumulative context tokens per episode and the number of traced test executions. The replay isolates the contribution of context selection: whatever the generative ability of a concrete LLM, its success is bounded by whether the faulty code is visible to it.
We compare static selection, which computes one context per episode (Static), re-selection on new feedback (Iter), re-selection with the evidence memory (Iter+Mem) and EFCS with cost-aware escalation (Esc). For a fair comparison, the memory mechanism is also applied to SBFL.

4.6 Implementation

EFCS is implemented in Python. The tracer is a pytest plugin built on the interpreter's trace hook and restricted to project files, and code units are extracted from the abstract syntax tree. The fusion weights were chosen by grid search for MRR on the development split, where fusion raised the MRR from 0.779 (spectrum only) to 0.816; μ was chosen from $\{0.05, 0.1, 0.2, 0.3, 0.5, 1.0\}$ by the mean resolution rate of the development episodes, and $\gamma = 0.5$ and $\sigma = 0.15$ were fixed beforehand. All experiments ran on a single CPU core.

5 RESULTS

5.1 RQ1: Effectiveness of Context Selection

Table 2 and Figure 2 report the results on the 613 held-out single-fault instances. The strategies used in LLM pipelines today fall far behind under realistic budgets. FocalFile reaches the faulty statement for 9.6% of the faults at 1K tokens and for only 30.8% even at 4K tokens, because library modules of several thousand lines cannot be pasted within a budget. BM25 retrieval reaches 38.2% at 1K tokens: lexical similarity locates the function that a test calls by name, but not the helper in which the fault resides. StackTrace reaches 58.7% at 1K tokens.

Table 2 Context Selection on Held-Out Single Faults (n = 613); Top-k, Hit@B and Suff@B in %, Tok = Tokens Used at B = 2K

Method	Top-1	Top-3	MRR	H@256	H@512	H@1K	H@2K	S@2K	Tok
FF	1.6	7.2	.068	0.8	3.6	9.6	18.6	13.1	1841
BM25	16.3	28.7	.259	20.1	30.8	38.2	43.9	29.4	2039
ST	37.5	50.4	.465	40.9	52.4	58.7	62.8	42.3	2039
SBFL	68.0	87.1	.785	73.4	88.3	93.8	96.1	82.1	1409
EFCS	**70.0**	**87.6**	**.797**	**81.9**	**89.6**	**94.6**	**97.1**	**95.3**	**1255**

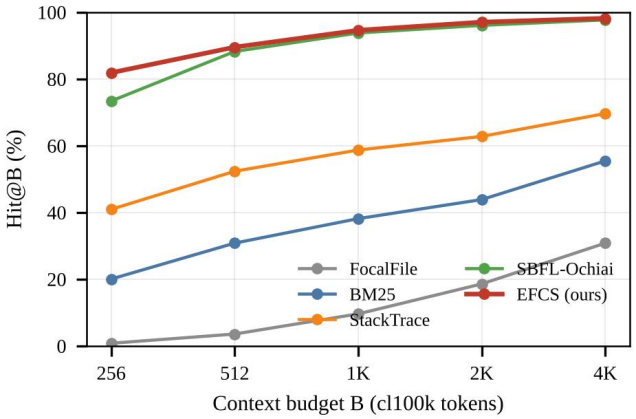


Figure 2 Share of Faults Whose Faulty Statement Is Contained in the Selected Context (Hit@B) as a Function of the Token Budget B on the Held-Out Split

Figure 3(a) explains the gap of StackTrace. For crashes it attains 82.9% at 1K tokens, but for assertion failures, whose tracebacks end in test code, it falls to 40.6%, whereas EFCS reaches 98.5% and 91.7%, respectively. Traceback evidence also sharpens the ranking of EFCS for crashes, raising Top-1 from 76.8% (SBFL) to 81.7%.
SBFL is the strongest baseline, which confirms that execution evidence rather than textual similarity is the decisive signal. EFCS improves on it where the budget is tight and where completeness matters. At 256 tokens, EFCS contains the faulty statement for 81.9% of the faults versus 73.4% for SBFL (+8.5 points); at 2K tokens, 95.3% of its contexts are interface-complete versus 82.1% (+13.2 points), while it consumes 1,255 instead of 1,409 tokens (−10.9%). EFCS also ranks the faulty unit first more often (70.0% vs. 68.0%) and attains a higher MRR (0.797 vs. 0.785). Per project (Figure 3(b)), EFCS matches or exceeds SBFL at 512 tokens in five of the seven projects, and its interface completeness at 2K tokens is higher or equal in all seven, with the largest gains on sqlparse (+31.8 points), Python-Markdown (+24.7) and marshmallow (+21.6).

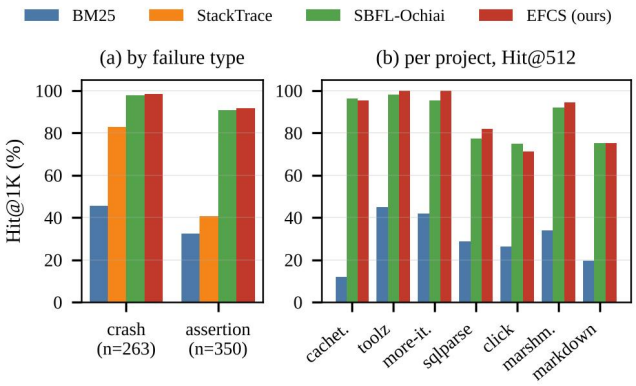


Figure 3 (a) Hit@1K by Failure Type on the Held-Out Split; (b) Hit@512 per Project for BM25, SBFL and EFCS

5.2 RQ2: Contribution of the Components

Table 3 isolates the components of EFCS. *Execution trimming* is the main source of the gains at tight budgets: without it, Hit@256 drops from 81.9% to 69.7% and the context grows from 1,255 to 1,400 tokens. *Interface skeletons* raise Suff@2K from 85.3% to 95.3%; the budget they occupy costs 3.3 points of Hit@256, a favorable trade because a context without the interfaces of the called helpers leaves the model to guess their behavior. *Traceback evidence* raises Top-1 from 68.8% to 70.0% and MRR from 0.789 to 0.797, with the effect concentrated on crashes. Removing the *lexical prior* changes every metric by at most 1.1 points, which indicates that the improvements stem from execution evidence rather than from textual similarity.

Table 3 Ablation on Held-Out Single Faults (%, Tok at B = 2K)

Variant	Top-1	MRR	H@256	H@512	H@1K	S@2K	Tok
EFCS (full)	**70.0**	**.797**	**81.9**	**89.6**	**94.6**	**95.3**	**1255**
Ranking only	70.0	.797	73.6	88.3	93.5	82.5	1409
w/o traceback	68.8	.789	81.1	88.9	94.6	95.1	1255
w/o lexical prior	70.6	.800	82.9	90.7	94.8	95.4	1254
w/o trimming	70.0	.797	69.7	87.4	93.3	94.3	1400
w/o skeletons	70.0	.797	85.2	91.5	94.6	85.3	1255
Failing-only feedback	48.1	.613	68.5	76.0	79.3	84.5	1256

Using the EFCS ranking with full bodies and without skeletons (ranking only) yields 73.6% at 256 tokens, close to SBFL (73.4%): the fused ranking contributes the higher Top-1, whereas the assembly operators convert the budget into coverage of the fault and its interfaces. The cheap failing-only variant, which traces only the failing tests, still places the faulty statement in a 1K context for 79.3% of the faults—well above StackTrace (58.7%) and BM25 (38.2%)—and therefore provides a strong first context for the escalation policy.

5.3 RQ3: Iterative Debugging

Table 4 and Figure 4 report the replay on the 141 held-out two-fault instances and the 613 held-out single faults. Static selection caps the achievable resolution: with a single context per episode, SBFL resolves 60.3% and EFCS 67.4% of the two-fault instances even with an ideal model (p = 1.0), because the second fault frequently lies outside the context selected for the initial failure set, and 44.8–50.9% of their calls are wasted. Re-selecting the context on new feedback raises the resolution to 90.8% (SBFL) and 92.2% (EFCS), and the evidence memory raises it further to 95.7% and 97.2%. The complete EFCS loop needs 2.14 calls per episode, wastes 8.6% of its calls and consumes 1,767 context tokens per episode, compared with 12.2% and 1,956 tokens for SBFL with the same memory, and with 88.3% and 4,612 tokens for iterative BM25 retrieval, which resolves only 14.9% of the instances.

Table 4 Iterative Debugging Replay (Held-Out). Res, Calls, Waste (%), Tok and Traced per Episode for Two-Fault Instances with p = 1.0; Res.5: Two-Fault, p = 0.5; Res1F: Single Fault, B = 512

Strategy	Res	Calls	Waste	Tok	Traced	Res.5	Res1F
BM25-Static	2.8	4.91	91.6	4980	0	2.2	30.8
BM25-Iter	14.9	4.55	88.3	4612	0	11.9	30.8
SBFL-Static	60.3	3.19	50.9	3085	584	49.1	88.3
SBFL-Iter	90.8	2.28	17.8	2027	1145	73.2	88.3
SBFL-Iter+Mem	95.7	2.21	12.2	1956	1156	73.3	94.0
EFCS-Static	67.4	2.98	44.8	2764	584	55.0	89.6
EFCS-Iter	92.2	2.23	15.2	1862	1149	74.0	89.6
EFCS-Iter+Mem	**97.2**	**2.14**	**8.6**	**1767**	**1162**	**75.5**	**95.4**
EFCS-Esc	96.5	2.64	26.1	2273	409	68.9	95.6

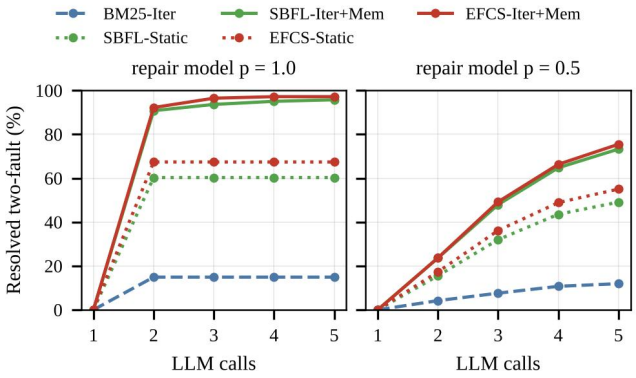


Figure 4 Cumulative Share of Resolved Two-Fault Instances over Model Calls (Held-Out Split, B = 1K) under the Ideal (p = 1.0) and the Imperfect (p = 0.5) Repair Model

With an imperfect model ($p = 0.5$), EFCS with memory resolves 75.5% of the two-fault instances, ahead of SBFL with memory (73.3%), EFCS without memory (74.0%) and static EFCS (55.0%), and it maintains the lowest share of wasted calls (6.5%). The additive penalty keeps a dominant faulty unit in the context after an unsuccessful attempt, so the memory remains beneficial when the model fails on visible faults. On single faults under a 512-token budget, the memory lifts EFCS from 89.6% to 95.4% (SBFL: from 88.3% to 94.0%).

Cost of feedback. Escalation reduces the number of traced test executions per two-fault episode from 1,162 to 409 (−64.8%) while resolving 96.5% of the instances with $p = 1.0$, at the price of 0.5 additional calls on average; on single faults it resolves 95.6% with 172 instead of 589 traced executions (−70.8%). With $p = 0.5$, escalation resolves 68.9% with 34.4% fewer traced executions. Escalation is therefore preferable when test execution dominates the cost of a debugging step, whereas collecting full spectra upfront maximizes the resolution when model calls dominate. Table 5 lists the wall-clock time of a fully traced run of each test suite: between 0.6 s and 18.2 s on a single core, a one-off cost per program version.

Table 5 Wall-Clock Time of One Complete Test-Suite Run (s)

Project	Untraced	Traced (full mode)
cachetools	0.27	1.28
toolz	0.27	0.64
more-itertools	1.18	3.52
sqlparse	0.51	6.46
click	0.67	7.17
marshmallow	0.96	6.60
Python-Markdown	1.29	18.22

6 DISCUSSION

Execution feedback as a retrieval signal. The results show that, in testing and debugging, the information that determines what a model should read is produced by running the tests. Textual retrieval and file-level prompting answer a different question—which code resembles the failing test—and leave most faults outside realistic budgets. Execution evidence narrows the search space to 17.7% of the units before any ranking, and its fusion with traceback information yields contexts that are both small and complete.

Budget as a design parameter. The gains of execution trimming and interface skeletons are largest at small budgets. Small code contexts matter beyond models with short windows: in conversational repair and in iterative test refinement [2, 9], every round adds failure reports and previous proposals to the prompt, so the share of the window available for code shrinks as the conversation proceeds.

Integration with LLM loops. EFCS is model-agnostic and acts as a context provider for existing loops. Conversational repair, self-debugging and reflective agents can keep their prompting strategies while delegating the choice of code to EFCS [3, 22], whose evidence memory complements their verbal feedback with a quantitative record of what has already been shown to the model. Whenever fixing one fault exposes another, feedback-driven re-selection is required to keep the next fault visible, as the gap between static and iterative selection in RQ3 demonstrates.

Beyond Python. EFCS requires statement coverage, stack traces and caller–callee information, which standard instrumentation provides for mainstream languages; the ranking, assembly and re-selection components are language-independent.

7 CONCLUSION

This paper introduced EFCS, an approach that selects the code context of LLM-based testing and debugging from execution feedback. EFCS fuses statement spectra, traceback proximity and a lexical prior, assembles token-budgeted

contexts from execution-trimmed bodies and runtime interface skeletons, and re-selects the context across iterations with an evidence memory and cost-aware feedback escalation. On 1,107 fault instances in seven real-world Python projects, EFCS contained the faulty statement in a 1K-token context for 94.6% of the faults, delivered interface-complete contexts for 95.3% of the faults under 2K tokens with 10.9% fewer tokens than spectrum-based selection, and resolved 97.2% of the two-fault instances within five model calls in the iterative-debugging replay. Future work will couple EFCS with open code LLMs in end-to-end repair and test-generation loops and extend the feedback model with data-flow evidence.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. arXiv:2107.03374, 2021.
[2] Xia C S, Zhang L. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385, 2023.
[3] Chen X, Lin M, Schärli N, et al. Teaching large language models to self-debug. arXiv:2304.05128, 2023.
[4] Xia C S, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023. DOI: 10.1109/ICSE48619.2023.00129.
[5] Jiang N, Liu K, Lutellier T, et al. Impact of code language models on automated program repair. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023. DOI: 10.1109/ICSE48619.2023.00125.
[6] Fan Z, Gao X, Mirchev M, et al. Automated repair of programs from large language models. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023. DOI: 10.1109/ICSE48619.2023.00128.
[7] Lemieux C, Inala J P, Lahiri S K, et al. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023.
[8] Schäfer M, Nadi S, Eghbali A, et al. An empirical evaluation of using large language models for automated unit test generation. arXiv:2302.06527, 2023.
[9] Yuan Z, Lou Y, Liu M, et al. No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv:2305.04207, 2023.
[10] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023.
[11] Kang S, An G, Yoo S. A preliminary evaluation of LLM-based fault localization. arXiv:2308.05487, 2023.
[12] Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: A systematic literature review. arXiv:2308.10620, 2023.
[13] Wang J, Huang Y, Chen C, et al. Software testing with large language model: Survey, landscape, and vision. arXiv:2307.07221, 2023.
[14] Liu K, Koyuncu A, Bissyandé T F, et al. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2019: 102-113.
[15] Jimenez C E, Yang J, Wettig A, et al. SWE-bench: Can language models resolve real-world GitHub issues? arXiv:2310.06770, 2023.
[16] Zhang F, Chen B, Zhang Y, et al. RepoCoder: Repository-level code completion through iterative retrieval and generation. arXiv:2303.12570, 2023.
[17] Ding Y, Wang Z, Ahmad W U, et al. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In: Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track, 2023.
[18] Liu N F, Lin K, Hewitt J, et al. Lost in the middle: How language models use long contexts. arXiv:2307.03172, 2023.
[19] Robertson S, Zaragoza H. The probabilistic relevance framework: BM25 and beyond. Foundations and Trends in Information Retrieval, 2009, 3(4): 333-389.
[20] Abreu R, Zoeteweij P, van Gemund A J C. On the accuracy of spectrum-based fault localization. In: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION), 2007: 89-98.
[21] Madaan A, Tandon N, Gupta P, et al. Self-Refine: Iterative refinement with self-feedback. arXiv:2303.17651, 2023.
[22] Shinn N, Cassano F, Berman E, et al. Reflexion: Language agents with verbal reinforcement learning. arXiv:2303.11366, 2023.

[23] Wu Y, Li Z, Zhang J M, et al. ConDefects: A new dataset to address the data leakage concern for LLM-based fault localization and program repair. arXiv:2310.16253, 2023.
[24] Olausson T X, Inala J P, Wang C, et al. Is self-repair a silver bullet for code generation? arXiv:2306.09896, 2023.
[25] Just R, Jalali D, Ernst M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014: 437-440.
[26] Just R, Jalali D, Inozemtseva L, et al. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014: 654-665.