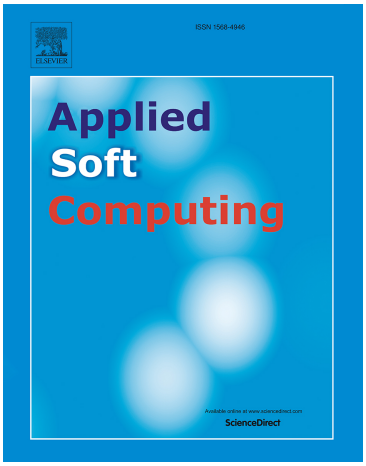

FedDualFix: Hierarchical federated learning for adaptive multi-agent program repair via dual-modality modeling

Xiaolin Ju, Qingyun Liu, Chang Li, Haochen Wang, Heling Cao, Xiang Chen

PII: S1568-4946(26)01346-3
DOI: https://doi.org/10.1016/j.asoc.2026.115898
Reference: ASOC 115898

To appear in: *Applied Soft Computing*

Received Date: 25 February 2026
Revised Date: 17 June 2026
Accepted Date: 30 June 2026

Please cite this article as: Ju X, Liu Q, Li C, Wang H, Cao H, Chen X, FedDualFix: Hierarchical federated learning for adaptive multi-agent program repair via dual-modality modeling, *Applied Soft Computing* (2026), doi: https://doi.org/10.1016/j.asoc.2026.115898.

Highlights:

- Proposes FedDualFix, a federated, hierarchical multi-agent framework for adaptive automated program repair under data-locality constraints..
- Introduces confidence-guided scheduling with early exits and layer escalation to reduce unnecessary computation.
- Models dual-modality coherence by aligning code edits with natural-language artifacts to improve semantic plausibility and ranking.
- On Defects4J, achieves 58 correct repairs and 52.1% Top-1 precision, outperforming Recoder and CodeT5.

FedDualFix: Hierarchical Federated Learning for Adaptive Multi-Agent Program Repair via Dual-Modality Modeling

Xiaolin Ju[a,b,*], Qingyun Liu[a], Chang Li[a,*], Haochen Wang[a], Heling Cao[c,*], Xiang Chen[a,b]

[a]*School of Artificial Intelligence and Computer Science, Nantong University, Nantong, 226019, Jiangsu, China*
[b]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[c]*College of Information Science and Engineering, Henan University of Technology, Zhengzhou, 450001, Henan, China*

Abstract

Recent advances in large language models (LLMs) have enhanced automated program repair (APR) by interpreting failures, generating context-aware patches, and iteratively refining fixes, thereby improving repair accuracy and coverage. However, existing methods still face three key limitations: they often rely on centralized training on sensitive codebases; their pipelines are rigid, one-pass processes that cannot adapt to the complexity of a defect; and they do not fully exploit auxiliary semantic signals, such as bug reports or documentation. To address these challenges, we propose FedDualFix, a federated multi-agent repair framework that improves repair accuracy, efficiency, and semantic coherence in heterogeneous and data-locality-constrained environments. FedDualFix decomposes the repair process into specialized agents for (i) context modeling, (ii) semantic slicing, (iii) fault localization, (iv) patch generation, and (v) cross-modal alignment. A hierarchical controller with confidence-guided scheduling enables early exits for simple defects and triggers deeper, structure-aware reasoning for complex or

*Corresponding authors.
Email addresses: ju.xl@ntu.edu.cn (Xiaolin Ju), liuqyai@outlook.com (Qingyun Liu), lichang@ntu.edu.cn (Chang Li), haochenwang@ntu.edu.cn (Haochen Wang), caohl@haut.edu.cn (Heling Cao), xchencs@ntu.edu.cn (Xiang Chen)

ambiguous cases. Dual-modality alignment further associates code-level edits with natural-language descriptions to enhance semantic plausibility. Experiments on Defects4J using several LLMs, including GPT-4o, show that FedDualFix produces 58 correct repairs and achieves 52.1% Top-1 precision, outperforming representative APR baselines and a same-backend GPT-4o Direct baseline. The adaptive scheduling mechanism reduces inference time by more than 25%, and ablation studies demonstrate the contribution of hierarchical control, multi-agent design, and dual-modality integration. These results highlight the potential of combining federated coordination with adaptive multi-agent reasoning to support accurate and scalable APR in real-world software maintenance settings.

Keywords: Automated Program Repair, Federated Learning, Hierarchical Multi-Agent Repair, Large Language Models, Dual-Modality Modeling.

1. Introduction

Software defects [1, 2] remain a persistent obstacle to the reliability and maintainability of modern software systems. Automated Program Repair (APR) [3, 4, 5, 6] has emerged as a promising approach to reducing the cost of manual debugging. With the rapid progress of Large Language Models (LLMs) [7, 8, 9], recent APR systems increasingly rely on LLMs to perform fault localization, patch synthesis, and consistency checking. Although LLM-based repair [10, 11, 12, 13, 14] has demonstrated impressive capability, existing approaches continue to exhibit several structural limitations. **Centralized training.** Most approaches train a single model on a monolithic, often proprietary, code base, which conflicts with privacy and organizational constraints. **Rigid execution.** Repair components are commonly executed in a fixed, one-pass order, regardless of defect complexity or intermediate confidence signals. **Weak semantic grounding.** Natural-language artifacts such as bug reports, commit messages, and documentation are often appended as free-form prompts rather than being modeled as structured semantic evidence. Together, these issues limit the scalability, flexibility, and deployability of current APR frameworks.

Recent advances [15, 16, 17, 18, 19] have begun to explore modular or tool-assisted repair, including retrieval-augmented prompting, multi-stage execution, and collaborative multi-agent debugging [20, 21, 22]. Despite this progress, three gaps remain. First, most systems still rely on centralized or

offline training paradigms, which are impractical when project boundaries, organizational policies, or data-locality requirements restrict code sharing. Second, current pipelines provide little adaptive control: modules are invoked in predetermined sequences even when early confidence signals could terminate execution or guide targeted escalation. Third, natural-language artifacts—such as bug descriptions, commit messages, or documentation—are seldom represented as structured semantic features that interact with code-side signals.

These gaps highlight the need for a framework that preserves locality, offers adaptive control, and is grounded in semantic understanding. We introduce FedDualFix, a federated, hierarchical multi-agent framework for APR, specifically designed for distributed environments where centralizing source code is challenging. In the following discussion, we refer to the system as **FedDualFix**. The name is derived from three key aspects: **Fed** represents its operation under federated data locality constraints, **Dual** emphasizes the incorporation of meaningful insights from both source code and textual perspectives, and **Fix** signifies its primary goal of program repair.

FedDualFix couples modular agents with adaptive coordination. Its task-specialized agents perform context aggregation, localization, patch generation, semantic alignment, and confidence evaluation, all orchestrated by a hierarchical controller equipped with confidence-guided early exits [23, 24]. Lightweight trainable components are optimized locally and aggregated globally under a federated learning paradigm [25, 26], ensuring that raw code never leaves organizational boundaries. On the semantic dimension, FedDualFix combines code-side cues (slices, AST/CFG structure, diffs, diagnostics) with language-side information (bug reports, commit summaries, documentation) and computes a patch-level code–text coherence score that directly influences ranking and scheduling decisions. Natural-language artifacts thus become first-class features rather than unstructured prompts.

To address the heterogeneous complexity of real-world defects, FedDualFix organizes its agents into a three-layer hierarchical architecture. The first layer performs rapid operations such as initial localization and patch generation. Cases with low or uncertain confidence escalate to deeper layers that incorporate semantic slicing, cross-modal alignment, and repository-level context retrieval. A dynamic scheduling mechanism selectively activates agents based on uncertainty and predicted difficulty, enabling fast early exits for simple defects while reserving deeper reasoning for ambiguous or semantically rich ones—mirroring human debugging strategies and improving

computational efficiency.

We evaluate FedDualFix on Defects4J[1] under a three-client federated setting using GPT-4o for patch generation. FedDualFix repairs 58 bugs and achieves 52.1% Top-1 Patch Precision (P@1), outperforming representative APR baselines and a same-backend GPT-4o Direct baseline. This controlled comparison indicates that the gains are not solely due to the GPT-4o backend, but also stem from hierarchical multi-agent coordination, semantic slicing, cross-modal alignment, and confidence-guided patch ranking. Ablation and sensitivity studies further confirm the benefits of hierarchical control, semantic integration, and consistent performance under the evaluated federated settings.

In summary, we make the following contributions.

- **Federated, hierarchical multi-agent APR.** FedDualFix couples modular agents with *confidence-gated* hierarchical control, enabling early acceptance or targeted escalation under data-locality constraints, without sharing raw source code.
- **Code–text semantic coherence.** Instead of concatenating natural language as free-form prompts, FedDualFix jointly models code-side and text-side signals and computes a coherence score that directly guides ranking and scheduling.
- **Effectiveness and efficiency.** FedDualFix improves correctness and Top-1 precision over representative APR baselines and a same-backend GPT-4o direct repair baseline, while reducing unnecessary inference through early-exit scheduling. Ablations attribute gains to hierarchical layering and semantic integration.

We open-source our implementation and experimental artifacts [2] to support reproducibility and future research.

The rest of the paper is organized as follows. Section 2 provides an overview of related work, focusing on automated program repair and federated learning within the context of software intelligence. Section 3 introduces the design of FedDualFix, detailing its hierarchical multi-agent architecture and dual-modality repair workflow. Section 4 outlines our experimental setup, including datasets, baselines, and evaluation metrics. Section 5

[1] https://github.com/rjust/defects4j/releases/tag/v2.0.0
[2] https://github.com/Liuqy1213/FedDualFix

presents and analyzes the empirical results to answer key research questions. Section 6 discusses design implications, scheduling strategies, and validity threats. Finally, Section 7 concludes the paper by summarizing the contributions and suggesting directions for future research.

2. Background and Related Work

We organize related work around three lines most relevant to FedDualFix: generation-based APR, multi-agent APR, and federated learning for software intelligence. For each line, we compare prior studies in terms of repair paradigm, coordination mechanism, and deployment assumption, thereby highlighting the gap in federated, confidence-guided, dual-modality program repair.

2.1. Generation-based Automated Program Repair

Automated Program Repair (APR) aims to automatically localize and correct bugs without requiring direct human intervention. Early research focused on search-based techniques such as GenProg [3], RSRepair [4], and Prophet [5], which explore candidate patches through mutation and genetic search. While effective for specific bug categories, these methods [6] often suffer from limited precision and brittle generalization.

With the rise of deep learning, neural-based APR [27, 28, 29, 30, 31] has become a dominant paradigm [32], treating repair as a sequence-to-sequence translation task from buggy code to fixed code. Architectures based on pre-trained code models and syntax-guided decoders [33, 34, 35] achieve competitive performance on benchmarks such as Defects4J [36] and QuixBugs [37]. More recently, LLMs [10, 11, 12, 13, 14] have demonstrated strong capabilities in code understanding and patch generation, with models such as GPT-4 [10] and LLaMA3-8B [38, 39] integrating code semantics, comments, and external context into the repair process.

To further improve robustness, recent work augments LLM-based repair with retrieval or analysis tools—for example, IRFL and SBFL for narrowing suspicious locations [40, 41, 42, 43, 44, 45]—and incorporates natural-language artifacts such as bug reports, commit messages, or stack traces [46, 47, 48]. Recent vulnerability-repair studies show that CVE/CWE-related textual evidence and structured security knowledge can help guide vulnerability verification, repair-pattern selection, and patch generation [49, 50]. Despite these advances, most pipelines still invoke repair components in fixed se-

quences and treat language inputs as loosely structured prompts rather than jointly modeling semantic features. Moreover, prior work typically assumes a centralized code corpus during training or evaluation, which limits applicability in distributed or cross-organizational development environments.

Overall, existing APR techniques have progressed from search-based and template-based repair to neural and LLM-assisted generation, but they still largely follow centralized and relatively fixed repair workflows. Search-based and template-based methods depend on predefined operators or historical repair patterns, while neural and LLM-based approaches commonly formulate repair as a generation problem over available code context. In contrast, FedDualFix focuses on adaptive and locality-preserving repair: it escalates analysis only when intermediate confidence is insufficient, models natural-language evidence as structured semantic signals rather than auxiliary prompts, and coordinates lightweight trainable components across decentralized clients without sharing raw project artifacts.

2.2. Multi-Agent Automated Program Repair

Large language models (LLMs) have motivated the design of multi-agent workflows in software engineering [7, 8]. By decomposing repair into explicit subtasks—such as localization, generation, validation, and refinement—multi-agent systems [51, 52] allow LLMs to operate over well-scoped responsibilities and interface cleanly with static analysis, test execution, and retrieval modules. This modular decomposition improves transparency and facilitates fine-grained control over how contextual information is gathered and used.

A variety of LLM-based APR frameworks instantiate this division of labor in different ways. RepairAgent [53] employs a tool-augmented agent that plans repair actions and validates them via tests and analyzers. UniDebugger [20] organizes reasoning into a hierarchical three-level workflow, escalating from quick heuristics to repository-scale analyses. VarPatch [54] focuses on variable-addition defects through an iterative conversational loop, while Smartify [55] tailors multi-agent repair to smart-contract languages using retrieval and vulnerability signals. SemAgent [56] integrates issue semantics, execution traces, and code features into a combined repair–review pipeline. More recent systems, such as RAMP [21] and MA-Critic [22], further specialize roles by introducing reflection-driven refinement loops or syntax- and semantics-oriented critics for multi-round patch improvement.

At the same time, recent work has questioned whether multi-agent decomposition is always necessary. Agentless [9] argues that a carefully de-

signed two-stage pipeline can match or outperform multi-agent frameworks in certain settings, highlighting that benefits from agent decomposition often depend on the nature of task segmentation and the availability of supporting tools. This perspective suggests that modularity should be used strategically—where it introduces meaningful structure—rather than as a default choice.

Across these diverse designs, several structural limitations remain. Most multi-agent APR frameworks still operate under centralized assumptions, evaluating on a single project or a consolidated codebase; such setups limit applicability when organizational boundaries or data-locality requirements prevent sharing raw repositories. In addition, agent execution flows are usually fixed, invoking every component regardless of defect complexity or the confidence of intermediate patches. This rigidity leads to unnecessary computation and prevents the system from allocating reasoning effort adaptively. Finally, natural-language artifacts—such as bug descriptions, commit summaries, and documentation—are often incorporated as loosely formatted prompts rather than as structured semantic signals that directly contribute to patch ranking or decision-making. These factors collectively restrict the flexibility and deployability of existing multi-agent repair pipelines, especially in distributed development environments.

Compared with existing multi-agent APR systems, FedDualFix emphasizes three differences: data-locality preservation, adaptive depth control, and structured dual-modality alignment. Existing systems improve repair reasoning through tool use, reflection, retrieval, or critic-based refinement, but they generally assume centralized access to project artifacts and rely on relatively fixed agent workflows. In contrast, FedDualFix allows each client to perform repair locally while sharing only lightweight calibration updates. Its confidence-gated hierarchical scheduler adjusts the depth of reasoning based on patch uncertainty, enabling rapid early acceptance for straightforward defects while allocating deeper analysis only when necessary. Moreover, language-based evidence is modeled as structured semantic cues that directly inform patch ranking and decision-making, rather than being used only as loosely formatted prompt context.

2.3. Federated Learning for Software Intelligence

Federated learning (FL) is a distributed training paradigm in which multiple clients collaborate on a shared model by exchanging model updates rather

than raw data [25, 26]. In software engineering workflows, this setup is particularly relevant when repositories, bug reports, or development artifacts cannot be consolidated due to organizational boundaries or data-governance policies. By maintaining data locality and aggregating only parameter updates, FL enables cross-project collaboration while accommodating heterogeneous development environments. Prior work has explored FL for tasks such as defect prediction [57], code summarization [58], and vulnerability detection [59], demonstrating the viability of decentralized training in software intelligence.

A range of optimization strategies addresses challenges arising from non-IID client distributions. FedAvg provides a simple aggregation rule for combining local updates [60]. FedProx introduces a proximal term to stabilize local objectives under heterogeneous data [61], and FedDyn employs dynamic regularization to mitigate client drift during local training [62]. Further work investigates robustness and fairness in federated settings, including consistency-based aggregation [63] and prototype-aligned adaptation [64]. These developments illustrate a growing interest in making FL stable and effective in realistic, multi-project software environments.

The application of FL to APR remains relatively underexplored. Unlike classification or retrieval tasks, APR requires generating syntactically valid and semantically coherent patches whose distributions vary substantially across projects, languages, and coding styles. This variability introduces practical difficulties. Local repair behaviors may diverge across clients, and naïve aggregation can amplify inconsistent update directions or integrate low-quality patterns. Moreover, APR pipelines typically combine multiple components—localization, synthesis, validation, and ranking—raising questions about which components should be trained locally, which should remain fixed, and how to evaluate the overall impact of repairs in a federated environment.

In this work, we adopt FedAvg as the aggregation method for the lightweight trainable components of FedDualFix, while the LLM backend operates strictly in inference mode. Our primary focus is not on optimizing federated objectives per se, but on examining how federated coordination can support hierarchical repair when clients maintain private repositories and update only small auxiliary modules. Unlike prior FL-based software intelligence studies that mainly focus on prediction, summarization, or representation learning, FedDualFix applies federated coordination to patch generation, where outputs must be compilable, test-passing, and semantically coherent. Thus,

federated learning in FedDualFix serves as a lightweight mechanism for calibrating alignment and confidence signals across decentralized repair clients, rather than a full-model training framework. Although FedDualFix supports an optional reliability-weighted variant (see Section 3.1) that generalizes FedAvg, we retain standard FedAvg in our main experiments to isolate the impact of hierarchical multi-agent scheduling.

3. Methodology

3.1. Overview of FedDualFix

FedDualFix is a federated multi-agent framework designed to support automated program repair across distributed and organizationally separated codebases. The framework follows a locality-preserving design: each client executes a complete yet lightweight repair pipeline locally, while only small trainable components are shared for coordination. Within a client, specialized agents collaborate to perform context aggregation, fault localization, patch generation, semantic evaluation, and confidence estimation. Raw project artifacts remain isolated within each client, and the LLM backend is used in inference-only mode rather than being updated during federated training.

Formally, let client i hold a local repair dataset $\mathcal{D}_i = (b_j, \mathcal{T}_j, \mathcal{E}_j)j = 1^{n_i}$, where b_j denotes a buggy program, $\mathcal{T}_j$ denotes optional textual artifacts such as bug reports, commit messages, or documentation, and $\mathcal{E}_j$ denotes available execution evidence such as failing tests, stack traces, or compiler diagnostics. Given $(b_j, \mathcal{T}_j, \mathcal{E}_j)$, the goal of FedDualFix is to generate a patch g_j that passes the official tests and is semantically consistent with the developer fix. The repair pipeline, therefore, optimizes not only syntactic validity and test-suite behavior, but also semantic agreement between code-side changes and textual evidence.

At the core of FedDualFix is a hierarchical control strategy that adapts repair effort to the difficulty of the defect. Rather than invoking all agents exhaustively, FedDualFix organizes repair into three progressively more resource-intensive layers and employs a confidence-gated scheduler to determine whether a candidate patch should be accepted early or escalated for deeper analysis. This design enables fast resolution for simple defects while reserving richer structural and semantic reasoning for ambiguous cases.

Three-layer adaptive control. As illustrated in Figure 1, let $C(g) \in [0, 1]$ denote the confidence score of a candidate patch g produced by *ConfEvalua-*

tion, and let (θ_1, θ_2) be two calibrated scheduling thresholds with $\theta_1 > \theta_2$. A patch exits immediately when $C(g) \geq \theta_1$; cases with $\theta_2 \leq C(g) < \theta_1$ trigger the second layer; and cases with $C(g) < \theta_2$ escalate to the deepest layer.

L1 — Fast Repair. In the first layer, *Localization* identifies candidate repair anchors and *Synthesis* generates an initial patch using the local LLM backend. *ConfEvaluation* then estimates its reliability. When $C \geq \theta_1$, the patch is returned directly, minimizing latency and avoiding unnecessary agent invocations.

L2 — Structure-Aware Repair. When $\theta_2 \leq C < \theta_1$, FedDualFix augments repair with structure- and semantics-aware analysis. *Slicing* extracts dependency- and semantics-based code slices, while *Alignment* incorporates available language-based evidence. *Alignment* verbalizes candidate edits, evaluates their coherence with textual artifacts, and produces a compact semantic summary. This enriched context is used to re-invoke *Localization* and *Synthesis*, followed by an updated confidence estimate.

L3 — Feedback-Based Refinement. For low-confidence cases, where $C(g) < \theta_2$, FedDualFix expands the search space using *Aggregation* and *Scoping*, and enters a bounded refinement loop governed by *Refinement*. Each iteration regenerates candidates, analyzes structural and semantic cues, updates confidence through *ConfEvaluation*, and either performs local adjustments or reanchors the search position. The loop terminates when a high-confidence patch is found or a maximum number of rounds $K_{\max}$ is reached.

Beyond local repair, FedDualFix supports lightweight federated coordination. Let ϕ_i^r denote the trainable lightweight parameters uploaded by client i at communication round r, and let $n_i = |\mathcal{D}i|$ be the number of local repair instances. The server aggregates client updates using client-size-weighted FedAvg:

$$\phi^{r+1} = \sum_{i=1}^{N} \frac{n_i}{\sum_{j=1}^{N} n_j} \phi_i^r, \tag{1}$$

Only lightweight scoring and calibration components participate in this aggregation, while the LLM backend remains frozen and inference-only. When textual artifacts are available, language-based evidence directly influences patch ranking and scheduling via *Alignment*. In their absence, FedDualFix falls back to structural code signals, ensuring robust behavior across projects with varying documentation quality.

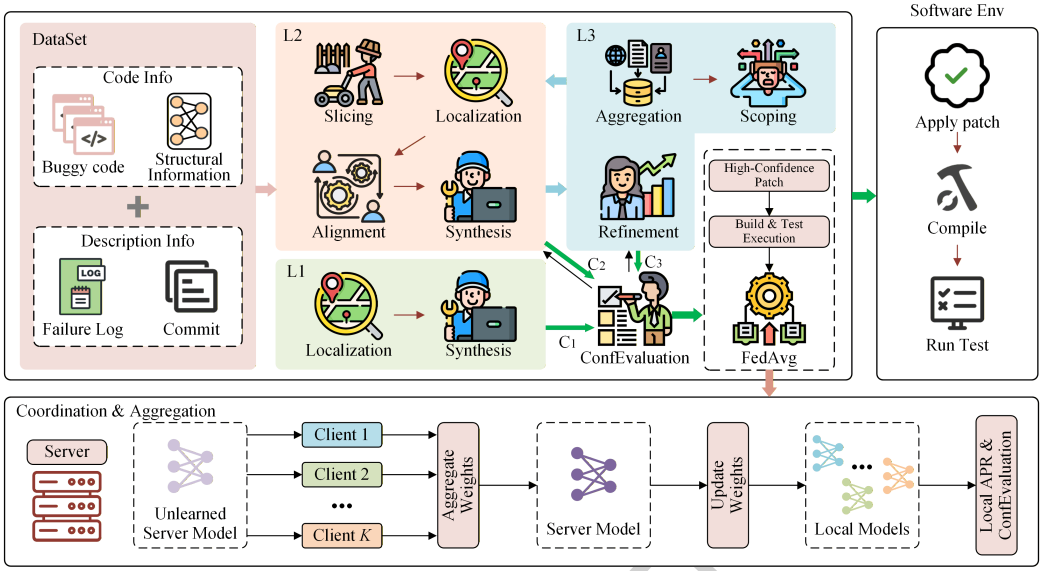

Figure 1: Framework of FedDualFix, a federated multi-agent program repair system. The framework is structured into three layers: (1) **Fast Repair Layer**, which quickly generates initial patches using minimal context; (2) **Structure-Aware Layer**, which enhances repair quality with static analysis, semantic slicing, and dual-modality support; and (3) **Feedback-Based Refinement Layer**, which evaluates and improves low-confidence patches through iterative refinement for low-confidence candidates.

3.2. Specialized Multi-Agent Modules

FedDualFix decomposes local repair into a coordinated set of specialized agents, each handling a distinct stage of analysis, generation, or validation. Instead of acting as standalone utilities, these agents collaborate in a structured workflow: project-level context is consolidated, the search space is progressively narrowed and sliced, faults are localized at fine granularity, candidate patches are synthesized, and their semantic and structural plausibility is assessed. Figure 2 provides a system-level overview of these agents, their inputs and outputs, and their placement within the federated aggregation loop. Collectively, they form a transparent, adaptive on-client repair pipeline capable of multimodal reasoning across both code and textual artifacts.

For clarity, the scoring coefficients in Scoping, Slicing, and Localization are treated as validation-calibrated heuristic coefficients rather than federated trainable parameters. They are initialized uniformly, selected on a held-out validation split, and kept fixed during testing. Therefore, the coefficients

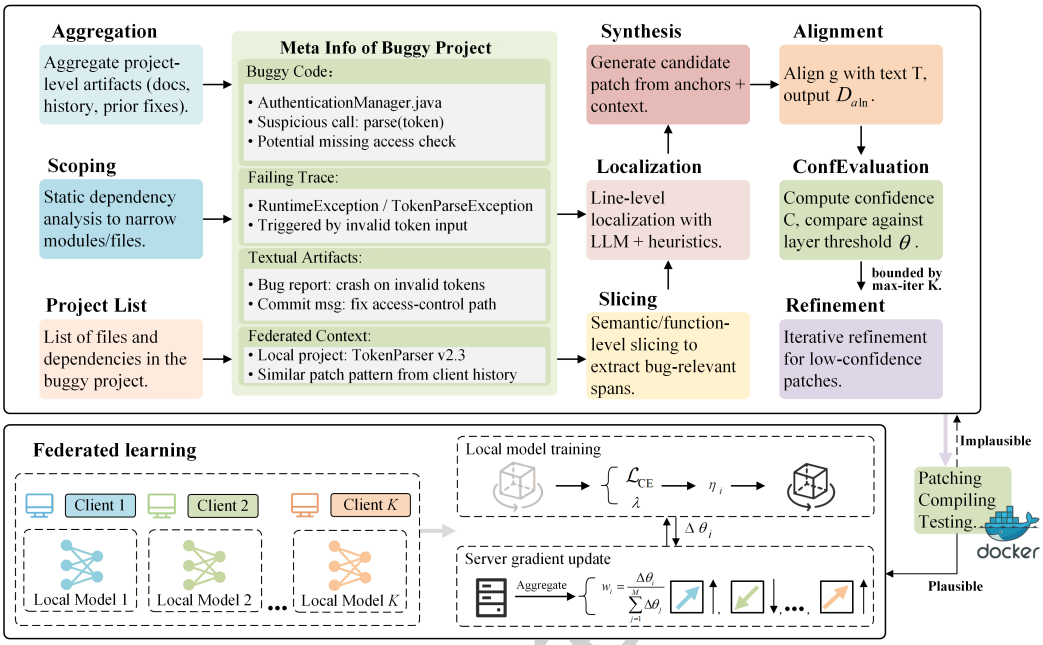

Figure 2: Module-level overview of FedDualFix: Inputs/Outputs of each agent and the federated aggregation loop.

in Equations (2)–(4) are not updated by FedAvg. Federated aggregation is applied only to lightweight scoring and calibration components used in Alignment and ConfEvaluation, such as alignment fusion weights and confidence-fusion heads.

Aggregation *(project-level context cache).* Gather consolidates project-wide information—repository metadata, documentation, commit logs, dependency structures, and prior fixes—into a unified context store. Its role is not to make repair decisions but to maintain a stable knowledge base that downstream modules can query when performing narrowing, slicing, or refinement. By normalizing heterogeneous sources into a consistent schema, Gather enables later modules to access API relationships, call-site structure, and historical fix patterns without repeatedly scanning the entire repository.

Scoping *(scope narrowing by static cues).* Narrow prioritizes files or modules that are most likely related to the defect. Using the contextual indices produced by **Aggregation**, it integrates lexical relevance, dependency proximity, and historical change frequency into a unified relevance score:

$$R(f) = \lambda_t \,\mathrm{TFIDF}(f, \mathrm{bug}) + \lambda_d \, e^{-\mathrm{dist}_{\mathrm{dep}}(f,\hat{m})} + \lambda_h \,\mathrm{churn}(f), \qquad (2)$$

where TFIDF(f, bug) measures textual similarity between file content and the bug description, $\text{dist}_{\text{dep}}(f, \hat{m})$ denotes the dependency-graph distance from a seed module $\hat{m}$, and churn(f) reflects file-level change frequency. The coefficients λ_t, λ_d, and λ_h are validation-calibrated weights that control the relative contributions of lexical relevance, dependency proximity, and historical change frequency. Files with higher $R(f)$ are forwarded to *Slicing* for deeper semantic analysis.

Slicing *(semantic/structural slicing).* Slice extracts fine-grained semantic units—functions, code blocks, or dependency-connected regions—that are most indicative of the defect locus. Instead of re-ranking files, Slice focuses on identifying the internal spans most correlated with the failure by combining data-flow coverage, code-text semantic similarity, and bug-pattern cues:

$$S(s) = w_{\text{dep}} \, \text{cov}_{\text{dep}}(s) + w_{\text{sim}} \, \text{sim}(s, \text{bug}) + w_{\text{pat}} \, \text{pat}(s), \tag{3}$$

where $\text{cov}_{\text{dep}}(s)$ quantifies dependency/data-flow coverage of slice s, $\text{sim}(s, \text{bug})$ is the cosine similarity between the slice representation and the bug description, and $\text{pat}(s)$ encodes defect-pattern or anti-pattern evidence. The coefficients w_{dep}, w_{sim}, and w_{pat} are calibrated on the validation split and then fixed during testing. Slices with higher $S(s)$ are forwarded to *Localization*.

Localization *(fine-grained repair anchors).* Locate refines the results produced by *Slicing* by identifying the most plausible faulty tokens or statements. It integrates three complementary signals—LLM plausibility, heuristic indicators, and AST validity—into a unified anchor score:

$$L(\tau) = \alpha s_{\text{LLM}}(\tau) + \beta h_{\text{heur}}(\tau) + \gamma v_{\text{AST}}(\tau), \tag{4}$$

where $s_{\text{LLM}}(\tau)$ reflects model-based suspicion, $h_{\text{heur}}(\tau)$ captures defect patterns or anti-patterns, and $v_{\text{AST}}(\tau)$ verifies structural consistency in the AST. The coefficients α, β, and γ are validation-calibrated weights. Anchors exceeding a threshold κ form the candidate anchor set:

$$\mathcal{A} = \{\tau \mid L(\tau) \geq \kappa\}. \tag{5}$$

Synthesis *(candidate patch synthesis).* Repair transforms the anchors produced by *Localization* into concrete code edits. Given anchors $\mathcal{A}$, local context, and any available textual clues (e.g., bug reports), it produces k patch candidates through structure-aware prompting:

$$\mathcal{G} = \text{Decode}(\text{LLM}, \, \text{prompt}(\mathcal{A}, \text{ctx}, \mathcal{T}); \, M), \tag{6}$$

where prompt$(\cdot)$ composes the generation context, $\mathcal{T}$ denotes optional textual artifacts, and M is the maximum number of candidate patches. Each candidate $g \in \mathcal{G}$ is passed to *Alignment* and *ConfEvaluation* for semantic verification and confidence scoring.

Alignment *(semantic alignment and lightweight representation).* Align assesses how well a proposed patch aligns with the available textual evidence and the buggy code's structural intent. It verbalizes each patch into a natural-language description $\sigma(g)$ and computes a cross-modal alignment score using three signals: textual entailment, structural overlap, and embedding similarity:

$$D_{\text{aln}}(g, \mathcal{T}; g_{\text{ctx}}) = \boldsymbol{\eta}^{\top} \begin{bmatrix} \text{NLI}_{\text{entail}}\big(\sigma(g), \mathcal{T}\big) \\ \text{CodeBLEU}\big(g, g_{\text{ctx}}\big) \\ \cos\big(\psi(g), \psi(\mathcal{T})\big) \end{bmatrix}, \qquad \boldsymbol{\eta} \in \Delta_3, \tag{7}$$

where $\mathcal{T}$ denotes textual artifacts, g_{ctx} is the reference context code, $\psi(\cdot)$ is a shared embedding function over code and text, and $\boldsymbol{\eta}$ is a lightweight trainable alignment head. The verbalization $\sigma(g)$ also serves as a state summary for *Refinement*.

ConfEvaluation *(confidence scoring and gating).* ConfEvaluate integrates syntactic validity, AST consistency, semantic alignment, and candidate-level uncertainty into a single confidence value:

$$C(g) = \sigma\left(w_0 + w_1 D_{\text{syn}}(g) + w_2 D_{\text{ast}}(g) + w_3 D_{\text{aln}}(g, \mathcal{T}) - w_4 U(g)\right), \tag{8}$$

where $D_{\text{syn}}(g)$ and $D_{\text{ast}}(g)$ represent syntax and AST diagnostics, $D_{\text{aln}}(g, \mathcal{T})$ is the alignment score from *Alignment*, and $U(g)$ is an uncertainty score. The weights $\{w_0, \ldots, w_4\}$ form a lightweight confidence-fusion head that can be calibrated locally and aggregated in the federated stage. To avoid relying on unavailable, truncated, or provider-dependent token-level log probabilities from commercial LLM APIs such as GPT-4o, we define uncertainty using candidate-level self-consistency:

$$U(g) = 1 - \frac{n_{\max}}{M}, \tag{9}$$

where M is the number of sampled candidate patches under the same repair context, and $n_{\max}$ is the size of the largest consistency cluster. Candidate patches are clustered using normalized edit distance, AST-level equivalence,

and test behavior. A lower $U(g)$ indicates more stable candidate generation, whereas a higher value indicates greater uncertainty and increases the likelihood of escalation.

Refinement *(iterative refinement loop).* Refine handles difficult cases by iteratively improving anchor selection and prompting. At each round k, diagnostics and alignment feedback are used to adjust the anchor–prompt pair $(A, P)_k$, after which new candidates are decoded:

$$(\mathcal{A}, P)_{k+1} = \mathcal{U}((\mathcal{A}, P)_k;\, \text{diagnostics}_k,\, D_{\text{aln},k},\, \text{Ctx}), \tag{10a}$$

$$g_{k+1} = \arg \max_{g \in \text{Decode}(P_{k+1}, \mathcal{A}_{k+1})} C(g), \tag{10b}$$

$$\text{stop if} \quad C(g_{k+1}) \geq \theta_1 \text{ or } k{+}1 = K_{\max}. \tag{10c}$$

Here $\mathcal{U}(\cdot)$ updates anchors and prompts using diagnostics, alignment feedback, and contextual evidence. Decode$(\cdot)$ regenerates candidates, and $C(g)$ selects the highest-confidence candidate at each refinement step.

The agents in FedDualFix collectively decompose repair into modular, interpretable stages—from coarse narrowing to semantic alignment and iterative refinement. By assigning each stage to a specialized component, the framework maintains transparency while supporting adaptive control under federated constraints.

3.3. Hierarchical Control and Feedback Coordination

FedDualFix regulates the repair process through a three-level hierarchical controller that adapts the depth of reasoning to the difficulty of each defect. Rather than executing every module in a fixed sequence, the controller integrates signals from the agents described in Section 3.2, including file relevance, slice scores, anchor scores, semantic alignment, diagnostics, and candidate-level uncertainty. These signals are fused by *ConfEvaluation* into a patch-level confidence score $C(g) \in [0, 1]$. The controller then selects the next action according to:

$$\textsc{Decision}(g) = \begin{cases} \textsc{Accept}, & C(g) \geq \theta_1, \\ \text{L2}, & \theta_2 \leq C(g) < \theta_1, \\ \text{L3}, & C(g) < \theta_2, \end{cases}$$

Here, θ_1 and θ_2 are configurable scheduling thresholds rather than universal constants. The upper threshold θ_1 controls early acceptance: increasing θ_1

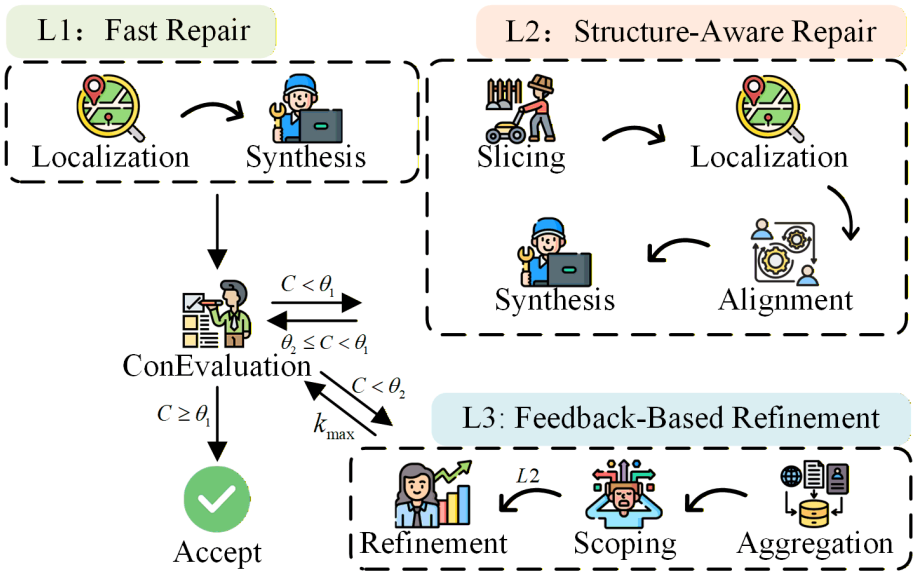

Figure 3: Hierarchical control flow in FedDualFix.

requires stronger confidence before a patch can exit at a lower layer. The lower threshold θ_2 controls escalation to L3: increasing θ_2 causes more uncertain candidates to enter the deepest refinement stage. The main experiments use the validation-selected pair $(\theta_1, \theta_2) = (0.60, 0.40)$. In practical deployments, the threshold pair can be recalibrated according to latency budgets, defect criticality, and organizational risk tolerance. By preserving intermediate signals, including alignment scores, diagnostics, updated anchors, and uncertainty estimates, the controller supports traceable transitions across repair layers without requiring a separate summarization module. Figure 3 illustrates the transitions across L1, L2, and L3, including the feedback path that returns refined candidates to lower-cost evaluation when confidence becomes sufficient.

L1 — Fast repair (open-loop). L1 handles cases where the defect is relatively easy to localize and repair. *Localization* proposes candidate anchors $\mathcal{A}$, and *Synthesis* generates an initial candidate patch g. *ConfEvaluation* computes $C(g)$. If $C(g) \ge \theta_1$, the patch is accepted immediately. This open-loop path minimizes latency by avoiding unnecessary reasoning and mirrors human developers' behavior when the fix is obvious.

L2 — Structure-aware repair (context-enriched). When $\theta_2 \le C(g) < \theta_1$,

the controller escalates to L2 and introduces deeper structural and semantic reasoning. *Slicing* refines the search space, *Localization* and *Synthesis* re-engage with enriched slices, and *Alignment* provides the updated cross-modal coherence signal $D_{\text{aln}}(g, \mathcal{T})$ and verbalization $\sigma(g)$. The revised candidate is scored again by *ConfEvaluation*. If $C(g) \geq \theta_1$, the patch is accepted; otherwise, when $C(g) < \theta_2$, the controller escalates to L3. L2 therefore acts as an intermediate repair stage: richer than L1 but less computationally intensive than L3.

L3 — Feedback-based refinement (closed loop). For complex or ambiguous cases where $C(g) < \theta_2$, the controller activates the full refinement path. In this stage, *Aggregation* and *Scoping* expand the available context by incorporating broader project and dependency information, while *Refinement* enters a bounded closed-loop adjustment cycle. Each iteration revisits the current anchors and prompts, updates them using diagnostics and semantic alignment feedback, and regenerates new candidates via *Synthesis*. *ConfEvaluation* then reassesses the new candidates and selects the highest-confidence candidate. The loop terminates when $C(g) \geq \theta_1$ or when the refinement budget $K_{\max}$ is reached. This iterative design mirrors human debugging practice: hypotheses are revised, evidence is re-evaluated, and proposed fixes evolve through successive refinement until the defect is resolved or the current budget is exhausted.

Feedback coordination. Effective coordination across layers relies on four feedback channels: (i) *diagnostics* $D_{\text{syn}}(g)$ and $D_{\text{ast}}(g)$, which enforce syntactic and structural validity; (ii) *semantic alignment* $D_{\text{aln}}(g, \mathcal{T})$, which connects code edits with textual artifacts; (iii) *uncertainty* $U(g)$, which captures instability among candidate patches and discourages premature acceptance; and (iv) *trace signals* z, which record intermediate anchors, invoked modules, confidence values, and layer transitions for reproducibility and federated analysis. By incorporating these signals, the controller transitions smoothly from open-loop L1 to context-enriched L2 and closed-loop L3, ensuring that additional reasoning effort is introduced only when justified.

Server-side coordination. At synchronization rounds, clients share updates of the lightweight trainable components together with aggregated confidence statistics, such as acceptance proportions and confidence distributions, without exposing raw code or local artifacts. The server aggregates trainable parameters using the FedAvg rule, while the LLM backend remains frozen and inference-only. Confidence statistics are used for monitoring and analysis rather than for transmitting raw repair artifacts. This design pre-

serves data locality while allowing clients to benefit from shared calibration of lightweight scoring components.

By orchestrating agents under this confidence-gated hierarchy, FedDualFix balances efficiency with robustness: simple bugs are resolved quickly in L1, more complex ones receive structure-aware context in L2, and complicated cases are refined iteratively in L3.

3.4. Prompt Schema and Token-Constrained Context Assembly

To ensure predictable and consistent behavior across LLM-driven modules, FedDualFix adopts a unified prompt schema shared by all agents that rely on generation. Each prompt instance is organized into the following 4-tuple:

$$\Pi = (\mathcal{I}, \mathcal{C}, \mathcal{T}, \mathcal{O}),$$

where $\mathcal{I}$ specifies the task instruction, $\mathcal{C}$ contains ranked code evidence, $\mathcal{T}$ includes optional textual signals, and $\mathcal{O}$ defines a constrained output format. Here, $\mathcal{C}$ is assembled from the ranked slices and anchors produced by *Slicing* and *Localization*, while $\mathcal{T}$ corresponds to the textual artifacts defined in Section 3.1, such as bug reports, commit messages, or documentation. This shared structure allows the hierarchical scheduler to vary prompt depth across layers: compact, minimally contextual prompts in L1; richer, context-enhanced prompts in L2; and semantically annotated prompts in L3 when deeper reasoning is required. Thus, prompt construction is directly coupled with the confidence-gated control policy: high-confidence cases use compact prompts for early exit, whereas low-confidence cases receive additional structural and textual evidence before regeneration.

Given the context limitations of modern LLMs, prompt assembly is formulated as a token-budget allocation problem. *Slicing* produces an ordered list of slices $\mathcal{S} = \{s_1 \succ s_2 \succ \ldots\}$, ranked by semantic relevance. Given a token budget B and a token-cost function $\text{cost}(\cdot)$, FedDualFix constructs the prompt context by selecting the largest prefix of ranked evidence whose total cost does not exceed B:

$$\mathcal{C}_B = \{s_1, \ldots, s_m\}, \quad m = \max \left\{ q \mid \sum_{r=1}^{q} \text{cost}(s_r) \leq B \right\}.$$

The selected context $\mathcal{C}_B$ is then combined with anchor-adjacent code regions, compact diagnostics, and optional textual evidence $\mathcal{T}$ to form the final

prompt context $\mathcal{C}$. High-priority elements—such as the top slice, anchor-adjacent regions, compact AST or syntax diagnostics, and verbalized summaries from *Alignment*—receive precedence. In our implementation, anchor-adjacent regions are bounded by the local context budget rather than a fixed global window, which avoids exceeding the prompt length when functions or files are unusually large. Non-essential content, including verbose comments or rarely used helper functions, is pruned automatically. This strategy ensures that prompt construction remains both budget-aware and semantically informative.

This token-constrained assembly achieves two goals. First, it stabilizes prompt structure across different LLM backends by enforcing predictable length and formatting, thereby reducing variance during inference. Second, it allows the hierarchical scheduler to dynamically adjust context richness: L1 prompts emphasize minimal evidence for rapid turnaround, whereas L2 and L3 prompts incorporate deeper structural context and auxiliary textual hints when the confidence signal is weak. The same schema also improves reproducibility because each agent receives an explicitly structured input and is required to return an output following $\mathcal{O}$, such as a unified diff, an anchor list, an alignment explanation, or a confidence report. Figure 4 illustrates a real prompt issued to *Synthesis*, combining buggy code, relevance-ranked slices, and a unified-diff specification. The LLM then produces a minimal patch in a single call, enabling consistent, reproducible repair cycles throughout FedDualFix.

4. Experimental Setup

4.1. Research Questions

To comprehensively evaluate the proposed FedDualFix framework, we design the following research questions:

RQ1: How does FedDualFix perform in comparison with representative APR baselines?

Motivation. We assess the effectiveness of FedDualFix by comparing it against representative APR techniques on Defects4J benchmarks. This evaluation examines whether FedDualFix improves repair accuracy and patch coverage under a unified experimental protocol. For reproducibility, all methods are evaluated using the same benchmark split, maximum repair attempts, and correctness criteria unless otherwise specified.

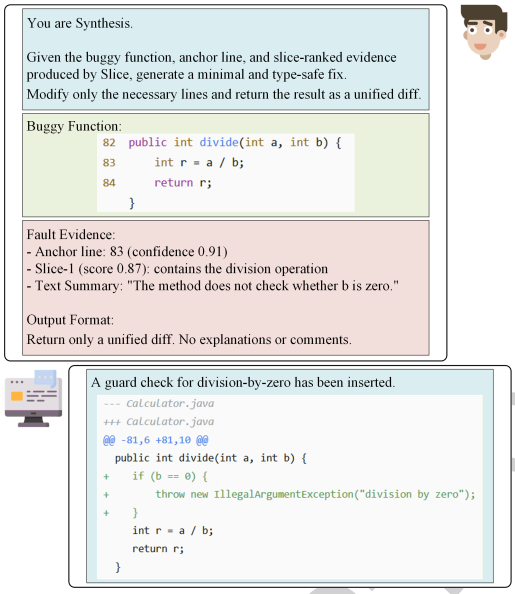

Figure 4: An example of the actual prompt issued to *Synthesis*, including the buggy code, slice-ranked evidence, and the unified-diff patch generated by the model in a single LLM call.

RQ2: What is the performance of FedDualFix when integrated with various LLMs?

Motivation. Since FedDualFix relies on LLMs in several agent modules, we investigate how its performance varies when switching among different models. This helps evaluate the system's adaptability and robustness across varying model capacities and inference behaviors.

RQ3: How do the different agent layers and specialized components within FedDualFix influence its performance in the hierarchical repair process?

Motivation. To understand the impact of hierarchical agent composition, we conduct ablation studies on individual layers and specialized modules. This reveals the contribution of each component to the repair's overall effectiveness. This includes analyzing both structural modules and cross-modal components such as *Alignment*, which represents the system's dual-modality support.

RQ4: What is the impact of a federated environment and confidence-guided agent scheduling on the performance and coordination efficiency of FedDualFix?

Motivation. We investigate how federated training and confidence-guided scheduling influence repair accuracy and resource allocation. The goal is to assess whether these mechanisms enhance client-level scalability and the efficiency of agent invocation.

4.2. Datasets

To ensure consistent evaluation across all research questions, we adopt Defects4J as the primary benchmark throughout our experiments. Defects4J contains over 800 real-world bugs spanning 17 mature open-source Java projects. Each bug instance provides a buggy version, a corresponding human-written patch, and a comprehensive test suite, enabling rigorous assessment of both patch correctness and behavioral preservation. Unless otherwise specified, we use the official Defects4J bug IDs and exclude only those instances that cannot be checked out, compiled, or tested successfully under our execution environment. The final list of evaluated bug IDs is fixed across all experiments and is provided in the replication package.

We partition the Defects4J benchmark into three disjoint client splits to simulate a federated repair setting. By default, we adopt a *project-wise, non-overlapping* split that mimics organizational boundaries: each client owns a disjoint subset of Defects4J projects, and no project is shared across clients. This setting induces natural domain skew due to project-specific coding styles, test distributions, and defect patterns. The same client partition is used throughout RQ1–RQ4 to maintain experimental consistency and ensure comparable results. The mapping from bug IDs to clients has been fixed and released to ensure reproducibility.

To strengthen empirical validation beyond a single benchmark, we further incorporate three supplementary datasets: QuixBugs [3], CodeFlaws [4], and ConDefects [5]. These datasets are selected to evaluate different aspects of external validity rather than simply increasing the number of test cases. QuixBugs comprises 40 small algorithmic programs with injected bugs, allowing us to examine repair behavior on compact algorithmic tasks and analyze LLM variability under simplified settings. CodeFlaws contains thousands of short C programs derived from programming contests, providing a

[3] QuixBugs: https://github.com/jkoppel/QuixBugs
[4] CodeFlaws: https://github.com/codeflaws
[5] ConDefects: https://github.com/appmlk/ConDefects

cross-language setting for evaluating whether the repair pipeline can transfer beyond Java. ConDefects is organized into naturally partitioned client groups with metadata labels, making it suitable for analyzing non-IID client heterogeneity and federated repair behavior.

The supplementary datasets are used only for external validity analysis in Section 5.2, while Defects4J remains the primary benchmark for the main RQ1–RQ4 comparisons. This design avoids mixing heterogeneous datasets in the main evaluation and allows us to separately analyze controlled Java APR performance, algorithmic repair generalization, cross-language robustness, and federated adaptability under heterogeneous client distributions. For each supplementary dataset, we keep the same evaluation metrics and repair-attempt budget as Defects4J whenever applicable. Dataset-specific preprocessing scripts and instance lists are included in the replication package.

Beyond the default project-wise split, we also evaluate FedDualFix under *controlled* non-IID conditions. Client distributions are varied using a Dirichlet prior [65] over project and patch-type proportions with $\alpha \in 0.3, 1.0, \infty$, where smaller α indicates stronger heterogeneity and $\alpha = \infty$ approximates an IID setting. To make this process reproducible, all Dirichlet-based partitions are generated using a fixed random seed of 42, and the resulting split manifests are released in the replication package. Non-IIDness is quantified as the pairwise mean base-2 Jensen–Shannon divergence of client-wise distributions over projects and patch types, with Laplace smoothing $\epsilon = 10^{-6}$. Under the default project-wise split, the mean JS divergence is **0.41**, confirming substantial client skew; under Dirichlet splits, JS divergence varies monotonically with α.

4.3. Baseline Methods

To evaluate the effectiveness of FedDualFix, we consider two complementary comparison settings. First, we conduct faithful system-level comparisons with representative APR methods spanning four mainstream paradigms: template-driven techniques, neural language model-based generators, structure-aware encoders, and multi-stage or component-driven repair pipelines. These methods retain their native implementations, protocols, and corresponding backbones. Second, we introduce same-backend and federated controlled baselines to examine specific factors, including the effect of the GPT-4o backend and federated aggregation. This distinction allows us to assess overall

repair effectiveness while avoiding the interpretation that all performance differences arise solely from the proposed architecture.

- **Search-based and template-driven methods.** We include TBar [66] and SimFix [67] as traditional APR baselines. TBar leverages predefined fix patterns derived from historical patches, while SimFix performs context-aware candidate retrieval and token-level transplantation.

- **Pretrained model-based methods.** We evaluate CodeT5 [34], ChatRepair [68], and CoCoNut [69] to represent neural and LLM-based repair. CodeT5 and CoCoNut generate patches using pretrained or neural sequence models, while ChatRepair formulates repair as an interactive prompting task under its original LLM setting.

- **Structure-aware models.** We select GraphCodeBERT [29] as a representative structure-informed baseline that incorporates syntactic and data-flow information into code representations.

- **Multi-stage and agent-inspired repair pipelines.** We include ITER [70] and Recoder [33] as staged or component-driven baselines. ITER adopts an identify-edit-verify routine, while Recoder focuses on edit-position modeling and copy-aware decoding.

- **Same-backend controlled LLM baseline.** To control for the effect of the foundation model, we introduce *GPT-4o Direct*, which uses the same GPT-4o backend, decoding configuration, repair-attempt budget, and maximum prompt budget as FedDualFix. It receives the same buggy code and failing-test information but performs direct single-agent patch generation without hierarchical agent invocation, semantic slicing, cross-modal alignment, confidence-guided scheduling, or federated calibration. GPT-4o Direct is not intended to represent a state-of-the-art multi-agent APR system; rather, it serves as a controlled reference for assessing the contribution of the proposed framework beyond the shared GPT-4o backend.

- **Federated neural baseline.** To provide a representative federated baseline, we implement *Fed-CodeT5*, which applies FedAvg to CodeT5 under the same client partition as FedDualFix. This baseline allows us

to compare FedDualFix with a trainable neural APR model in a federated setting. Fed-CodeT5 uses the same three-client partition and testing instances as FedDualFix, while its federated training configuration is detailed in Section 4.5.

Baseline execution protocol. Because the compared APR systems rely on different repair mechanisms and model backbones, we distinguish faithful system-level evaluation from controlled comparison. Search-based, template-based, neural, structure-aware, and multi-stage baselines are evaluated using their released implementations, original protocols, and, when available, corresponding backbones. For neural baselines, we use their native checkpoints or reported training configurations rather than replacing their inference engines. Retrofitting architecture-specific systems such as CodeT5, CoCoNut, GraphCodeBERT, Recoder, or ITER to GPT-4o would alter their original architectures, training objectives, or decoding mechanisms and would therefore no longer constitute a faithful reproduction of the corresponding methods.

Consequently, comparisons with these native baselines should be interpreted as system-level comparisons: the observed performance gaps may reflect both differences in the underlying model backbones and differences in repair architecture. They are not used to attribute the full improvement exclusively to the proposed multi-agent design. The same-backend comparison with GPT-4o Direct, together with the layer-wise and module-wise ablation studies, provides the primary controlled evidence for the contribution of hierarchical coordination, semantic alignment, confidence-guided scheduling, and other framework components. Fed-CodeT5 also serves as a practical federated neural reference, although its model capacity differs from that of the GPT-4o backend used by FedDualFix.

For reproducibility, all baselines are evaluated on the same Defects4J testing instances. For methods that generate multiple candidate patches, we retain at most 5 candidates per bug and rank them using the method-specific ranking strategy. If a baseline cannot generate a patch for a given bug, the instance is counted as unrepaired rather than removed from the evaluation. For the controlled baselines we implemented, including GPT-4o Direct and Fed-CodeT5, we release the prompt templates, client splits, configuration files, and evaluation scripts in the replication package.

4.4. Performance Metrics

To assess the effectiveness of FedDualFix, we adopt a set of evaluation metrics that jointly measure semantic repair quality, test-suite satisfaction, developer-facing ranking quality, and operational efficiency. This choice is motivated by the practical workflow of automated program repair: a repair system should not only generate test-passing patches, but also prioritize semantically correct patches early, reduce developer inspection effort, and maintain acceptable runtime and coordination overhead. Therefore, we report Correct Repairs (CR), Plausible Repairs (PR), Precision@1 (P@1), Time Cost, and Module Invocation Count (MIC). Together, these metrics evaluate FedDualFix from three complementary perspectives: repair correctness, patch usability, and deployment efficiency.

Correct Repairs (CR) measures the number of bugs for which a method generates a semantically correct patch. We adopt a two-stage criterion. First, a candidate patch must compile and pass all official tests; that is, it must be *plausible*. Second, semantic correctness is determined by comparing the generated patch with the developer-provided fix. If the patch is textually identical to or AST-equivalent to the developer fix, it is automatically marked as correct. Otherwise, two independent reviewers inspect whether the patch is semantically equivalent to the developer's patch. A patch is labeled as correct only when both reviewers agree.

Plausible Repairs (PR) count the number of bugs for which a method generates at least one patch that compiles and passes all official test cases. A plausible patch may be semantically correct or may overfit the test suite. Therefore, PR reflects a method's ability to generate test-passing patches, while CR further measures semantic correctness.

Precision@1 (P@1) indicates the proportion of bugs for which the top-ranked patch is correct. This metric reflects both generation quality and patch-ranking effectiveness and is particularly important in practical APR scenarios, where developers typically inspect only the highest-ranked candidate.

Time Cost denotes the average time spent per bug, covering localization, patch generation, candidate ranking, and validation. This metric reflects the practical efficiency of the repair system.

Module Invocation Count (MIC) records the number of distinct agent modules activated for each bug in FedDualFix, with repeated invocations of the same module counted only once. It serves as an indicator of

coordination breadth and the efficiency of the scheduling strategy. We report two efficiency metrics: (i) **Time/Bug**—the end-to-end wall-clock time per bug, including localization, patch generation, compilation, testing, and orchestration overhead, while excluding model training; and (ii) **MIC (Avg Agents)**—the average number of distinct agent modules activated per bug, where a lower value indicates lower coordination overhead.

All metrics are reported on the testing set of each dataset. For multi-shot systems, each method is allowed to generate up to five candidate patches per bug. Candidate patches are ranked according to the method-specific ranking protocol described in Section 4.3, and P@1 is computed using only the top-ranked patch. The official test suite and semantic inspection are then used to determine whether the selected patch is correct or plausible. This avoids oracle-based selection among multiple candidates and keeps the Top-1 evaluation consistent across methods.

To ensure reproducibility, we use the same validation procedure for all generated patches whenever applicable. Each candidate patch is first applied to the corresponding buggy program, then compiled and tested with the official benchmark test suite. Patches that fail to apply, compile, or pass any official test are counted as unsuccessful attempts. For each bug, only the results obtained within the five-attempt budget are considered. The final CR, PR, P@1, Time Cost, and MIC values are averaged over the same fixed set of testing instances for each experimental setting.

We conduct paired significance tests using results obtained from the same testing instances and matched random seeds. For per-instance binary repair outcomes, including CR, PR, and P@1, McNemar's test is used to compare the numbers of discordant successes between two methods. For the supplementary cross-dataset experiments, whose P@1 and Time/Bug values are summarized over 10 matched random seeds, we apply the two-sided Wilcoxon signed-rank test to the paired seed-level results rather than to the aggregate percentages. Holm–Bonferroni correction is applied separately within each metric family. In particular, the six supplementary-dataset P@1 comparisons consist of three datasets and two baselines per dataset. Unless otherwise specified, statistical significance is determined using an adjusted $p < 0.05$. Table-specific superscripts identify the corresponding comparison baseline, and unmarked differences are interpreted as numerical rather than statistically established improvements.

4.5. Implementation Details

We implemented FedDualFix in Python 3.10 using PyTorch 2.0. All locally executed components and released baselines were run on a Windows 10 workstation equipped with an Intel Core i9 (3.50 GHz) CPU and an NVIDIA GeForce RTX 4090 GPU with 24 GB VRAM. The system combines lightweight trainable modules with an LLM backend that remains frozen and is accessed only during inference.

Following the baseline execution protocol described in Section 4.3, all methods were evaluated on the same fixed testing instances and were allowed at most five repair attempts per bug. Released APR baselines, including TBar, SimFix, CodeT5, CoCoNut, GraphCodeBERT, Recoder, and ITER, retain their native implementations, checkpoints, decoding mechanisms, and corresponding backbones. ChatRepair is evaluated under its original LLM setting. Model-specific decoding parameters are not forcibly standardized when doing so would alter a method's original execution protocol.

For the same-backend controlled comparison, FedDualFix and GPT-4o Direct use the same GPT-4o backend, with temperature 0.2, top-$p = 0.95$, a maximum generation length of 256 tokens, the same maximum per-attempt prompt budget, and the repair-attempt budget defined above. GPT-4o Direct receives the same buggy code and failing-test evidence as the initial repair stage of FedDualFix, but performs direct single-agent generation without semantic slicing, cross-modal alignment, confidence-guided hierarchical scheduling, confidence fusion, or federated calibration. Thus, the foundation-model and decoding conditions are controlled specifically in the comparison between FedDualFix and GPT-4o Direct, whereas the remaining baselines retain their native settings for faithful system-level evaluation.

In the federated comparison, Fed-CodeT5 and FedDualFix use the same three-client project-wise partition, five communication rounds ($R = 5$), and client-size-weighted FedAvg rule. Because their trainable scopes differ substantially, model-specific local optimization hyperparameters are selected separately on the same validation split and fixed before testing: Fed-CodeT5 updates CodeT5 parameters, whereas FedDualFix keeps GPT-4o frozen and aggregates only lightweight scoring, alignment-calibration, and confidence-fusion modules. No testing instances are used for hyperparameter selection. We treat $R = 5$ as a practical communication budget rather than as evidence of theoretical convergence.

The scheduling thresholds are selected by grid search on a held-out validation subset comprising 10% of the training instances, sampled proportionally

within each client and project, and kept disjoint from the test set. We search θ_1 from 0.50 to 0.70 and θ_2 from 0.30 to 0.50 in increments of 0.05, subject to $\theta_1 > \theta_2$. For each candidate pair, we record P@1, average repair time, and MIC. The default pair is selected by equally weighting normalized P@1 and inverse normalized average repair time, with lower MIC used to break ties. This procedure selects $(\theta_1, \theta_2) = (0.60, 0.40)$ for the main experiments. For a new deployment, the same validation-guided search can be repeated on local validation data, with the quality–latency weights or an explicit latency constraint adjusted to match the operational requirements.

For FedDualFix, candidate patches are ranked using the confidence score C produced by ConfEvaluation. Baselines retain their native ranking mechanisms. For GPT-4o Direct, compilable patches are prioritized, followed by patches satisfying more available tests, with ties resolved by generation order. Only the top-ranked candidate is used for P@1 evaluation.

For uncertainty estimation, ConfEvaluation does not use provider-dependent token-level log probabilities. For each repair context, we sample up to $M = 5$ candidate patches under the same decoding budget and compute $U(g)$ from candidate-level self-consistency. Candidates are grouped using normalized edit distance, AST-level equivalence, and test behavior, where $n_{\max}$ denotes the size of the largest consistency cluster. A larger $n_{\max}/M$ indicates a more stable generation.

We use the official benchmark test suites as test oracles and execute them through a Docker-based validation harness. A patch is labeled correct if it passes all tests and semantically matches the reference fix, with manual inspection applied when necessary; a test-passing but semantically inconsistent patch is labeled plausible. Configurations involving stochastic sampling or trainable components are repeated 10 times with different random seeds, and intermediate artifacts such as slices, alignment outputs, and confidence traces are cached to improve consistency and reduce repeated computation.

Repair inference and federated calibration are instrumented separately. For repair inference, agent-level pipeline time and test-harness time are recorded independently. For federated calibration, we measure client-side local update time, parameter transfer time, server aggregation time, synchronization wait time caused by the slowest client, and total round-trip completion time. We also introduce controlled delays to one client to evaluate straggler effects. Because all clients are simulated on the same workstation, the reported latency characterizes the controlled experimental environment rather than a geographically distributed deployment. Communication-round sensitivity and

Table 1: LLM invocation statistics of FedDualFix on Defects4J.

Metric	Value
Avg. LLM calls / bug	3.8
– Locate	1.0
– Repair	1.8
– Align	0.6
– ConfEvaluate	0.4
Avg. input tokens / call	648
Avg. output tokens / call	173
Avg. tokens / bug	3.3k
API latency / call (s)	0.92
Approx. cost / bug (USD)	0.007–0.009

the measured calibration overhead are analyzed in Sections 6.3.1 and 6.3.2, respectively.

To support reproducibility, Table 1 reports aggregated LLM usage statistics, including average calls, token counts, API latency, and approximate cost per bug. The LLM backend remains frozen within each experimental configuration. Therefore, the privacy benefits of federated learning mainly concern client-to-client isolation of code and execution traces, rather than hiding source code from the LLM provider. For stronger privacy requirements, FedDualFix can be deployed with local LLMs such as Llama-3 or StarCoder-2, while the federated coordination mechanism remains unchanged.

5. Experimental Results

5.1. RQ1: How does FedDualFix perform in comparison with representative APR baselines?

Approach. To evaluate the effectiveness of FedDualFix, we compare it against a set of representative APR baselines across three major categories: template-based methods (TBar and SimFix), neural generation-based methods (CodeT5, CoCoNut, and ChatRepair), and structure-aware or multi-stage systems (GraphCodeBERT, Recoder, and ITER). These baselines cover the main methodological spectrum of modern APR systems and serve as representative references for evaluating repair effectiveness, ranking quality, and runtime efficiency.

We conduct all evaluations on the Defects4J benchmark. For FedDualFix, the benchmark is partitioned into three disjoint clients, each of which retains its local data and independently executes the repair pipeline. As described in Sections 4.3 and the implementation details, the existing APR baselines retain their native implementations and corresponding backbones, whereas GPT-4o Direct uses the same GPT-4o backend and decoding configuration as FedDualFix. Accordingly, comparisons with the native APR baselines are interpreted as system-level comparisons that may reflect both backbone capability and repair architecture, while the comparison with GPT-4o Direct provides a same-backend control for examining the contribution of the proposed framework beyond the shared foundation model.

Result. Table 2 summarizes the performance of all evaluated methods. FedDualFix obtains the highest repair-quality values, producing 58 correct and 84 plausible repairs, with a Top-1 precision (P@1) of 52.1%. Compared with the best-performing native-backbone baseline, ITER, FedDualFix improves CR from 50 to 58 and P@1 from 43.3% to 52.1%. Compared with ChatRepair and Recoder, FedDualFix improves P@1 by 10.6 and 11.5 percentage points, respectively.

We further conduct paired statistical significance tests on the same Defects4J testing instances following the protocol in Section 4.4. Compared with the best-performing competing baseline for each corresponding metric, namely ITER for CR and GPT-4o Direct for P@1, the improvements of FedDualFix are statistically significant under McNemar's test ($p < 0.05$ after Holm–Bonferroni correction). These results indicate that the observed differences are unlikely to arise solely from random variation across the evaluated bugs.

The comparison with GPT-4o Direct is particularly important for interpreting the source of the improvement. GPT-4o Direct uses the same foundation model and decoding configuration as FedDualFix, but performs direct single-agent generation without hierarchical coordination, semantic slicing, cross-modal alignment, confidence fusion, or federated calibration. It correctly repairs 49 bugs and achieves 44.1% P@1, compared with 58 correct repairs and 52.1% P@1 achieved by FedDualFix. While FedDualFix benefits from the strong GPT-4o backend, this same-backend comparison provides controlled evidence that the observed improvement is not attributable to the foundation model alone and that the proposed multi-agent framework contributes additional repair capability.

Nevertheless, comparisons with native baselines such as ITER, Recoder,

Table 2: Repair performance comparison on Defects4J with explicit inference engines. GPT-4o Direct serves as a same-backend controlled baseline.

Method	Engine	CR ↑	PR ↑	P@1 ↑	Time (s) ↓
TBar	Template engine	31	54	28.5%	**9.4**
SimFix	Retrieval engine	28	49	24.3%	10.2
CodeT5	CodeT5	39	63	34.8%	13.2
CoCoNut	CoCoNut decoder	43	68	38.2%	13.9
ChatRepair	Original ChatRepair LLM	47	74	41.5%	15.6
GraphCodeBERT	GraphCodeBERT	45	70	39.1%	14.3
Recoder	Recoder edit model	46	71	40.6%	14.4
ITER	ITER pipeline	50	76	43.3%	14.9
GPT-4o Direct	GPT-4o direct prompting	49	76	44.1%	11.3
FedDualFix	GPT-4o + multi-agent pipeline	**58**†	**84**	**52.1%**†	12.2

Note: † indicates that FedDualFix significantly outperforms the best-performing competing baseline for the corresponding metric at $p < 0.05$ after Holm–Bonferroni correction.

GraphCodeBERT, and CoCoNut should be interpreted with caution. Because these methods retain their original and generally smaller backbones, the performance gaps between FedDualFix and these baselines reflect both differences in the underlying model capability and differences in repair architecture. The individual effects of hierarchical control, semantic slicing, cross-modal alignment, and confidence-guided ranking are further examined through the layer-wise and module-wise ablation studies in RQ3.

In terms of efficiency, FedDualFix maintains a competitive average repair time of 12.2 seconds despite incorporating a hierarchical agent pipeline and feedback loop. TBar remains the fastest method overall because of its lightweight template engine, although its repair effectiveness is substantially lower than that of the learning-based and LLM-assisted methods. Under their respective native execution protocols, FedDualFix reports higher repair-quality values and lower average repair time than ChatRepair, Recoder, and ITER. The runtime efficiency of FedDualFix is primarily due to its early-exit design, which allows high-confidence patches to be accepted at lower layers without invoking the complete repair pipeline.

Answer to RQ1: FedDualFix achieves the highest repair-quality values on Defects4J, with 58 correct repairs and 52.1% P@1. The comparison with GPT-4o Direct supports benefits beyond the shared backend, while the gaps to native APR baselines reflect both backbone and architectural differences.

5.2. RQ2: What is the performance of FedDualFix when integrated with various LLMs?

1) LLM Variants.

Approach. Since FedDualFix integrates LLMs to drive patch generation and semantic reasoning, the choice of LLM backend directly influences its repair accuracy, efficiency, and cost. This research question investigates how various LLMs impact the system's performance when integrated into our multi-agent pipeline.

We evaluate six widely adopted LLMs from both closed and open-source domains:

GPT-3.5-Turbo [71]: A cost-efficient API model frequently used in existing APR tools.

GPT-4[72]: A more capable but expensive model known for its multi-step reasoning abilities.

GPT-4o[73]: A recently released variant that offers faster inference and multimodal capabilities at a reduced cost.

StarCoder-15B[74]: An open-source transformer pretrained on permissively licensed code.

CodeLlama-13B-Instruct[75]: A fine-tuned model optimized for instruction-based code tasks.

CodeT5+ (small)[76]: A lightweight encoder-decoder model commonly used in baseline APR systems.

Each model is plugged into the *Synthesis* and *Alignment* agents of FedDualFix using identical prompts and context structures. All backends also share the same token-budget constraints defined in Section 3.4, ensuring consistency in context size across models.

Result. As shown in Table 3, FedDualFix achieves the best results when using GPT-4o, which produces 58 correct and 84 plausible repairs. This model also ranks highest in Top-1 accuracy (52.1%), showing strong patch quality. GPT-4 performs slightly worse on CR and P@1, despite being more powerful, possibly due to higher response times and higher API usage costs.

Table 3: Evaluating the performance of FedDualFix across different LLM configurations.

LLM	Type	Size	CR ↑	PR ↑	P@1 ↑	Cost ($) ↓
GPT-3.5-Turbo	Closed (API)	–	47	72	41.3%	0.015
GPT-4	Closed (API)	–	52	79	46.7%	0.091
GPT-4o	Closed (API)	–	**58**	**84**	**52.1%**	0.028
StarCoder-15B	Open-source	15B	50	77	44.9%	N/A
CodeLlama-13B	Open-source	13B	42	68	38.5%	N/A
CodeT5+	Open-source	220M	35	60	32.0%	N/A

GPT-3.5-Turbo is faster and more cost-effective, but it struggles with more complex bug patterns, resulting in fewer accurate patches. Among open-source models, StarCoder performs best, with 50 correct repairs and 44.9% Top-1 accuracy, outperforming CodeLlama and CodeT5+, which repair fewer bugs and rank lower in patch precision. These differences may result from model size and training coverage—StarCoder is larger and trained on a more diverse code base. CodeT5+, while lightweight and fast, often misses subtle semantic errors due to its limited capacity. This result also reveals an important limitation of FedDualFix: its effectiveness is still bounded by the underlying model's semantic reasoning and patch-generation capabilities. The multi-agent pipeline can improve context selection, patch ranking, alignment, and confidence calibration, but it cannot fully compensate for a weak backbone. Therefore, the CodeT5+ result should be interpreted as a lightweight lower-bound setting, where better deployability and lower cost come at the expense of repair accuracy on challenging benchmarks such as Defects4J.

2) External Validity: Cross-Dataset Generalization.

Approach. To assess whether the observed behavior extends beyond the primary Defects4J benchmark, we evaluate FedDualFix on CodeFlaws, QuixBugs, and ConDefects. CodeFlaws examines cross-language transfer to C programs, QuixBugs contains compact algorithmic repair tasks, and ConDefects provides a naturally heterogeneous setting for federated evaluation.

For each dataset, we compare FedDualFix with GPT-4o Direct and CodeT5+. GPT-4o Direct uses the same GPT-4o backend and therefore serves as the same backend control for evaluating the hierarchical multi-agent pipeline's contribution. CodeT5+ is retained as a lightweight neural reference, although its model capacity differs from that of GPT-4o. The controller and scheduling policy of FedDualFix remain unchanged, while only dataset-specific build,

test, syntax, and AST interfaces are adapted. We report P@1 and Time/Bug using results averaged over 10 matched random seeds. Statistical comparisons follow the paired testing protocol described in Section 4.4, with Holm–Bonferroni correction applied across the six supplementary-dataset P@1 comparisons.

Results. As shown in Table 4, FedDualFix obtains the highest mean P@1 on CodeFlaws, QuixBugs, and ConDefects. To determine whether these numerical differences are statistically supported, we compare the paired seed-level P@1 results of FedDualFix with GPT-4o Direct and CodeT5+ using the protocol defined in Section 4.4. Superscripts in Table 4 indicate comparisons that remain significant after Holm–Bonferroni correction; unmarked differences are treated only as numerical advantages.

On CodeFlaws, FedDualFix reaches 48.2% P@1, compared with 40.5% for GPT-4o Direct and 30.8% for CodeT5+. This indicates improved repair performance when the framework is transferred from Java to C, although the additional structural processing increases Time/Bug from 10.8 s for GPT-4o Direct to 11.7 s. On QuixBugs, FedDualFix achieves 55.1% P@1, compared with 48.0% and 36.5% for the two baselines, while requiring 7.0 s per bug. Because QuixBugs contains only 40 programs, its statistical results should be interpreted cautiously, particularly when a corrected comparison does not reach significance. On ConDefects, FedDualFix obtains 50.0% P@1, exceeding GPT-4o Direct by 7.3 percentage points and CodeT5+ by 16.8 percentage points, with a corresponding increase in Time/Bug.

The supplementary experiments show that FedDualFix retains the highest mean P@1 across the evaluated datasets, but the strength of the statistical evidence varies by dataset and comparison. The results, therefore, support cross-dataset applicability without assuming that every numerical improvement is statistically significant. They also reveal a consistent accuracy–runtime trade-off: the hierarchical pipeline improves repair quality relative to direct generation, but introduces moderate additional inference cost.

Table 4: Cross-dataset generalization with lightweight and same-backend baselines.

Dataset	Method	CR Rate ↑	PR Rate ↑	P@1 ↑	Time/Bug (s) ↓
Defects4J	CodeT5+	31.5%	54.1%	32.0%	**8.6**
	GPT-4o Direct	44.1%	68.5%	44.1%	11.3
	FedDualFix	**52.3%**	**75.7%**	**52.1%**	12.2
CodeFlaws	CodeT5+	28.6%	50.4%	30.8%	**7.9**
	GPT-4o Direct	38.9%	64.7%	40.5%	10.8
	FedDualFix	**46.8%**	**72.5%**	**48.2%**†,‡	11.7
QuixBugs	CodeT5+	34.2%	57.8%	36.5%	**5.2**
	GPT-4o Direct	46.9%	70.6%	48.0%	6.4
	FedDualFix	**54.6%**	**78.4%**	**55.1%**‡	7.0
ConDefects	CodeT5+	30.1%	52.6%	33.2%	**9.1**
	GPT-4o Direct	40.8%	66.4%	42.7%	12.5
	FedDualFix	**49.2%**	**73.8%**	**50.0%**†,‡	13.8

Note: Results are averaged over 10 matched random seeds. † and ‡ indicate significant P@1 improvements over GPT-4o Direct and CodeT5+, respectively, under the paired two-sided Wilcoxon signed-rank test with Holm–Bonferroni correction ($p_{\text{adj}} < 0.05$).

Answer to RQ2: FedDualFix generally benefits from more capable LLM backends, with GPT-4o yielding comparatively stronger repair quality in our evaluation, while lightweight backbones offer lower execution cost at the expense of accuracy. Across the supplementary datasets, the framework also shows consistently competitive mean P@1, although the statistical evidence varies with dataset size. Overall, these results suggest that the hierarchical multi-agent pipeline can be applied across different languages and client distributions, with a moderate increase in runtime.

5.3. RQ3: How do the different agent layers and specialized components within FedDualFix influence its performance in the hierarchical repair process?

To understand the inner workings of FedDualFix, we conduct an internal analysis of its hierarchical architecture. Rather than evaluating it against external baselines or different LLM configurations, this research question investigates how individual agent layers and specialized modules contribute to the system's overall repair performance. FedDualFix operates in three progressively complex layers—ranging from fast preliminary fixes to structure-aware reasoning and feedback-driven refinement. Each layer activates a subset of agents responsible for distinct roles in localization, generation, alignment,

and evaluation. We divide this section into two parts: the first assesses the benefits of progressively adding repair layers, and the second examines the effect of removing individual modules from the system as a whole.

1) What is the impact of the hierarchical repair layers?

Approach. We evaluate FedDualFix on the same Defects4J split with GPT-4o in three progressively sophisticated repair layers. The first layer (L1) focuses on fast localization and patching for easily localizable, low-context defects, using Locate, Repair, and ConfEvaluate. The second layer (L2) enhances reasoning through Slice and Align, enabling structure-aware repair and text-informed selection. The third layer (L3) introduces deeper analysis via Gather, Narrow, and Refine to recover complex bugs or those initially mislocalized. To evaluate the effect of these layers, we incrementally enable them and measure system performance under three configurations: L1 only, (L1 + L2), and full pipeline (L1 + L2 + L3). All experiments use GPT-4o on the same Defects4J partition.

Result. Table 6 shows a steady improvement as layers are enabled. From L1 to (L1+L2), CR rises from 43 to 52, an increase of 20.9%; PR increases from 64 to 76, up 18.8%; and P@1 improves from 39.2% to 47.0%, a gain of 7.8 points. The average time increases from 8.7s to 11.5s, a 32.2% rise. Adding L3 further increases CR from 52 to 58 (11.5% relative), PR from 76 to 84 (10.5% relative), and P@1 from 47.0% to 52.1% (a gain of 5.1 points), while time rises from 11.5s to 12.2s (up 6.1%). Overall, relative to L1, the full pipeline improves CR by 15 to 58 (34.9% increase), PR by 20 to 84 (31.3% increase), and P@1 by 12.9 points to 52.1%, with an average time increase of 3.5s to 12.2s. Paired significance tests further show that the full pipeline significantly improves CR and P@1 over the L1-only configuration ($p < 0.05$ after Holm–Bonferroni correction), confirming that the hierarchical escalation mechanism provides non-trivial gains beyond random variation.

To characterize the runtime behavior of the hierarchical controller, Table 5 reports each layer's share of agent-side pipeline time, the average end-to-end repair time for bugs terminating at that layer, the acceptance share, and the average MIC conditioned on acceptance. This breakdown clarifies how early exits keep L1 lightweight and how selective escalation concentrates cost on the remaining hard cases in L2/L3.

Beyond the absolute gains, the incremental efficiency of each step is informative. From L1 to L2, the additional correctness per extra second is approximately 3.2 (resulting in 9 more correct repairs over 2.8 additional seconds). From L2 to L3, this value rises to about 8.6 (6 more correct repairs over 0.7

Table 5: Layer-wise runtime and acceptance analysis.

Layer	Share of Pipeline Time (%)	Avg Time (s, total)	Acceptance Share (%)	Avg MIC (by accepted)
L1 (Fast Repair)	61.9	9.3	45	3.2
L2 (Structure-Aware)	30.4	13.0	35	5.0
L3 (Refinement)	7.6	16.8	20	6.7

Note: Avg Time denotes the end-to-end repair time, including an average test-harness overhead of approximately 3.2 s per bug for compilation and testing.

additional seconds). This pattern suggests that L3—although deeper—tends to be invoked selectively under the early-exit policy and converts its added computation into correctness more effectively in the remaining complex cases. P@1 moves in the same direction as CR across layers, indicating that higher layers not only increase the number of successful repairs but also improve the ranking quality of first-choice patches. Overall, these observations align with the intended division of labor: L1 addresses straightforward bugs quickly, L2 strengthens structure-aware editing and dual-modality alignment, and L3 provides targeted refinement and broader context when earlier attempts are uncertain.

The progression is internally consistent: enabling additional layers does not introduce regressions, and the total runtime increase from 8.7 to 12.2 seconds remains moderate relative to the quality gains. The selective activation implied by early exit helps contain overhead, which likely explains the higher incremental efficiency observed for L3. Though repair time increases slightly at each level, the gains in correctness and ranking justify the added effort. These results confirm that each layer introduces distinct capabilities. L1 enables early exits for straightforward bugs, L2 improves patch quality and ranking, and L3 provides robustness for more complex defects. The hierarchical structure supports a flexible, cost-aware execution model where computation scales with bug complexity.

Because L2 is the first layer to introduce *Alignment*, we briefly quantify how textual artifacts (bug reports, commit messages, docstrings) affect the selection. Consistent with our design, dual-modality primarily improves Top-1 ranking and confidence-gated early exits. When text corroborates the edit intent, candidates are accepted earlier without escalating to L3. When text is absent or noisy, the controller falls back to code-only signals. We further

substantiate this observation in the following subsection via an ablation that removes *Alignment* (Table 7).

Table 6: Evaluating the performance of FedDualFix across different layer configurations (GPT-4o).

Configuration	CR ↑	PR ↑	P@1 ↑	Avg Time (s) ↓
L1	43	64	39.2%	**8.7**
L1 + L2	52	76	47.0%	11.5
L1 + L2 + L3	**58**	**84**	**52.1%**	12.2

2) What is the contribution of individual specialized modules?

Approach. To assess the marginal value of specialized modules within FedDualFix's hierarchy without changing the evaluation scope, we conduct a component-wise ablation on four agents—*Slicing*, *Refinement*, *Alignment*, and *Scoping*—each removed from the full pipeline (L1+L2+L3) one at a time. This preserves the L1 backbone (*Localization*, *Synthesis*, *ConfEvaluation*) and avoids breaking the controller loop that couples *ConfEvaluation* with *Aggregation*. Beyond the quantitative results, we also provide a brief *mechanistic interpretation* of how each module influences controller signals (file/slice/anchor scores $R(\cdot)$, $S(\cdot)$, $L(\cdot)$, cross-modal alignment D_{aln}, diagnostics, and uncertainty), and thereby the gating thresholds (θ_1, θ_2).

Result. Table 7 shows that removing *Slicing* leads to the most significant decline: CR falls from 58 to 52 (down 6), PR from 84 to 76 (down 8), and P@1 from 52.1% to 45.6% (down 6.5 points). The next most pronounced effect occurs when *Refinement* is removed: CR decreases to 53 (down 5), PR to 78 (down 6), and P@1 to 46.4% (down 5.7 points), suggesting that iterative refinement helps turn low-confidence candidates into test-passing fixes. Removing *Alignment* produces modest yet consistent drops—CR decreases to 55 and P@1 to 48.9%—indicating benefits from cross-modal coherence and localized state summarization in ranking and early-exit behavior. The effect of removing *Scoping* is mild (CR 56; P@1 49.3%), which is plausible if its structural cues are primarily complementary to slicing and context aggregation. For component ablations, the drops caused by removing *Slicing* and *Refinement* are statistically significant for CR and P@1 ($p < 0.05$), while the effects of *Alignment* and *Scoping* are smaller but directionally consistent.

These patterns align with the intended roles of the modules. *Slicing* removes distractors before generation, so its absence most strongly affects both

Table 7: Impact of removing individual agent modules (GPT-4o).

Module Removed	CR ↑	PR ↑	P@1 ↑
Full System (All Agents)	**58**	**84**	**52.1%**
– Slice	52	76	45.6%
– Refine	53	78	46.4%
– Align	55	81	48.9%
– Narrow	56	82	49.3%

correctness and P@1. *Refinement* rescues low-confidence candidates identified in earlier layers via iterative adjustment. Introduced at L2, *Alignment* sharpens P@1 and coherence more than raw pass counts: removing it lowers P@1 while only modestly affecting CR (Table 7). Operationally, its cross-modal coherence signal D_{aln} is fused into the confidence C, which (i) promotes earlier acceptance when textual cues corroborate the edit intent, and (ii) prevents unnecessary escalation to L3 when the initial patch is already coherent. In addition, because Align now consolidates localized context and alignment cues into a compact scalar summary that the controller passes, it effectively serves as the lightweight state abstraction used by the controller. This helps stabilize ranking and gating decisions with minimal computational overhead. When textual artifacts are absent, the pipeline falls back to code-only signals, so dual-modality primarily influences ranking and early-exit decisions on the subset with available text.

Table 8 shows that *Synthesis* dominates the pipeline latency, accounting for approximately 69.0% of the agent-side runtime. This cost primarily arises from LLM decoding and its average invocation frequency of 1.8 calls per bug. *Refinement* contributes 10.1% despite being invoked only 0.35 times per bug on average, reflecting its selective activation for difficult cases that escalate to L3. The structure-aware modules, *Slicing* and *Alignment*, introduce relatively modest overheads of 5.0% and 3.7%, respectively, while contributing to the P@1 and CR improvements reported in Table 7. Lightweight confidence control through *ConfEvaluation* accounts for approximately 3.0%.

Based on the rounded values reported in the table, summing the products of Avg Time/Call and Avg Calls/Bug yields approximately 9.00 s of agent-side pipeline time. Adding the average test-harness overhead of approximately 3.2 s gives the overall average repair time of 12.2 s reported elsewhere. This runtime profile suggests that *Slicing* provides a favorable performance–

Table 8: Agent-wise runtime analysis excluding test-harness overhead.

Agent	Avg Time/Call (s)	Avg Calls/Bug	Time Contr. (%)
Localization	0.58	1.20	7.7
Slicing	0.82	0.55	5.0
Alignment	0.55	0.60	3.7
Synthesis	3.45	1.80	69.0
Refinement	2.60	0.35	10.1
ConfEvaluation	0.13	2.10	3.0
Aggregation	0.38	0.20	0.8
Scoping	0.28	0.20	0.6

Note: Time contribution is normalized over pipeline-only runtime, excluding test-harness overhead. Avg Calls/Bug counts all module invocations, including repeated calls to the same module.

cost trade-off, whereas *Refinement* concentrates additional computation on the more difficult subset of defects.

Answer to RQ3: FedDualFix's hierarchical design and specialized agents each contribute to repair performance. Layered control enhances robustness across varying bug complexities, while modules such as *Slicing* and *Refinement* yield the greatest gains. Other agents provide complementary benefits, such as alignment and summarization, that support both accuracy and coordination.

5.4. RQ4: What is the impact of a federated environment and confidence-guided agent scheduling on the performance and coordination efficiency of FedDualFix?

Approach. This question examines three aspects of FedDualFix: (i) the impact of federated training under project-wise non-IID client partitions, (ii) the sensitivity of FedDualFix to different client data partitions, and (iii) the effect of confidence-guided dynamic scheduling. Beyond the default project-level client split that mimics organizational boundaries, we quantify heterogeneity using the pairwise mean base-2 Jensen–Shannon divergence over client-wise project and patch-type distributions with Laplace smoothing ϵ=10^{-6}. The observed mean JS divergence is **0.41**, confirming substantial non-IID skew. Under this project-wise split, clients exhibit distinct code styles, test distributions, and defect patterns while sharing no projects.

To provide a federated neural reference, we include *Fed-CodeT5*. Fed-CodeT5 uses the same three-client project-wise partition, five communication

rounds ($R = 5$), and client-size-weighted FedAvg rule as FedDualFix, thereby keeping the federation-level protocol consistent across the two systems. Because Fed-CodeT5 updates CodeT5 parameters locally, whereas FedDualFix keeps the GPT-4o backend frozen and aggregates only lightweight scoring and calibration modules, their model-specific local optimization hyperparameters are selected separately on the same validation split and fixed before testing, as detailed in Section 4.5. We report repair effectiveness (CR, PR, and P@1), average runtime (Avg Time), and coordination overhead, measured by the mean number of invoked components per bug.

Federated Training Evaluation. We compare centralized and federated training strategies. The centralized setting pools all available training data, while the federated setting partitions Defects4J across three clients and shares only model updates rather than raw code or execution traces. To isolate the impact of federated training from the repair architecture, we compare centralized CodeT5 with Fed-CodeT5, and centralized FedDualFix with the federated version of FedDualFix.

Data Partitioning Evaluation. To further analyze the impact of client data partitioning, we evaluate FedDualFix under four representative partition settings: IID partitioning, Dirichlet-based mild non-IID partitioning, Dirichlet-based strong non-IID partitioning, and the default project-wise non-overlapping partition. The IID setting approximates balanced client distributions, while the Dirichlet settings control the degree of client heterogeneity using $\alpha \in \{0.3, 1.0\}$. The project-wise split represents the most realistic repository-level setting, where each client owns disjoint projects and therefore exhibits natural differences in coding styles, test distributions, and defect patterns. We report JS divergence, CR, P@1, Avg Time, and MIC to assess whether FedDualFix remains stable across varying levels of client skew.

Scheduling Evaluation. To evaluate confidence-guided control, we compare the full federated FedDualFix with a variant that disables adaptive scheduling. The unscheduled variant executes the complete agent sequence without early exits, while the scheduled variant accepts high-confidence patches at lower layers and escalates only uncertain cases. We track Avg Time and MIC alongside CR/PR/P@1.

Threshold Sensitivity. The default thresholds are obtained from the validation-guided grid search described in Section 4.5, rather than being manually chosen from the three configurations reported below. Table 11 presents three representative operating points from the evaluated range: the

aggressive setting emphasizes earlier exits, the default setting corresponds to the selected quality–efficiency trade-off, and the conservative setting escalates more cases to deeper repair layers. These configurations illustrate how the validated scheduler can be adapted to different operational preferences.

For the main federated-setting comparison, we compare five configurations:

- **CodeT5**: A centralized neural APR baseline using the native CodeT5 backbone.
- **Fed-CodeT5**: A federated version of CodeT5 trained under the same three-client partition and FedAvg protocol as FedDualFix.
- **Centralized FedDualFix**: A centralized variant of the multi-agent repair pipeline without federated client partitioning.
- **FedDualFix**: The full system with federated training and confidence-guided scheduling.
- **FedDualFix w/o scheduler**: A federated variant that disables adaptive early exits and executes a fixed agent sequence.

Result. Table 9 summarizes the results. Fed-CodeT5 performs slightly worse than centralized CodeT5 under the project-wise non-IID split, with CR decreasing from 39 to 36 and P@1 from 34.8% to 32.6%. This result suggests that directly applying FedAvg to CodeT5 does not improve its repair performance under the evaluated heterogeneous setting. FedDualFix achieves 58 correct repairs and 52.1% P@1, compared with 36 correct repairs and 32.6% P@1 for Fed-CodeT5. However, this cross-system comparison is not capacity-controlled: Fed-CodeT5 uses the 220M-parameter CodeT5 backbone, whereas FedDualFix relies on the substantially more capable proprietary GPT-4o backend. The observed difference, therefore, reflects both backbone capability and framework design and should not be interpreted as an isolated estimate of the contribution of the hierarchical multi-agent architecture. A capacity-controlled evaluation using CodeT5 within the FedDualFix framework is left for future work.

Compared with the centralized FedDualFix baseline, the full federated FedDualFix achieves comparable repair quality, with CR decreasing slightly from 60 to 58 and P@1 from 53.2% to 52.1%. This modest gap is consistent with the project-wise non-IID partition, where clients observe different

Table 9: Effect of federated training and confidence-guided scheduling on repair performance.

Setting	Training	CR ↑	PR ↑	P@1 ↑	Avg Time (s) ↓	MIC ↓
CodeT5	Centralized	39	63	34.8%	13.2	N/A
Fed-CodeT5	Federated	36	60	32.6%	13.9	N/A
Centralized FedDualFix	Centralized	60	85	53.2%	11.8	5.0
FedDualFix	Federated	**58**	**84**	**52.1%**	12.2	**4.1**
FedDualFix w/o scheduler	Federated	57	83	51.0%	16.6	6.0

Note: MIC denotes the average number of agent modules invoked per bug and is applicable only to multi-agent systems. CodeT5 and Fed-CodeT5 are included to compare centralized and federated neural APR under the same client partition. Because Fed-CodeT5 and FedDualFix use different backbones, the performance gap is not due to a capacity-controlled architectural comparison.

Table 10: Impact of data partitioning on FedDualFix under federated settings.

Partition Setting	**JS Div.**	**CR ↑**	**P@1 ↑**	**Avg Time (s) ↓**	**MIC ↓**
IID ($\alpha = \infty$)	0.08	60	53.4%	11.9	4.0
Dirichlet ($\alpha = 1.0$)	0.26	59	52.8%	12.0	4.0
Dirichlet ($\alpha = 0.3$)	0.37	57	51.2%	12.4	4.2
Project-wise	0.41	58	52.1%	12.2	4.1

code styles and defect distributions. Importantly, the federated setting preserves data locality and avoids sharing raw source code or execution traces, while maintaining repair effectiveness close to centralized training. Wilcoxon signed-rank tests on per-bug runtime and module-invocation records show that the reductions in Avg Time and MIC achieved by confidence-guided scheduling are statistically significant ($p < 0.05$ after Holm–Bonferroni correction).

On the control side, confidence-guided scheduling substantially improves execution efficiency. Compared with the unscheduled variant, FedDualFix reduces average inference time from 16.6s to 12.2s and lowers MIC from 6.0 to 4.1, while improving CR from 57 to 58 and P@1 from 51.0% to 52.1%. These gains come from early exits triggered by high-confidence patches, allowing the system to bypass expensive higher-layer modules when lower-layer reasoning is sufficient.

Table 10 further shows the influence of client data partitioning. As ex-

Table 11: Sensitivity analysis of scheduling thresholds on Defects4J.

Threshold setting	CR ↑	P@1 ↑	Avg Time (s) ↓	MIC ↓
Aggressive (θ_1=0.55, θ_2=0.35)	56	50.4%	10.8	3.6
Default (θ_1=0.60, θ_2=0.40)	58	52.1%	12.2	4.1
Conservative (θ_1=0.70, θ_2=0.50)	59	52.4%	14.7	5.2

pected, the IID setting obtains the best repair performance because each client observes a more balanced distribution of projects and defect patterns. When the client distribution becomes more heterogeneous, performance slightly decreases, especially under the stronger Dirichlet non-IID setting with $\alpha = 0.3$. Nevertheless, FedDualFix remains stable under the default project-wise split, where the JS divergence reaches 0.41. Compared with the IID setting, the project-wise split decreases CR only from 60 to 58 and P@1 from 53.4% to 52.1%, while maintaining similar Avg Time and MIC. These results indicate that FedDualFix is not overly sensitive to a specific client partition and can maintain repair effectiveness under realistic repository-level data skew.

Table 11 shows that the scheduling thresholds provide a tunable quality–cost trade-off. The aggressive setting reduces Avg Time from 12.2s to 10.8s and lowers MIC from 4.1 to 3.6 by accepting more patches at earlier layers, but it slightly decreases CR and P@1. The conservative setting increases the use of deeper reasoning layers, improving CR from 58 to 59 and slightly increasing P@1 from 52.1% to 52.4%, but at the cost of higher runtime and MIC. The default setting corresponds to the validation-selected trade-off used in the main experiments. These results indicate that θ_1 and θ_2 are configurable scheduling parameters rather than fixed universal constants. In practical CI deployments, they can be recalibrated according to latency budgets, defect criticality, and organizational risk tolerance.

Overall, federated training enables FedDualFix to maintain repair quality close to centralized training under project-wise non-IID partitions, while confidence-guided scheduling provides clear efficiency gains. Fed-CodeT5 serves as a federated neural reference and shows that directly applying FedAvg to CodeT5 does not improve its performance under the evaluated non-IID setting. However, because Fed-CodeT5 and FedDualFix use backbones with substantially different capacities, their performance gap should not be interpreted as a capacity-controlled estimate of the contribution of the hier-

archical multi-agent architecture. The threshold sensitivity analysis further shows that the scheduler can be adjusted to different quality–cost preferences rather than relying on a universal threshold setting.

Answer to RQ4: FedDualFix maintains repair quality close to centralized training while preserving data locality and shows limited performance variation across the evaluated partitions. Confidence-guided scheduling reduces runtime and agent usage, while configurable thresholds provide a tunable quality–cost trade-off.

6. Discussion

6.1. Architectural Implications and Design Insights

FedDualFix employs a three-layer hierarchical architecture in which execution depth adapts to perceived difficulty. As shown in Table 6, enabling deeper layers yields steady gains in correctness (CR) and Top-1 precision (P@1), while incurring only moderate runtime increases. L1 handles straightforward cases with rapid localization and single-step patching; L2 adds semantic slicing and description alignment for more ambiguous bugs; L3 supplements broader context and iterative refinement to recover complex or initially mislocalized defects. This escalation pattern is consistent with common debugging practices and helps explain why quality improves without incurring high costs.

The task-specialized agents make the pipeline easier to reason about and to schedule selectively. Among them, *Slicing* and *Refinement* have the most pronounced influence on CR and P@1 in our ablations, whereas *Alignment* provides auxiliary benefits for semantic coherence and cross-agent coordination. After merging lightweight state summarization into *Alignment*, the alignment module now also produces concise verbalizations of candidate edits, stabilizing downstream decision-making and reducing ambiguity across repair stages. A confidence-guided scheduler further tailors execution depth at runtime: compared with a static sequence, it reduces average latency by 26.5% and the number of agent invocations by 31.7% (Table 9), while preserving accuracy close to that of the whole pipeline. This indicates that early exits allow computation to be concentrated on the genuinely hard subset of bugs.

Dual-modality cues (*e.g.*, commit messages or comments) in our framework are instantiated through a *patch-level code-to-text alignment* mechanism. Concretely, *Alignment* verbalizes each candidate edit, evaluates its consistency with available textual artifacts, and outputs an alignment score D_{aln} alongside a short natural-language explanation. Removing *Alignment* reduces P@1 by 3.2 points while leaving CR nearly unchanged. Given the often brief textual artifacts in Defects4J, this suggests that alignment primarily helps discriminate among plausible candidates and improve first-choice selection, rather than increasing the number of ultimately validated repairs.

Operationally, the alignment signal plays three practical roles: it downranks intent-mismatched edits during CI/code review; it informs runtime triage by enabling confident early acceptance or targeted escalation; and it provides a concise human-readable trace $(\sigma(g), D_{\text{aln}})$ for transparency in pull requests and federated reporting. All textual artifacts are consumed locally during federated execution, and when the text is sparse or noisy, the controller defaults to code-side cues to ensure that CR is not adversely affected.

Overall, the architecture aligns naturally with federated deployment. Its hierarchical control structure and modular agent pipeline allow clients to activate only the relevant components under local constraints, improving scalability and privacy compliance. By avoiding raw data sharing and exchanging only lightweight parameter updates, FedDualFix supports decentralized repair under heterogeneous client distributions.

6.2. Failure Case Analysis

To better understand the remaining limitations of FedDualFix, we analyze two types of unsuccessful outcomes on Defects4J: (i) plausible-but-incorrect patches that pass the official tests but do not match the intended semantics of the developer fix, and (ii) bugs for which no plausible patch is generated within the five-attempt budget. The quantitative analysis below focuses on the first group, consisting of 26 cases derived from the difference between 84 plausible and 58 correct repairs. For each case, we inspect the localization output, selected slices, generated patch, alignment score, confidence trace, validation log, and terminal repair layer. Each case is assigned a primary failure category based on its dominant cause.

As shown in Table 12, incomplete semantic repair is the most frequent failure mode, accounting for 11 cases (42.3%), followed by incorrect localization with 7 cases (26.9%). Together, these two categories account for 69.2% of the plausible-but-incorrect repairs. In many of these cases, FedDualFix

Table 12: Distribution of plausible-but-incorrect repairs by failure mode and terminal layer.

Failure category	**Total**	**L1**	**L2**	**L3**
Incomplete semantic repair	11 (42.3%)	3	5	3
Incorrect localization	7 (26.9%)	2	3	2
Overfitting plausible patch	4 (15.4%)	2	2	0
Insufficient cross-file context	3 (11.5%)	1	1	1
API/type constraint mismatch	1 (3.8%)	0	1	0
Total	**26**	**8**	**12**	**6**

identifies a relevant region and produces a compilable, test-passing edit, but the patch either addresses only a local symptom or is generated from an incomplete repair location. Hidden behavioral conditions, long-range dependencies, and interactions among multiple methods are therefore the dominant sources of residual error.

The layer distribution shows that 8 incorrect patches are accepted at L1 and 12 at L2, while 6 remain incorrect after L3 refinement. The 20 cases terminating at L1 or L2 indicate that confidence-based acceptance can occasionally favor locally plausible patches before deeper semantic or contextual evidence is considered. However, the L3 failures show that the remaining errors cannot be attributed solely to aggressive early exits. These six cases are associated with incomplete semantic reasoning, incorrect localization, or insufficient cross-file context, suggesting that deeper refinement is still limited when the required behavioral or repository-level evidence is unavailable.

Overfitting plausible patches account for 4 cases, reflecting the limitations of test-suite-based validation, while cross-file context and API/type constraints account for the remaining 4 cases. Bugs for which no plausible patch is generated are analyzed separately because they have no terminal acceptance layer; their typical causes include patches that fail to apply, compile, or pass the official tests within the five-attempt budget. Overall, the analysis suggests two priorities for improvement: more conservative confidence calibration for locally plausible candidates and stronger repository-level retrieval, cross-file slicing, and semantic validation for cases that remain unresolved after deeper refinement.

Table 13: Sensitivity analysis of communication rounds under the project-wise non-IID split.

R	CR ↑	P@1 ↑	Avg. Time (s) ↓	MIC ↓
5	58	52.1%	12.2	4.1
10	58	52.3%	12.6	4.1
15	59	52.5%	12.9	4.2
20	59	52.4%	13.1	4.2

6.3. Parameter and Configuration Sensitivity

To assess the robustness and deployability of FedDualFix, we conduct sensitivity and operational analyses on several key configuration factors that govern its control behavior, communication cost, and runtime efficiency.

6.3.1. Communication Round Sensitivity

The main experiments use five communication rounds ($R = 5$) as a practical communication budget rather than as a claim that the federated optimization has reached theoretical convergence. In FedDualFix, the GPT-4o backend remains frozen, and only the lightweight scoring heads, alignment-calibration heads, and confidence-fusion modules participate in federated aggregation. Additional communication rounds, therefore, affect the cross-client calibration of confidence and alignment signals rather than the underlying patch-generation capability.

To examine whether a larger communication budget materially changes repair performance, we evaluate FedDualFix with $R \in \{5, 10, 15, 20\}$ under the same project-wise non-IID partition. Table 13 reports the results. Increasing R from 5 to 10 leaves CR unchanged at 58 and improves P@1 by only 0.2 percentage points. With $R = 15$ or $R = 20$, the largest improvement over the default setting is one additional correct repair and 0.4 percentage points in P@1. Meanwhile, Avg Time increases from 12.2 s at $R = 5$ to 12.9 s and 13.1 s at $R = 15$ and $R = 20$, respectively, and MIC increases slightly from 4.1 to 4.2.

These observations do not constitute a formal convergence proof for the lightweight trainable modules. Instead, they show that, in the evaluated three-client, project-wise non-IID setting, increasing the communication budget beyond five rounds yields limited empirical benefit relative to the additional coordination cost. We therefore retain $R = 5$ as the default practical budget and use larger values to characterize sensitivity to communication

Table 14: Federated calibration overhead of FedDualFix.

Metric	Value
Local calibration	
Trainable parameters / payload	0.65M / 2.60 MB
Local epochs / client / round	5
Mean local update / client / round	5.84 s
Peak additional GPU memory	32 MB
Communication and synchronization	
Bidirectional traffic / round	15.60 MB
Bidirectional traffic, $R = 5$	78.00 MB
Slowest-client gap / round	0.71 s
Parameter transfer / round	0.18 s
Server aggregation / round	0.05 s
Round wall-clock time	6.78 s
Five-round calibration	33.90 s (0.57 min)
Straggler sensitivity	
Injected delay	0.5 / 1.0 / 2.0 s
Round-time increase	0.49 / 0.98 / 1.96 s

rounds.

6.3.2. Federated Calibration Overhead

We evaluate federated calibration separately from repair-time LLM inference. Because the GPT-4o backend remains frozen, only the lightweight scoring, alignment-calibration, and confidence-fusion modules participate in local optimization and federated aggregation. Table 14 summarizes their training, communication, and synchronization costs under the default three-client setting.

The lightweight scoring, alignment-calibration, and confidence-fusion modules contain 0.65M trainable parameters in total. Each client performs five local epochs per communication round, requiring an average of 5.84 s for local optimization and approximately 32 MB of peak additional GPU memory. Over the default five communication rounds, this corresponds to 29.20 s of cumulative local-update time per client. Including parameter transfer, server aggregation, and synchronization waiting, the complete federated calibration procedure takes 33.90 s, or approximately 0.57 min. These measurements cover only the lightweight trainable components and exclude GPT-4o inference latency and memory consumption.

The communication payload is 2.60 MB per client update, resulting in

15.60 MB of bidirectional traffic per round and 78.00 MB over the default five-round protocol. In the controlled test environment, bidirectional parameter transfer requires 0.18 s per round, server-side aggregation requires 0.05 s, and the mean synchronization wait caused by differences in client completion time is 0.71 s. The resulting end-to-end round-completion time is 6.78 s. These measurements show that parameter-transfer latency accounts for only a small portion of the wall-clock cost, while local optimization and synchronization waiting contribute more substantially.

When controlled delays of 0.5 s, 1.0 s, and 2.0 s are introduced to the slowest client, the round-completion time increases by 0.49 s, 0.98 s, and 1.96 s, respectively. This near-linear increase reflects the synchronous FedAvg protocol, which cannot complete aggregation until all participating client updates have been received. Thus, a modest communication payload does not eliminate the effect of client stragglers. Because these measurements are obtained from a three-client simulation on a single workstation, the reported transfer latency should not be interpreted as wide-area network round-trip latency. Geographically distributed deployments may incur additional overhead because of network variability, resource heterogeneity, and more pronounced straggler effects.

6.3.3. Repair Attempt Cap

To avoid unnecessary inference overhead, FedDualFix limits the number of patch generation attempts per bug. An empirical analysis of 100 randomly selected bugs shows that 97.4% of successful repairs are achieved within the first five attempts. Increasing this cap beyond 5 yields negligible gains in correctness, while runtime rises linearly. We therefore set the default cap to 5, striking a balance between effectiveness and cost.

6.3.4. Agent Scheduling Strategy

We compare FedDualFix's confidence-guided scheduling with two static baselines: one that always executes agents up to L2, and another that fully exhausts all three layers regardless of patch confidence. While the L3-forced configuration slightly improves the correct repair count, it incurs an additional 40% in inference time. These results validate the effectiveness of FedDualFix's adaptive control mechanism, where computational effort scales with patch uncertainty and bug complexity—avoiding redundant agent activations on simple cases.

FedDualFix exhibits limited performance variation across the evaluated hyperparameter settings. Its default configuration—$(\theta_1, \theta_2) = (0.60, 0.40)$, a maximum of five repair attempts, and dynamic scheduling—provides a practical trade-off between repair correctness and computational efficiency. The sensitivity results further show that moderate changes to the scheduling thresholds produce only limited differences in repair performance and execution cost under the evaluated settings. Moreover, the use of interpretable and bounded hyperparameters, together with the validation-guided selection procedure described above, facilitates deployment in both high-throughput and resource-constrained environments.

Metaheuristic optimization also provides a promising direction for future configuration tuning in FedDualFix. Recent studies have demonstrated its effectiveness in global optimization, hyperparameter search, and parameter selection across domains [77, 78, 79, 80]. Inspired by these works, future extensions may use metaheuristic algorithms to optimize scheduling thresholds, agent invocation policies, patch-ranking weights, and communication-round configurations. We do not incorporate such optimizers in the current framework because they would introduce an additional search budget and validation protocol beyond the scope of this study.

6.4. Threats to Validity

To ensure a rigorous evaluation of FedDualFix, we analyze potential threats across three standard validity dimensions: *internal*, *external*, and *construct*. Beyond standard validity concerns, this subsection also explicitly discusses the methodological limitations of FedDualFix. These limitations include its dependence on the underlying LLM, bounded context selection, limited-scale federated simulation, absence of formal privacy guarantees, and reliance on test-suite-based repair validation. We discuss these issues critically to clarify the scope within which the reported results should be interpreted.

6.4.1. Internal Validity

While our ablation studies are designed to isolate the contribution of each agent module, residual interdependencies may still influence attribution. For example, removing *Slicing* may implicitly reduce the quality of contextual inputs for *Localization*, potentially underestimating Slice's full effect. To mitigate such confounds, we preserve the original control flow and

substitute ablated outputs with deterministic surrogates to avoid cascading degradation.

Another threat stems from LLMs' inherent nondeterminism. Within each controlled experimental setting, the LLM backend and decoding configuration are kept fixed; nevertheless, stochastic decoding may still cause slight variations in generated patches and downstream metrics. More importantly, FedDualFix still depends on the semantic reasoning capability of the underlying LLM. When the LLM fails to infer hidden invariants, implicit preconditions, or long-range behavioral dependencies, the multi-agent pipeline may still generate incomplete or overfitting patches. Confidence evaluation and alignment can reduce such cases, but they cannot fully replace stronger semantic oracles or formal verification. We address stochastic variation by fixing sampling parameters, using deterministic decoding when possible, and repeating evaluation runs to ensure that reported differences are robust to such randomness.

A further source of internal threat arises from the scheduler's confidence thresholds and escalation rules. These thresholds determine whether a defect triggers an early exit or deep analysis, which may, in turn, affect agent invocation patterns. Thresholds were tuned on a small validation subset, and although stable across runs, they may introduce minor bias in repair-path selection.

Finally, token truncation and prompt structuring may also impact performance. Since complex bugs may produce lengthy context, the framework applies a predefined truncation policy to stay within token limits. This bounded-context design is a practical efficiency choice, but it may omit cross-file dependencies, external API contracts, or class-level invariants that are necessary for certain repairs. While this policy is applied consistently across all methods, it remains possible that specific contextual cues are omitted during truncation, thereby affecting the LLM's behavior. We control for this by employing uniform prompt templates and fixed context-selection rules across all experimental conditions.

6.4.2. External Validity

Our primary evaluation is conducted on Defects4J, which contains real-world Java defects but does not capture the broader diversity of programming languages, development paradigms, or repository structures. Consequently, results derived from Java-centric projects may not fully generalize to ecosystems such as Python, C++, or multi-module enterprise software,

which have substantially different coding conventions. Although we include additional benchmarks (CodeFlaws, QuixBugs, and ConDefects) to provide supplementary evidence, their size and linguistic variety remain limited; thus, cross-dataset conclusions should be interpreted with caution.

The federated configuration used in this work partitions Defects4J into three clients, approximating a multi-project collaboration setting rather than a large-scale federated learning deployment. This is an important limitation of the current evaluation. Although the setting captures project-level data locality and non-IID repository distributions, it does not fully represent industrial federations with dozens or hundreds of organizations, highly imbalanced client sizes, unreliable clients, or asynchronous communication. Therefore, our results should be interpreted as evidence for project-level decentralized repair rather than as a complete validation of large-scale federated optimization. Exploring richer non-IID patterns, stronger organizational boundaries, larger client populations, and production-grade deployment constraints remains an important direction for future work. As reported in Section 6, increasing the communication budget beyond five rounds produces only limited changes in CR and P@1 under the evaluated project-wise non-IID setting.

Another external threat stems from the reliance on instruction-following LLMs with strong semantic reasoning capabilities. As shown in the backbone analysis, replacing GPT-4o with a lightweight model, such as CodeT5+, results in noticeably lower repair performance on Defects4J. This suggests that FedDualFix's multi-agent coordination, semantic slicing, and confidence evaluation can improve the use of a given backbone, but they cannot fully compensate for limited code reasoning or patch-generation capability in a weaker model. In scenarios where access to commercial LLMs is restricted or only smaller on-premise models are available, repair accuracy may therefore degrade. Thus, the choice of LLM backend represents an important deployment trade-off: stronger LLMs provide better repair accuracy, while lightweight models improve controllability, cost, and local deployability. Adapting FedDualFix to fine-tuned lightweight models or hybrid pipelines that combine local models with partial LLM support would improve portability across a wider range of deployment environments.

6.4.3. Construct Validity

We adopt standard APR metrics—Correct Repairs (CR), Plausible Repairs (PR), and Top-1 patch precision (P@1)—to measure repair success. Although widely used, these metrics rely on test-suite adequacy; incomplete

or under-specified tests may classify incorrect patches as plausible or reject semantically correct repairs that do not satisfy all assertions. As a result, the measured performance partially reflects benchmark characteristics rather than the intrinsic correctness of generated patches. While FedDualFix employs semantic alignment and confidence estimation to mitigate such inconsistencies, these mechanisms cannot fully overcome the limitations inherent in test-based evaluation.

Our multimodal reasoning also relies on pre-existing natural-language artifacts, such as commit messages and comments. These artifacts may be noisy, incomplete, or only loosely related to the underlying defect, thereby affecting the reliability of alignment signals. Moreover, interactive or dialog-based bug reporting is not modeled, limiting the ecological validity of the textual inputs relative to real developer workflows.

Another construct-related consideration concerns the architectural interpretation of FedDualFix. Although the framework includes several lightweight modules, its improvements over agentless pipelines do not stem from the number of components. Instead, the core value lies in the hierarchical control mechanism and the dual-signal feedback loop that integrates structural diagnostics with semantic alignment. These features enable calibrated escalation, early exits, and interpretable decision traces—benefits that cannot be reproduced by flat or agentless designs. Thus, the conceptual contribution of FedDualFix is rooted in its coordination and feedback principles rather than in the mere presence of multiple agent modules.

Finally, while the framework preserves data locality in the federated setting, its privacy benefit is architectural rather than formally guaranteed. We do not quantify privacy leakage or formally evaluate privacy guarantees, such as differential privacy, secure aggregation, or resistance to gradient-inversion attacks. Moreover, when commercial LLM APIs are used, source code snippets included in prompts may still be exposed to the LLM provider. Thus, strict end-to-end privacy is not guaranteed unless organizations deploy a local LLM backend or combine FedDualFix with stronger privacy-preserving mechanisms. Federated learning in FedDualFix therefore provides privacy primarily across clients, not against the LLM provider.

7. Conclusion and Future Work

This paper presents FedDualFix, a federated multi-agent framework for dual-modality automated program repair. Motivated by the need for repair

in privacy-sensitive and data-locality-constrained development settings, FedDualFix introduces a hierarchical repair architecture composed of specialized agents and a confidence-guided scheduling strategy. By coordinating localization, patch generation, alignment, and refinement across three adaptive layers, the system enables efficient, modular, and scalable repair workflows.

On Defects4J, FedDualFix delivers competitive repair performance under the evaluated federated setting. With GPT-4o, it attains CR=58, PR=84, and P@1=52.1%. Compared to a centralized baseline, the gaps are modest (approximately 3–4% on CR/PR and 1.1 points on P@1), suggesting that federated training preserves most of the repair quality under our three-client partition. Confidence-guided scheduling reduces the average runtime from 16.6s to 12.2s (26.5%) and lowers the mean number of components invoked from 6.0 to 4.1 (31.7%) relative to a static sequence, while maintaining similar accuracy. Ablation studies indicate that *Slicing* and *Refinement* contribute most to correctness and Top-1 ranking, and that the enhanced *Alignment*—which now also distills key intermediate signals into a compact, alignment-oriented representation—improves first-choice quality by approximately 3.2 points in P@1.

To further advance this line of research, we plan to extend the evaluation to larger multilingual repair benchmarks and larger-scale federated settings with more clients, asynchronous participation, and stronger privacy constraints. Interactive modality integration—such as dialog-based bug reporting—will be explored to strengthen semantic alignment. We also aim to investigate learning-based scheduling strategies and federated personalization mechanisms to improve adaptability and repair relevance across diverse client environments.

Overall, FedDualFix offers a flexible and extensible foundation for multi-agent, multimodal, and privacy-conscious program repair. We hope this work encourages further research into adaptive, decentralized, and semantically grounded repair systems.

CRediT Authorship Contribution Statement

Xiaolin Ju: Conceptualization, Methodology, Writing -review & editing, and Supervision. **Qingyun Liu:** Data curation, Software, Validation, Conceptualization, Methodology, Writing -review & editing. **Chang Li:** Supervision, Writing - review & editing. **Haochen Wang:** Conceptualization, Methodology, Writing -review & editing. **Heling Cao:** Conceptualization,

Methodology, Writing -review & editing. **Xiang Chen:** Conceptualization, Methodology, and Supervision.

Acknowledgement

This work is partially supported by the National Key R&D Plan of China (Grant No. 2024YFE0202700) and the National Natural Science Foundation of China (Grant No. 61872263).

References

[1] N. E. Fenton, M. Neil, A critique of software defect prediction models, IEEE Transactions on Software Engineering 25 (5) (2002) 675–689.

[2] C. E. McDowell, D. P. Helmbold, Debugging concurrent programs, ACM Computing Surveys (CSUR) 21 (4) (1989) 593–622.

[3] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: A generic method for automatic software repair, IEEE Transactions on Software Engineering 38 (1) (2011) 54–72.

[4] Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang, The strength of random search on automated program repair, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 254–265.

[5] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 298–312.

[6] X. Kong, L. Zhang, W. E. Wong, B. Li, The impacts of techniques, programs and tests on automated program repair: An empirical study, Journal of Systems and Software 137 (2018) 480–496.

[7] J. He, C. Treude, D. Lo, LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead, ACM Transactions on Software Engineering and Methodology 34 (5) (2025) 1–30.

[8] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, H. Chen, From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future, arXiv preprint arXiv:2408.02479 (2024).

[9] C. S. Xia, Y. Deng, S. Dunn, L. Zhang, Agentless: Demystifying LLM-based software engineering agents, arXiv preprint arXiv:2407.01489 (2024).

[10] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, Z. Chen, A critical review of large language model on software engineering: An example from ChatGPT and automated program repair, arXiv preprint arXiv:2310.08879 (2023).

[11] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, Z. Chen, A systematic literature review on large language models for automated program repair, arXiv preprint arXiv:2405.01466 (2024).

[12] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, X. Yang, ThinkRepair: Self-directed automated program repair, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1274–1286.

[13] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, F. Liu, Enhancing automated program repair with solution design, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1706–1718.

[14] M. Lajkó, V. Csuvik, T. Gyimothy, L. Vidács, Automated program repair with the GPT family, including GPT-2, GPT-3 and CodeX, in: Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair, 2024, pp. 34–41.

[15] S. Kang, G. An, S. Yoo, A quantitative and qualitative evaluation of LLM-based explainable fault localization, Proceedings of the ACM on Software Engineering 1 (FSE) (2024) 1424–1446.

[16] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, D. Lo, FlexFL: Flexible and effective fault localization with open-source large language models, IEEE Transactions on Software Engineering (2025).

[17] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, Y. Liu, Large language models in fault localisation, arXiv preprint arXiv:2308.15276 (2023).

[18] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, X. Mao, SoapFL: A standard operating procedure for LLM-based method-level fault localization, IEEE Transactions on Software Engineering 51 (4) (2025) 1173–1187.

[19] S. Bin Murtaza, A. Mccoy, Z. Ren, A. Murphy, W. Banzhaf, LLM fault localisation within evolutionary computation based automated program repair, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2024, pp. 1824–1829.

[20] C. Lee, C. S. Xia, L. Yang, J.-t. Huang, Z. Zhu, L. Zhang, M. R. Lyu, Unidebugger: Hierarchical multi-agent framework for unified software debugging, in: Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, 2025, pp. 18248–18277.

[21] N. Akbarpour, M. S. Benis, F. H. Fard, A. Ouni, M. A. Saied, Collaborative agents for automated program repair in ruby, arXiv preprint arXiv:2511.03925 (2025).

[22] W. Luo, J. W. Keung, B. Yang, J. Klein, T. F. Bissyande, H. Tian, B. Le, Unlocking llm repair capabilities in low-resource programming languages through cross-language translation and multi-agent refinement, arXiv preprint arXiv:2503.22512 (2025).

[23] Y.-H. Jia, Y. Mei, M. Zhang, Confidence-based ant colony optimization for capacitated electric vehicle routing problem with comparison of different encoding schemes, IEEE Transactions on Evolutionary Computation 26 (6) (2022) 1394–1408.

[24] H. Rahmath P, V. Srivastava, K. Chaurasia, R. G. Pacheco, R. S. Couto, Early-exit deep neural network-a comprehensive survey, ACM Computing Surveys 57 (3) (2024) 1–37.

[25] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, D. Bacon, Federated learning: Strategies for improving communication efficiency, arXiv preprint arXiv:1610.05492 (2016).

[26] Y. Yang, X. Hu, Z. Gao, J. Chen, C. Ni, X. Xia, D. Lo, Federated learning for software engineering: A case study of code clone detection and defect prediction, IEEE Transactions on Software Engineering 50 (2) (2024) 296–321.

[27] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, M. Monperrus, SequenceR: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering 47 (9) (2019) 1943–1959.

[28] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, D. Poshyvanyk, An empirical study on learning bug-fixing patches in the wild via neural machine translation, ACM Transactions on Software Engineering and Methodology (TOSEM) 28 (4) (2019) 1–29.

[29] K. Huang, S. Yang, H. Sun, C. Sun, X. Li, Y. Zhang, Repairing security vulnerabilities using pre-trained programming language models, in: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), IEEE, 2022, pp. 111–116.

[30] R. Gupta, S. Pal, A. Kanade, S. Shevade, DeepFix: Fixing common C language errors by deep learning, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 31, 2017.

[31] N. Jiang, K. Liu, T. Lutellier, L. Tan, Impact of code language models on automated program repair, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 1430–1442.

[32] Q. Zhang, C. Fang, Y. Ma, W. Sun, Z. Chen, A survey of learning-based automated program repair, ACM Transactions on Software Engineering and Methodology 33 (2) (2023) 1–69.

[33] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, L. Zhang, A syntax-guided edit decoder for neural program repair, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 341–353.

[34] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).

[35] W. Wang, Y. Wang, S. Joty, S. C. Hoi, RAP-Gen: Retrieval-augmented patch generation with CodeT5 for automatic program repair, in: Proceedings of the 31st ACM Joint European Software Engineering Confer-

ence and Symposium on the Foundations of Software Engineering, 2023, pp. 146–158.

[36] R. Just, D. Jalali, M. D. Ernst, Defects4J: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 437–440.

[37] D. Lin, J. Koppel, A. Chen, A. Solar-Lezama, QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge, in: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2017, pp. 55–56.

[38] W. Yang, H. Wang, Z. Liu, X. Li, Y. Yan, S. Wang, Y. Gu, M. Yu, Z. Liu, G. Yu, Enhancing the code debugging ability of LLMs via communicative agent based data refinement, arXiv preprint arXiv:2408.05006 (2024).

[39] P. Xue, L. Wu, Z. Yang, Z. Yu, Z. Jin, G. Li, Y. Xiao, S. Liu, X. Li, H. Lin, et al., Exploring and lifting the robustness of LLM-powered automated program repair with metamorphic testing, arXiv preprint arXiv:2410.07516 (2024).

[40] K. C. Youm, J. Ahn, J. Kim, E. Lee, Bug localization based on code change histories and bug reports, in: 2015 Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2015, pp. 190–197.

[41] W. Zhang, Z. Li, Q. Wang, J. Li, FineLocator: A novel approach to method-level fine-grained bug localization by query expansion, Information and Software Technology 110 (2019) 121–135.

[42] S. Tsumita, S. Hayashi, S. Amasaki, Large-Scale evaluation of method-level bug localization with FinerBench4BL, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2023, pp. 815–824.

[43] R. Abreu, P. Zoeteweij, A. J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), IEEE, 2007, pp. 89–98.

[44] W. E. Wong, V. Debroy, R. Gao, Y. Li, The DStar method for effective software fault localization, IEEE Transactions on Reliability 63 (1) (2013) 290–308.

[45] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, L. Zhang, Can automated program repair refine fault localization? a unified debugging approach, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 75–87.

[46] A. Schröter, N. Bettenburg, R. Premraj, Do stack traces help developers fix bugs?, in: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), IEEE, 2010, pp. 118–121.

[47] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y.-G. Guéhéneuc, Is it a bug or an enhancement? A text-based approach to classify change requests, in: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, 2008, pp. 304–318.

[48] C. Treude, M. P. Robillard, Augmenting API documentation with insights from stack overflow, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 392–403.

[49] M. Seo, W. Choi, M. You, S. Shin, Autopatch: Multi-agent framework for patching real-world cve vulnerabilities, arXiv preprint arXiv:2505.04195 (2025).

[50] J. Li, Z. Chen, Y. Su, M. R. Lyu, VulKey: Automated vulnerability repair guided by domain-specific repair patterns, arXiv preprint arXiv:2605.01769 (2026).

[51] R. C. Cardoso, A. Ferrando, A review of agent-based programming for multi-agent systems, Computers 10 (2) (2021) 16.

[52] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, Y. Lou, Large language model-based agents for software engineering: A survey, arXiv preprint arXiv:2409.02977 (2024).

[53] I. Bouzenia, P. Devanbu, M. Pradel, RepairAgent: An autonomous, LLM-based agent for program repair, arXiv preprint arXiv:2403.17134 (2024).

[54] E. Zhang, S. Sun, Y. Xing, K. Sun, Poster: Repairing bugs with the introduction of new variables: A multi-agent large language model, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, 2024, pp. 4961–4963.

[55] R. Karanjai, S. Blackshear, L. Xu, W. Shi, A multi-agent framework for automated vulnerability detection and repair in solidity and move smart contracts, arXiv preprint arXiv:2502.18515 (2025).

[56] A. Pabba, A. Mathai, A. Chakraborty, B. Ray, Semagent: A semantics aware program repair agent, arXiv preprint arXiv:2506.16650 (2025).

[57] H. Yamamoto, D. Wang, G. K. Rajbahadur, M. Kondo, Y. Kamei, N. Ubayashi, Towards privacy preserving cross project defect prediction with federated learning, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2023, pp. 485–496.

[58] J. Kumar, S. Chimalakonda, Code summarization without direct access to code-towards exploring federated LLMs for software engineering, in: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, 2024, pp. 100–109.

[59] C. Zhang, T. Yu, B. Liu, Y. Xin, Vulnerability detection based on federated learning, Information and Software Technology 167 (2024) 107371.

[60] B. McMahan, E. Moore, D. Ramage, S. Hampson, B. A. y Arcas, Communication-efficient learning of deep networks from decentralized data, in: Artificial intelligence and statistics, Pmlr, 2017, pp. 1273–1282.

[61] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, V. Smith, Federated optimization in heterogeneous networks, Proceedings of Machine learning and systems 2 (2020) 429–450.

[62] D. A. E. Acar, Y. Zhao, R. M. Navarro, M. Mattina, P. N. Whatmough, V. Saligrama, Federated learning based on dynamic regularization, arXiv preprint arXiv:2111.04263 (2021).

[63] X. Yang, W. Huang, M. Ye, Fedas: Bridging inconsistency in personalized federated learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2024, pp. 11986–11995.

[64] Y. Chen, W. Huang, M. Ye, Fair federated learning under domain skew with local consistency and domain diversity, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2024, pp. 12077–12086.

[65] J. Gawlikowski, S. Saha, A. Kruspe, X. X. Zhu, An advanced dirichlet prior network for out-of-distribution detection in remote sensing, IEEE Transactions on Geoscience and Remote Sensing 60 (2022) 1–19.

[66] K. Liu, A. Koyuncu, D. Kim, T. F. Bissyandé, TBar: Revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 31–42.

[67] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, pp. 298–309.

[68] C. S. Xia, L. Zhang, Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 819–831.

[69] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, L. Tan, CoCoNuT: Combining context-aware neural translation models using ensemble for program repair, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 101–114.

[70] H. Ye, M. Monperrus, Iter: Iterative neural repair for multi-location patches, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.

[71] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, et al., A comprehensive capability analysis of GPT-3 and GPT-3.5 series models, arXiv preprint arXiv:2303.10420 (2023).

[72] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., GPT-4 technical report, arXiv preprint arXiv:2303.08774 (2023).

[73] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al., GPT-4o system card, arXiv preprint arXiv:2410.21276 (2024).

[74] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al., StarCoder 2 and the stack v2: The next generation, arXiv preprint arXiv:2402.19173 (2024).

[75] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al., Code Llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).

[76] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, S. C. Hoi, CodeT5+: Open code large language models for code understanding and generation, arXiv preprint arXiv:2305.07922 (2023).

[77] H. Akbulut, A modified starfish optimization algorithm (m-sfoa) for global optimization problems and its application to heart disease risk prediction, Expert Systems with Applications (2026) 131088.

[78] H. Akbulut, S. Atasever, E. Sıramkaya, Machine learning-based orange quality classification: A hyperparameter optimization approach through puma optimizer, in: 2025 7th International Congress on Human-Computer Interaction, Optimization and Robotic Applications (ICHORA), IEEE, 2025, pp. 1–5.

[79] H. Akbulut, Meta-heuristic optimization for optimal block size in multi-focus image fusion: a comprehensive comparative study, The Visual Computer 41 (13) (2025) 11025–11051.

[80] H. Akbulut, V. Aslantas, An optimal multi-exposure image fusion using genetic algorithm, Arabian Journal for Science and Engineering (2025) 1–25.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: