Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Semantic Spectrum: Fault Localization via Method Behavioral Divergence

Tu Peng

pengtucdtu@gmail.com

Chengdu Technological University

Xianju Zheng

Chengdu Technological University

Yin Kuang

Chengdu Normal University

Abdelmounaim Mekaoui

Chengdu Technological University

Yazhi Yang

Chengdu Technological University

Ling Xiong

Beijing Language and Culture University

Research Article

Keywords: fault localization, semantic spectrum, method output, runtime monitoring, Defects4J

Posted Date: August 20th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10584016/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Semantic Spectrum: Fault Localization via Method Behavioral Divergence

Abstract

Automated fault localization (FL) aims to reduce the effort required for software debugging. Many widely used dynamic FL techniques, particularly spectrum-based fault localization (SBFL), rely primarily on coverage spectra that record which statements or methods are executed by passing and failing tests, and may therefore struggle to distinguish program entities with similar coverage profiles. This paper proposes Semantic Spectrum-based Fault Localization (SSFL), a method-level approach that exploits behavioral information contained in runtime output values. SSFL transforms heterogeneous method outputs into a unified token representation and constructs semantic spectra from token-frequency distributions observed in passing and failing tests. Two complementary measures—a Bidirectional Normalized Euclidean Distance and a Weighted Frequency Difference—quantify behavioral divergence and support suspicious-method ranking through an output-profile-aware adaptive combination strategy. We evaluate SSFL on 357 real faults from five Defects4J projects. SSFL reports mean first rank (MFR) values that are 60–90% lower than those of the best-performing traditional SBFL formulae. Compared with representative learning-based and large language model (LLM)-based approaches, SSFL achieves the highest overall Top-1, Top-3, and Top-5 localization counts, identifying 172, 242, and 262 faults, respectively, while requiring no model training or online LLM reasoning during ranking. The results indicate that runtime output-value distributions provide diagnostic evidence complementary to structural coverage and source-code reasoning.

Keywords: fault localization, semantic spectrum, method output, runtime monitoring, Defects4J

1 Introduction

Software reliability is a critical concern in modern software development. According to a comprehensive survey Wong et al. (2016), software failures remain widespread and costly, with debugging activities consuming a substantial portion of development budgets. Recent large-scale service incidents further illustrate the importance of rapidly identifying faulty runtime behavior. On 12 June 2025, an invalid automated quota-policy update triggered a previously unexercised code path in Google Cloud's Service

Control system. Inadequate error handling caused a null-pointer failure and repeated process crashes, disrupting multiple Google Cloud and Workspace services worldwide. Google reported that its engineers began triage within two minutes and identified the root cause within ten minutes Google Cloud (2025). This incident demonstrates how a localized software defect, when activated by unexpected runtime data, can propagate rapidly through large-scale software infrastructure. Consequently, rapidly and accurately localizing software faults is important for improving both software quality and development efficiency.

To automate this diagnostic process, fault localization techniques have been widely studied to identify program entities, such as statements or methods, that may contain defects Wong et al. (2023). Among existing techniques, **spectrum-based fault localization (SBFL)** has become one of the most widely studied fault localization paradigms Wong et al. (2016). Classic SBFL formulae such as Ochiai Abreu et al. (2006), DStar Wong et al. (2014), and GP13 Yoo (2012) statistically correlate program element execution with test outcomes, providing a ranked list of suspicious locations.

However, despite its popularity, SBFL suffers from a **fundamental weakness**. It relies exclusively on **structural coverage profiles**—essentially recording *which* statements were executed—while remaining oblivious to *what* those statements actually produced. This reliance on binary execution traces creates an **information bottleneck**: when a defective method and a correct method are executed by the same set of test cases, they exhibit a "coverage tie," and a coverage-only SBFL formula assigns identical evidence to them Xu et al. (2011). A related but distinct phenomenon is coincidental correctness, in which a passing test executes defective code without exposing the failure. Empirical studies have shown that in the Defects4J benchmark, 54–56% of faulty methods share the same suspiciousness score with at least one other method Sarhan and Beszédes (2022), preventing the suspiciousness score alone from prioritizing the true defect among tied candidates. To address the tie problem, Yan et al. (2023) proposed weighting suspicious values using fault propagation context, demonstrating that context-aware approaches can effectively break ties and improve localization accuracy. Nevertheless, the fundamental limitation remains: structure-based rankers that rely solely on coverage information cannot distinguish program entities with identical execution profiles.

Recent years have seen growing interest in Deep Learning (DL)-based fault localization. These approaches leverage additional software information beyond the coverage data, such as structural relationships and code modification history Chen et al. (2024), execution contexts Zhang et al. (2023), deep semantic features Meng et al. (2022), probabilistic program semantics Zeng et al. (2022), or large language model (LLM)-based combinations of multiple diagnostic sources Xu et al. (2025), to refine suspiciousness calculations Zhang et al. (2017),Zhang et al. (2019). While these methods demonstrate notable improvements over classical formulae, they still face significant challenges: (1) many retain coverage as a core input and thus remain vulnerable to the identical coverage problem; (2) learning-based methods require training data and computational resources, while hybrid methods may depend on additional artifacts, such as code change histories, test contexts, or bug reports, that are costly or unavailable in some industrial scenarios. Consequently, defects that manifest as

incorrect data values without altering the control flow path can remain elusive when runtime values are not modeled explicitly.

To overcome this limitation, we propose shifting fault localization from a **path-centric** perspective to a **value-centric** one. We observe that even when a defective method shares an identical execution path with a non-defective one, its runtime behavior—specifically, the distribution of its output values—often diverges significantly in failing executions. This **behavioral divergence** is a rich, yet underutilized source of diagnostic information.

Driven by this insight, we propose a novel framework: **Semantic Spectrum-based Fault Localization (SSFL)**. Unlike traditional approaches, we explicitly model the output semantics of the methods. After applying a fixed tokenization procedure, SSFL aggregates runtime output values into **frequency matrices**, converting the fault localization problem into a geometric one: quantifying the distance between the "passing" and "failing" semantic distributions. We introduce a **Bidirectional Normalized Euclidean Distance** to measure the raw distributional difference, and a **Weighted Frequency Difference** to capture changes in token-level weighted frequencies, supporting the ranking of suspicious methods when structural coverage provides limited discriminative information.

The principal contributions of this paper are summarized as follows:

- **Perspective Shift:** We propose shifting the fault localization perspective from structural coverage to output semantics, effectively addressing the limitations of coverage indistinguishability and coverage ties.
- **Frequency-Based Semantic Spectrum:** We formulate per-test output behavior as token-frequency mappings and aggregate them into passing and failing semantic-spectrum matrices, preserving how frequently each output token occurs across test executions.
- **Behavioral-Divergence Metrics:** We develop a matrix-based mathematical model and introduce a Bidirectional Normalized Euclidean Distance combined with a Weighted Frequency Difference to quantify the semantic divergence between passing and failing output distributions.
- **Empirical Validation:** Extensive evaluation on the *Defects4J* benchmark demonstrates that SSFL reduces the Mean First Rank (MFR) by 60%–90% over classical SBFL formulae and achieves the highest overall Top-1, Top-3, and Top-5 localization counts among the representative learning-based and large language model (LLM)-based techniques considered, without requiring model training or online LLM reasoning during ranking.

2 Materials and Methods

In this section, we present the methodology of our proposed framework, **Semantic Spectrum-based Fault Localization (SSFL)**.

The core innovation of our approach is the introduction of the **Semantic Spectrum**—a novel representation that captures the runtime behavioral semantics of program entities. Unlike traditional **Structural Spectra** used in SBFL (which record *whether* code elements are executed, providing only a hit-or-miss execution profile),

the **Semantic Spectrum** records *what values* are produced during execution (providing a rich, value-centric profile). By quantifying the behavioral divergence between the Semantic Spectra of passing and failing tests, our approach can rank program entities that are indistinguishable from their structural spectra alone.

We begin by providing a motivating example to illustrate the fundamental limitations of Structural Spectra in handling "coverage ties" and how Semantic Spectra resolve this tie problem. Then, we present an overview of the SSFL framework, detailing the workflow from semantic capture to spectrum construction. Finally, in the subsequent subsections, we introduce the mathematical model for quantifying behavioral divergence, including the **Bidirectional Normalized Euclidean Distance** and the **Weighted Frequency Difference**.

2.1 Motivating Example: Structural vs. Semantic Spectra

To expose the fundamental limitation of structural spectra and to show how semantic spectra overcome it, we draw on a real defect from the Defects4J benchmark: bug Closure v131, which resides in the JavaScript compiler's logic for telling identifiers apart from keywords. Two closely related methods, `isJSIdentifier` and `isKeyword`, serve similar purposes and are invoked in similar contexts. In the faulty version, both are exercised by exactly the same set of tests—one passing test and two failing tests—yet the defect resides solely in `isJSIdentifier`.

Where the structural spectrum fails. Scoring these two methods with the classic Ochiai formula yields identical suspiciousness values. This is the textbook "coverage tie": from a purely structural standpoint the two methods are mathematically indistinguishable, and a developer following the ranked list has no basis for choosing between them.

Reading behavior from the outputs. Although the structural spectrum treats the two methods alike, their runtime outputs tell a very different story. Table 1 reports, for each test, how many times each method returns `True` (denoted `T`) and `False` (denoted `F`).

Table 1 Output statistics of `isJSIdentifier` and `isKeyword` over the three test cases of Closure v131.

Test	Result	`isJSIdentifier`	`isKeyword`
t_1	Pass	`T` × 9	`F` × 9
t_2	Fail	`F` × 16, `T` × 3	`F` × 1, `T` × 2
t_3	Fail	`F` × 4, `T` × 5	`F` × 5

The two methods exhibit markedly different behavioral profiles. For `isJSIdentifier`, the passing test produces a stable run of nine `T` values, whereas the failing tests are dominated by `F` (16 and 4 occurrences) with `T` collapsing to only 3 and 5—the method's prevailing return value flips from `T` under success to `F` under failure. `isKeyword`, by contrast, barely shifts: it returns `F` nine times on the passing

test, and although a handful of `T` values surface under failure, `F` remains its dominant output throughout.

Building the semantic spectrum. We encode these output counts as frequency matrices. Each row corresponds to an output token (here `T` or `F`), each column j to a specific number of occurrences within a single test, and the cell $M_{\tau,j}$ records how many tests observe token τ exactly j times. The matrix built over the passing tests is M^p and the one built over the failing tests is M^f; Table 2 lists both for each method.

Table 2 Semantic spectrum matrices of the two methods.

`isJSIdentifier` **(defective)**

(a) M^p **(passing)**

Token	$j = 9$
`T`	1

(b) M^f **(failing)**

Token	$j = 3$	$j = 4$	$j = 5$	$j = 16$
`T`	1	0	1	0
`F`	0	1	0	1

`isKeyword` **(non-defective)**

(c) M^p **(passing)**

Token	$j = 9$
`F`	1

(d) M^f **(failing)**

Token	$j = 1$	$j = 2$	$j = 5$
`F`	1	0	1
`T`	0	1	0

Quantifying the divergence. The pair (M^p, M^f) characterizes how a method's output distribution shifts between successful and failing runs. When the two distributions stay close, the method behaves consistently and is an unlikely fault source; when they diverge sharply—and especially when the relative prominence of a token reverses between the two settings—the method's behavior has become anomalous and is more likely implicated in the failure.

To capture this shift we use the **Weighted Frequency Difference** (ΔW_{freq}). For an output token τ, its weighted frequency is calculated from its total number of occurrences in the test set being considered, multiplied by its token weight w_τ, and divided by the sum of these weighted occurrence counts over all output tokens. The values calculated from passing and failing tests are denoted by $W^p_{freq}(\tau)$ and $W^f_{freq}(\tau)$, respectively.

ΔW_{freq} measures the aggregate difference between these two values over all output tokens. Intuitively, a larger ΔW_{freq} indicates a greater change in the weighted frequencies and therefore provides stronger evidence of failure-associated behavioral change. Its formal definition is given in Section 2.4.3.

In this Boolean example, $w_T = w_F = 1$. Using the fixed token order (T, F), the weighted frequencies of `isJSIdentifier` under passing and failing tests are

$$\left(W^p_{freq}(T), W^p_{freq}(F)\right) = \left(\frac{9}{9}, \frac{0}{9}\right) = (1, 0),$$

and

$$\left(W^f_{freq}(T), W^f_{freq}(F)\right) = \left(\frac{3+5}{3+5+16+4}, \frac{16+4}{3+5+16+4}\right) = \left(\frac{8}{28}, \frac{20}{28}\right).$$

The corresponding difference is therefore

$$\Delta W_{freq} = \sqrt{\left(1 - \frac{8}{28}\right)^2 + \left(0 - \frac{20}{28}\right)^2} \approx 1.0102.$$

For `isKeyword`, the two failing tests produce two occurrences of T and six occurrences of F, giving

$$\left(W^p_{freq}(T), W^p_{freq}(F)\right) = \left(\frac{0}{9}, \frac{9}{9}\right) = (0, 1),$$

and

$$\left(W^f_{freq}(T), W^f_{freq}(F)\right) = \left(\frac{2}{2+1+5}, \frac{1+5}{2+1+5}\right) = \left(\frac{2}{8}, \frac{6}{8}\right),$$

and hence

$$\Delta W_{freq} = \sqrt{\left(0 - \frac{2}{8}\right)^2 + \left(1 - \frac{6}{8}\right)^2} \approx 0.3536.$$

Thus, for `isJSIdentifier`, the weighted frequency of T decreases from 1 to 8/28, while that of F increases from 0 to 20/28, reversing the dominant output token. For `isKeyword`, F retains the larger weighted frequency. The approximately 2.86-fold larger ΔW_{freq} therefore exposes the stronger behavioral shift of the defective method. Where Ochiai stalls on a coverage tie, the semantic spectrum recovers the difference hidden in the output distributions and ranks the faulty method ahead of its sibling.

2.2 Approach Overview

The premise of our work is that faults may alter observable program states even when a faulty method shares the same execution path as a correct one. Such faults can generate output patterns that differ between passing and failing executions. Based on this observation, we shift the fault localization focus from the **Structural Spectrum** (control flow) to the **Semantic Spectrum** (data values).

We hypothesize that defective methods exhibit a measurable **behavioral divergence** in their output distributions—a unique signature that separates them from correct methods. To capture this signature, our framework operates in three sequential phases:

1. **Candidate Filtering:** Although we move beyond structural coverage, we recognize its utility as a coarse-grained filter. To narrow the search space, we first identify the set of potentially suspicious methods – those executed by at least one failing test case –using standard structural coverage analysis. Methods that are never invoked during any failure test are excluded from further analysis. This step ensures computational efficiency by focusing only on relevant entities.

2. **Semantic Spectrum Construction:** For each candidate method, we employ BytemanRed Hat (2024), a Java bytecode instrumentation agent, to intercept its runtime outputs. The raw output values are first transformed into a unified token sequence through a tokenization process, which maps different return types (boolean, numerical, string, etc.) into a common token vocabulary $\mathcal{TK}$. These tokenized outputs are then aggregated into **Frequency Matrices**:

   - We construct a *Passing Semantic Spectrum* (M^p) from passing test cases.
   - We construct a *Failing Semantic Spectrum* (M^f) from failing test cases.

   In this representation, the matrix rows correspond to the tokens produced, and the columns represent their frequency of occurrence. The cell $M_{\tau,j}$ represents the number of test cases in which the token τ appears exactly j times. Together, M^p and M^f form the dual semantic profile of the method, capturing its behavioral characteristics under normal and faulty execution, respectively.
3. **Divergence Quantification:** Finally, we calculate the suspiciousness of each method by measuring the divergence between M^p and M^f. We first compute a bidirectional normalized Euclidean distance ($Susp_{norm}$) over the two matrices, which quantifies the raw distributional difference. However, when M^p and M^f have largely disjoint token-frequency distributions, this metric saturates near 1.0 for many methods, reducing its discriminative power. To address this, we introduce a complementary metric, the weighted frequency difference (ΔW_{freq}), which captures changes in token-level weighted frequencies between passing and failing tests. The resulting signals are incorporated through an output-profile-aware adaptive combination strategy: depending on the behavioral characteristics of the subject program, SSFL either emphasizes semantic-spectrum divergence or combines it with complementary coverage evidence. The detailed combination mechanism is presented in Section 2.4.4. A higher score indicates stronger evidence that a method is associated with a fault.

By integrating these phases, our approach effectively constructs a **Semantic Spectrum** for each method, enabling precise fault localization that transcends the limitations of traditional structure-based techniques. We illustrate the general discussion in Figure 1.

2.3 Basic Mathematical Model

Let P be a program with method set $\mathcal{M}$, and let $F \in \mathcal{M}$ denote a method. Let T be a set of test cases and let $t \in T$ be a test case. We define $F(t) = [s_1, s_2, \ldots, s_r]$ as the captured output sequence of method F during test t, where s_i is the string representation of the i-th captured value. Repeated values are retained. The captured values are return values for non-void methods and selected mutable parameter states for void methods. Each s_i belongs to $\mathcal{A}^*$, where $\mathcal{A}$ is the alphabet of possible output symbols and $\mathcal{A}^*$ is the set of all finite strings over $\mathcal{A}$.

Let Tok : $\mathcal{A}^* \to \mathcal{TK}^*$ be a fixed tokenization function, where $\mathcal{TK}$ is the token vocabulary and $\mathcal{TK}^*$ is the set of all finite token sequences over that vocabulary. Applying Tok to $s_1, s_2, \ldots, s_r$ and concatenating the resulting token sequences gives

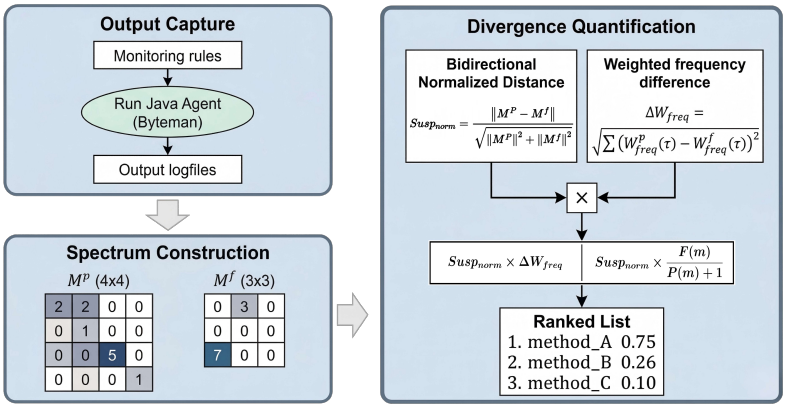


Fig. 1 Overview of Semantic Spectrum-based Fault Localization (SSFL)

$S_t = (\tau_1, \tau_2, \ldots, \tau_q)$. Repeated outputs therefore contribute repeatedly to S_t. Because SSFL uses only token occurrence counts, the order of the captured outputs does not affect the resulting semantic spectrum. The concrete tokenization rules used in our implementation are described in Section 3.4; the mathematical model requires only that the same fixed mapping be applied to all executions.

From S_t, we construct a token-frequency mapping $\Delta_t : \mathcal{TK} \to \mathbb{N}$ that counts the occurrences of each token during test t.

Definition 1 (Frequency Mapping) Given the token sequence $S_t = (\tau_1, \tau_2, \ldots, \tau_q)$ defined above, the frequency mapping Δ_t is defined for each token $\tau \in \mathcal{TK}$ as:

$$\Delta_t(\tau) = \sum_{i=1}^{q} \lambda(\tau_i = \tau) \tag{1}$$

where λ is a predicate function:

$$\lambda(cond) = \begin{cases} 1 & cond \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

We use an example to illustrate how a frequency mapping is obtained.

Example 1 Suppose that the output of method F in test case t_1 is $F(t_1) =$ [“aabd”, “dcab”]. That is, the output sequence $F(t_1)$ contains two strings, and the resulting token sequence is $S = (a, a, b, d, d, c, a, b)$. Its token frequency mapping Δ_{t_1} is as follows:

$$\Delta_{t_1}(a) = 3, \Delta_{t_1}(b) = 2, \Delta_{t_1}(c) = 1, \Delta_{t_1}(d) = 2.$$

Equivalently, relative to the ordered token list (a, b, c, d), the mapping is represented by the vector

$$(\Delta_{t_1}(a), \Delta_{t_1}(b), \Delta_{t_1}(c), \Delta_{t_1}(d)) = (3, 2, 1, 2).$$

To accumulate the frequency of each token in all test cases, we define the accumulative frequency mapping that counts the total number of test cases in which a token τ appears exactly n times.

Definition 2 (Accumulative Frequency Mapping) Define $\Theta : \mathcal{TK} \times \mathbb{N}_{>0} \to \mathbb{N}$ as follows:

$$\Theta(\tau, n) = \sum_{t \in T} \lambda(\Delta_t(\tau) = n) \tag{3}$$

$\Theta(\tau, n)$ gives the total number of test cases in which the token τ appears exactly n times. We use an example to illustrate the accumulative frequency mapping.

Example 2 Let the test case set be $T = \{t_1, t_2, t_3\}$, and the outputs of method F be

$$F(t_1) = [\text{“aabd”}, \text{“dcab”}], \quad F(t_2) = [\text{“ecaAa”}], \quad F(t_3) = [\text{“dxBaA”}].$$

By applying Definition 1, the frequency mappings with respect to T are obtained as follows. For the ordered token list $\boldsymbol{\tau} = (a, b, c, d, e, x, A, B)$, let $\Delta_t[\boldsymbol{\tau}]$ denote the vector of values of Δ_t in that order:

$$\Delta_{t_1}[\boldsymbol{\tau}] = (3, 2, 1, 2, 0, 0, 0, 0)$$
$$\Delta_{t_2}[\boldsymbol{\tau}] = (2, 0, 1, 0, 1, 0, 1, 0)$$
$$\Delta_{t_3}[\boldsymbol{\tau}] = (1, 0, 0, 1, 0, 1, 1, 1)$$

With these data, we obtain Table 3, in which the cell in row n and the column corresponding to token τ contains the set of test cases where τ appears exactly n times:

Table 3 Test cases grouped by the within-test occurrence count n of each token τ.

$n\backslash\tau$	a	b	c	d	e	x	A	B
1	$\{t_3\}$	{}	$\{t_1, t_2\}$	$\{t_3\}$	$\{t_2\}$	$\{t_3\}$	$\{t_2, t_3\}$	$\{t_3\}$
2	$\{t_2\}$	$\{t_1\}$	{}	$\{t_1\}$	{}	{}	{}	{}
3	$\{t_1\}$	{}	{}	{}	{}	{}	{}	{}

Taking the cardinality of the test-case set in each cell of Table 3 yields Table 4, which represents the mapping Θ.

Table 4 Accumulative frequency mapping Θ over test set T.

$n\backslash\tau$	a	b	c	d	e	x	A	B
1	1	0	2	1	1	1	2	1
2	1	1	0	1	0	0	0	0
3	1	0	0	0	0	0	0	0

From Table 4, we see that the mapping Θ can be represented as a matrix M. The rows and columns of this matrix correspond to the tokens and their within-test occurrence counts, respectively; Tables 3 and 4 display the transpose of this orientation for compactness. We describe the correspondence by the following equation:

$$\Theta(\tau, n) = M_{\tau,n} \tag{4}$$

2.4 Suspiciousness Quantification via Matrix Distance

In this subsection, we partition the test case set T into the set of passing test cases T_p and the set of failing test cases T_f, and construct the accumulative token frequency mappings on each set, respectively. This yields two matrices for each method F: the passing matrix M^p, which captures the output semantics of F under normal execution, and the failing matrix M^f, which captures the output semantics of F under faulty execution. Together, M^p and M^f form a dual semantic profile of method F. The divergence between these two matrices reflects the degree to which the method's behavior deviates between passing and failing tests and serves as the foundation for computing the suspiciousness score.

2.4.1 Dual-Matrix Representation

Based on this idea, we define two accumulative mappings as follows:

Definition 3 (Passing/Failing Accumulative Frequency Mappings) Let T_p and T_f be the sets of passing and failing test cases, respectively. Their accumulative frequency mappings are:

$$\Theta_p(\tau, n) = \sum_{t \in T_p} \lambda(\Delta_t(\tau) = n) \tag{5}$$

$$\Theta_f(\tau, n) = \sum_{t \in T_f} \lambda(\Delta_t(\tau) = n) \tag{6}$$

Because each accumulative mapping corresponds to a matrix, we use a pair of matrices to represent the two mappings. The following example illustrates how Θ_p and Θ_f are obtained.

Example 3 Let $T_p = \{t_1, t_2\}$ be the set of passing test cases and $T_f = \{t_3\}$ be the set of failing test cases. We have already obtained the accumulative frequency mapping Θ with respect to T in Table 4. We now split it into Table 5 for T_p and Table 6 for T_f.

Definition 4 (Method Output Matrix) Let M^p and M^f be the matrices that correspond to Θ_p and Θ_f, with rows indexed by tokens τ and columns indexed by positive occurrence counts n. Their entries satisfy:

$$\Theta_p(\tau, n) = M^p_{\tau,n} \tag{7}$$

$$\Theta_f(\tau, n) = M^f_{\tau,n} \tag{8}$$

Table 5 Accumulative frequency mapping Θ_p over the passing-test set T_p.

$n\backslash\tau$	a	b	c	d	e	x	A	B
1	0	0	2	0	1	0	1	0
2	1	1	0	1	0	0	0	0
3	1	0	0	0	0	0	0	0

Table 6 Accumulative frequency mapping Θ_f over the failing-test set T_f.

$n\backslash\tau$	a	b	c	d	e	x	A	B
1	1	0	0	1	0	1	1	1
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0

2.4.2 Bidirectional Normalized Euclidean Distance

To quantify the behavioral difference of a method between passing and failing tests, we first define a bidirectional normalized Euclidean distance over the frequency matrices. The passing test frequency matrix M^p and failing test frequency matrix M^f are treated as vectors in a high-dimensional space indexed by (token, frequency) pairs. Since M^p and M^f may contain different sets of tokens $\mathcal{TK}^p$ and $\mathcal{TK}^f$, we align them over the union $\mathcal{TK} = \mathcal{TK}^p \cup \mathcal{TK}^f$. For any token $\tau \in \mathcal{TK}$ that appears in only one matrix, the missing entries in the other matrix are treated as zero.

Definition 5 The bidirectional normalized Euclidean distance is defined as:

$$Susp_{norm}(m) = \frac{\|M^p - M^f\|}{\sqrt{\|M^p\|^2 + \|M^f\|^2}} \tag{9}$$

where

$$\|M^p\|^2 = \sum_{\tau \in \mathcal{TK}} \sum_j {M^p_{\tau,j}}^2 \tag{10}$$

$$\|M^f\|^2 = \sum_{\tau \in \mathcal{TK}} \sum_j {M^f_{\tau,j}}^2 \tag{11}$$

$$\|M^p - M^f\|^2 = \|M^p\|^2 + \|M^f\|^2 - 2 \cdot M^p \cdot M^f \tag{12}$$

$$M^p \cdot M^f = \sum_{\tau \in \mathcal{TK}} \sum_j M^p_{\tau,j} \cdot M^f_{\tau,j} \tag{13}$$

The denominator normalizes the distance to the range $[0, 1]$, ensuring comparability across methods with different call frequencies.

In practice, M^p and M^f often have largely disjoint token-frequency distributions, particularly when the failing tests exercise the method under different inputs or when the defect causes the method to produce entirely different output tokens. In such cases, the dot product $M^p \cdot M^f \approx 0$, which yields $\|M^p - M^f\|^2 \approx \|M^p\|^2 + \|M^f\|^2$, and consequently $Susp_{norm}(m) \approx 1$ for many methods regardless of whether they are actually defective. This saturation effect reduces the discriminative power of the metric.

2.4.3 Weighted Frequency Difference

To address this limitation, we introduce a complementary metric ΔW_{freq} that captures behavioral divergence through token-level weighted frequencies rather than through raw matrix entries. It aggregates the differences between $W^p_{freq}(\tau)$ and $W^f_{freq}(\tau)$ over the output tokens. A token's weighted frequency incorporates both its total occurrences and its token weight w_τ. Sign tokens ($\oplus$ for positive and $\ominus$ for negative numerical returns) receive a higher token weight, reflecting the principle that a change in the sign of a numerical output is a stronger indicator of faulty behavior than a change in digit composition alone. A higher ΔW_{freq} indicates that the method exhibits substantially different output patterns under failing tests compared with passing tests.

Definition 6 The weighted frequency difference is defined as:

$$\Delta W_{freq}(m) = \sqrt{\sum_{\tau \in \mathcal{TK}^p \cup \mathcal{TK}^f} \left(W^p_{freq}(\tau) - W^f_{freq}(\tau) \right)^2} \tag{14}$$

where $\mathcal{TK}^p$ and $\mathcal{TK}^f$ are the token sets of M^p and M^f respectively, and for tokens appearing in only one matrix, the corresponding weighted frequency in the other is treated as zero.

The weighted frequency for each token τ is computed as:

$$W^p_{freq}(\tau) = \frac{\sum_j w_\tau \cdot j \cdot M^p_{\tau,j}}{\sum_{\tau' \in \mathcal{TK}^p} \sum_j w_{\tau'} \cdot j \cdot M^p_{\tau',j}} \tag{15}$$

$$W^f_{freq}(\tau) = \frac{\sum_j w_\tau \cdot j \cdot M^f_{\tau,j}}{\sum_{\tau' \in \mathcal{TK}^f} \sum_j w_{\tau'} \cdot j \cdot M^f_{\tau',j}} \tag{16}$$

where the token weight function w_τ is defined as:

$$w_\tau = \begin{cases} \alpha & \text{if } \tau \in \{\oplus, \ominus\} \\ 1 & \text{otherwise} \end{cases} \tag{17}$$

with $\alpha = 5$ as the weight coefficient of the sign token.

Figures 2 and 3 illustrate the computation of ΔW_{freq} for the non-defective method A and the defective method B from our running example, respectively. In each figure,

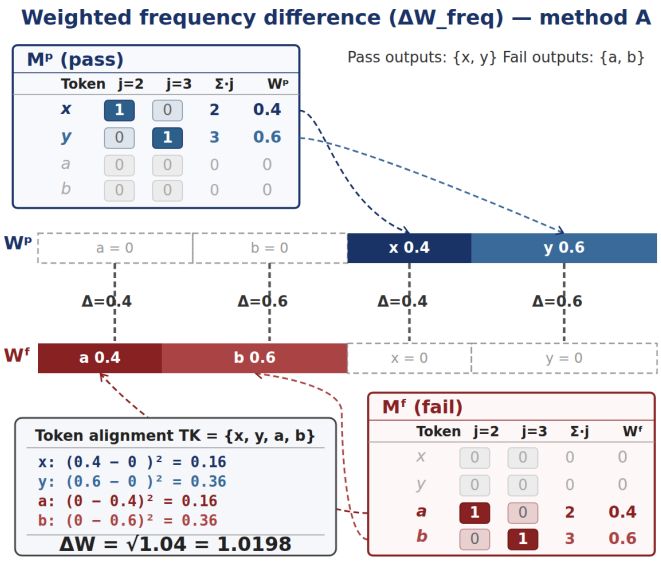


Fig. 2 Illustration of the Weighted Frequency Difference (ΔW_{freq}) computation for method A

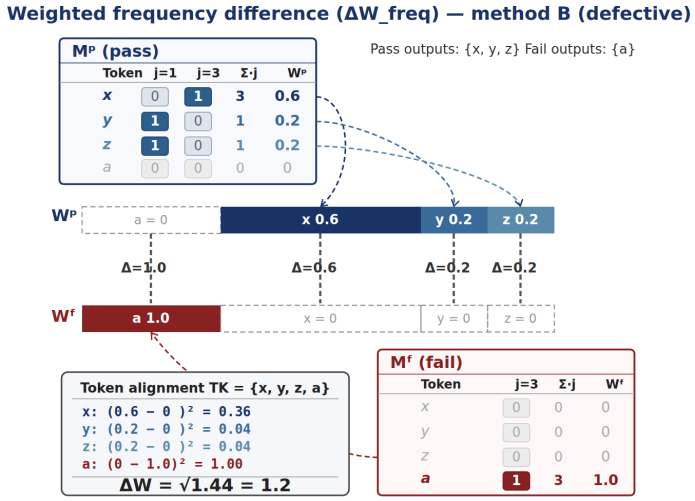


Fig. 3 Illustration of the Weighted Frequency Difference (ΔW_{freq}) computation for method B

the frequency matrices M^p and M^f are shown with their frequency-weighted sums ($\Sigma \cdot j$) and the resulting weighted frequencies (W_{freq}). The two horizontal bars show $W^p_{freq}(\tau)$ (blue) and $W^f_{freq}(\tau)$ (red) for each token, where dashed slots indicate tokens with zero weighted frequency and vertical lines annotate the per-token difference.

For method A, the passing tokens $\{x, y\}$ and failing tokens $\{a, b\}$ are entirely disjoint, producing $\Delta W_{freq} = 1.0198$. For the defective method B, the divergence is even

more pronounced: the passing distribution spreads across three tokens $\{x, y, z\}$, while the failing distribution concentrates entirely on a single token $\{a\}$, producing a higher $\Delta W_{freq} = 1.2$. The greater ΔW_{freq} of B correctly reflects its stronger behavioral shift under failure, enabling SSFL to rank the defective method above the non-defective one.

2.4.4 Adaptive Combination of Suspiciousness Signals

The most direct way to combine the two metrics is to take their product. This yields our primary, *spectrum-divergence* score:

$$S_{DW}(m) = Susp_{norm}(m) \times \Delta W_{freq}(m). \tag{18}$$

In practice, however, ΔW_{freq} is not always a dependable discriminator: when a method's outputs barely shift between passing and failing tests, ΔW_{freq} carries only a weak signal. To guard against this, we introduce a second, *coverage-based* score built on the ratio $F/(P+1)$, where F and P denote, respectively, the number of failing and passing tests that cover method m:

$$S_{FP}(m) = Susp_{norm}(m) \times \frac{F(m)}{P(m)+1}. \tag{19}$$

The two scores probe different fault manifestations. S_{DW} exposes *spectrum-divergent* faults, whose output token distributions shift visibly under failure; S_{FP} exposes *coverage-anomalous* faults—methods that are exercised by the failing tests but rarely by the passing ones, so that the very fact of being executed during failure is itself suspicious.

The two signals capture complementary fault manifestations, but the need for the auxiliary coverage signal depends on the output profile of the subject project. We therefore employ an output-profile-aware adaptive combination strategy at the project level. Projects dominated by relatively simple output representations use the semantic-spectrum divergence directly because ΔW_{freq}, together with the normalized matrix distance, provides sufficient discrimination between passing and failing executions. Projects with richer token patterns or more complex output structures use CombMAX to retain the stronger evidence supplied by either semantic-spectrum divergence or coverage anomaly.

- **Chart and Time (spectrum mode).** These projects exhibit relatively concentrated output profiles dominated by primitive and void return types (Table 7). All versions are therefore scored by S_{DW}, allowing the Weighted Frequency Difference, supported by the normalized matrix distance, to directly capture changes in the monitored runtime values.
- **Lang, Math, and Closure (normalized max-fusion).** Their semantic spectra contain richer token patterns arising from strings, numerical computations, and complex object or abstract syntax tree (AST) outputs. Spectrum divergence and coverage anomaly therefore provide complementary evidence. We fuse the two scores with the classical CombMAX rule from score fusion in information retrieval Fox and Shaw (1994):

$$Susp(m) = \max\left(\frac{S_{DW}(m)}{\max_k S_{DW}(k)}, \frac{S_{FP}(m)}{\max_k S_{FP}(k)}\right), \tag{20}$$

where the maxima in the denominators range over all candidates of the version and ties are broken by the sum of the two normalized scores. Consequently, a tie under either component score does not necessarily remain after fusion; it persists only when both the fused maximum and the tie-breaking sum are equal. Each method is thus credited with the stronger of its two normalized pieces of evidence.

These assignments are fixed for all versions of each project; the adaptive policy is deterministic and uses no learned model, calibration stage, version-specific threshold, or per-version switch. For the CombMAX projects, the normalization terms are computed solely from the candidate methods of the version being analyzed.

2.4.5 Procedure for Fault Localization

The specific procedure for fault localization presented in this paper is outlined as follows.

1. First, targeting *Defects4J*, we obtain the list of failing tests for the buggy project. This list is crucial for our subsequent work as it forms the foundation for identifying suspicious methods. The underlying rationale is that if the defective code or method is not executed, the defect will not be triggered, and consequently, no test failure will occur. Therefore, we treat the methods covered by the failing tests as our candidate list of suspicious methods. We utilize the `Defects4J coverage` tool to extract this list of methods covered by the failing tests.

2. This is the most critical step: capturing the output of the suspicious methods. To accomplish this, we employ a Java Agent, *Byteman*. We then use the list of suspicious methods to generate the necessary *Byteman* rule files. Upon running the tests, *Byteman* automatically captures the output of the method. For void methods, it monitors changes in parameter states and captures the parameter values as the output. Since *Byteman*'s log files can be substantial—often exceeding 100 KB—we limit the length of each captured string representation to a fixed maximum of 64 characters. The resulting output log contains the signatures and output of the method, and the name of the test case that executed the method.

3. Next, we apply algorithms to statistically analyze the tokens in the output log and generate matrices and construct M^p and M^f. In these matrices, the row (or horizontal axis) represents the token appearing in the method's output during testing, and the column (or vertical axis) represents the frequency of the token's appearance. The value at a given cell denotes the number of test cases in which that token appears exactly that many times.

4. Finally, for each method with output observations from both passing and failing tests, we calculate the normalized Euclidean matrix distance and the weighted frequency difference between M^p and M^f to obtain its semantic-spectrum divergence score. We then apply the output-profile-aware adaptive combination strategy described in Section 2.4.4 to produce the final suspiciousness ranking. When a method has passing-test outputs but no failing-test output ($M^p \neq \varnothing$ and $M^f = \varnothing$), we instead examine the captured exception-exit information, represented as an exception stack.

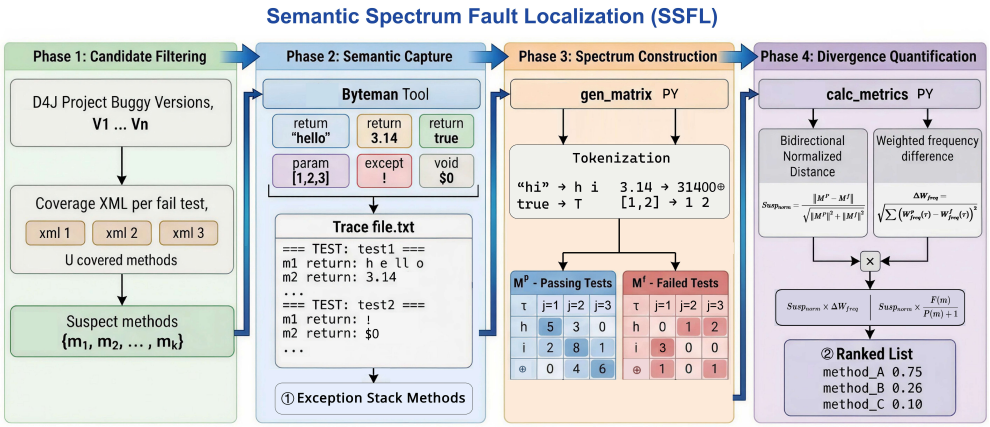


Fig. 4 Procedure of Semantic Spectrum-based Fault Localization (SSFL)

If the method appears in the exception stack, it is assigned the highest suspiciousness score. When multiple methods are identified in the exception stack, their relative ranking follows their order in the call stack—methods closer to the top of the stack (i.e., closer to the point of exception throwing) are ranked higher, as they are more directly involved in the fault propagation path. This stack-augmented ranking ensures that no suspicious method is silently omitted from consideration.

We summarize the overall procedure of fault location in Figure 4.

3 Experimental Evaluation

In this section, we present a comprehensive empirical evaluation of our proposed **Semantic Spectrum-based Fault Localization (SSFL)** framework. The primary objective of this study is to investigate whether shifting the diagnostic paradigm from *structural coverage* to **runtime output semantics** yields a quantifiable improvement in localization precision.

Specifically, we aim to validate if the **Semantic Spectrum** can effectively distinguish defective behaviors that remain indistinguishable under traditional structural spectra (i.e., resolving the "tie problem"). To ensure a rigorous and reproducible assessment, we conduct our experiments on the widely-adopted *Defects4J* benchmark Just et al. (2014). Our evaluation is guided by the following four research questions:

3.1 Research Questions

Our investigation is structured around the following research questions:

RQ1: How does SSFL compare to traditional SBFL techniques? This question validates our core hypothesis: by analyzing method output behavior instead of code coverage, SSFL can effectively localize faults even when methods exhibit identical coverage spectra, thereby overcoming a fundamental limitation of SBFL.

RQ2: How does SSFL compare to state-of-the-art deep learning-based and LLM-based fault localization methods? This question examines whether SSFL, as a training-free and lightweight approach, can achieve competitive or superior performance against techniques that leverage project-specific training data (HetFL) or large language model reasoning combined with multiple information sources (FlexFL).

RQ3: How much does each component of SSFL contribute? This question conducts an ablation study that contrasts the two scoring signals of SSFL—the spectrum-divergence score $Susp_{norm} \times \Delta W_{freq}$ and the distance-based score $Susp_{norm} \times F/(P+1)$—and quantifies the additional contribution of the adaptive combination strategy on top of either signal alone.

RQ4: What is the runtime overhead of SSFL? This question quantifies the practical cost that SSFL incurs across its stages—Byteman instrumentation, frequency matrix construction, and suspiciousness computation—and compares it against the overhead of traditional SBFL.

3.2 Experimental Subjects

Our dataset is the original five-project, 357-bug Defects4J benchmark, commonly referred to as the Defects4J v1.0 bug set. It comprises the Chart, Closure, Lang, Math, and Time projects. Although later Defects4J releases have expanded the benchmark, we retain this established subject set because it is widely used by prior fault localization studies and therefore provides the most consistent basis for comparison with the published baseline results considered in this work. The brief statistics of all projects are presented in Table 7. The table also lists the four most frequent return types in each project, providing a qualitative overview of the distinct output profiles processed by SSFL.

Table 7 Subject projects and their dominant method return types.

Project	No. of bugs	KLOC	No. of tests	1st	2nd	3rd	4th
Chart	26	96	2.2k	void	bool	double	int
Closure	133	91	7.9k	void	bool	node	string
Lang	65	22	2.3k	string	void	bool	int
Math	106	84	4.4k	double	void	int	bool
Time	27	28	4.0k	int	long	void	bool
Total	357						

Table 7 shows that the subject projects exhibit distinct return-type profiles, which affect the form and diversity of their semantic spectra. These differences motivate the output-profile-aware adaptive combination strategy described in Section 2.4.4; the table is intended as a qualitative characterization rather than as a learned or threshold-based selection rule.

3.3 Evaluation Metric

To quantitatively evaluate the effectiveness of SSFL, we report complementary rank-based metrics commonly used in fault localization: Top-k, Mean First Rank (MFR), and Mean Average Rank (MAR). When methods receive identical suspiciousness scores, their rank is determined using the expected inspection position represented by the **Wasted Effort** metric (also referred to as Absolute Average Rank) Xuan and Monperrus (2014); Zou et al. (2021).

This metric models the practical cost of debugging by estimating the number of non-faulty methods a developer must inspect before locating the root cause. Unlike simple ranking, Wasted Effort rigorously accounts for the "tie problem"—scenarios where multiple methods share identical suspiciousness scores—by calculating the expected rank under a random inspection order.

Formally, let M be the set of all candidate methods, and $m_f \in M$ be the actual faulty method. Let $s(m)$ denote the suspiciousness score assigned to method m by a fault localization technique. The Wasted Effort, denoted as $E(m_f)$, is defined as:

$$E(m_f) = \frac{|\{m \in M \mid s(m) > s(m_f)\}| + |\{m \in M \mid s(m) \geq s(m_f)\}| + 1}{2} \tag{21}$$

Where:

- $|\{m \mid s(m) > s(m_f)\}|$ represents the count of methods strictly more suspicious than the fault.
- $|\{m \mid s(m) \geq s(m_f)\}|$ represents the count of methods with scores greater than or equal to the fault .

A lower value of $E(m_f)$ indicates higher localization efficiency, meaning the developer inspects fewer irrelevant methods before finding the bug. It is worth noting that a normalized variant of Wasted Effort, known as the **EXAM score** Wong et al. (2008), is often employed to express inspection cost as a percentage of the total code base (i.e., $EXAM = E/|M|$, where $|M|$ is the total number of methods).

For any given subject program, $|M|$ remains constant. Consequently, the EXAM score and the absolute Wasted Effort are linearly dependent and yield identical relative performance rankings when comparing different techniques. We therefore use $E_i(d)$ as the tie-aware expected rank of defective method d in bug version i and aggregate these ranks through MFR and MAR.

Let N be the number of evaluated bug versions and D_i the set of defective methods in version i. Mean First Rank (MFR) measures the average rank of the first defective method:

$$\text{MFR} = \frac{1}{N} \sum_{i=1}^{N} \min_{d \in D_i} E_i(d). \tag{22}$$

MFR reflects the expected inspection effort required to reach the first defective method.

Mean Average Rank (MAR) additionally accounts for all defective methods in versions containing multiple defective methods:

$$\mathrm{MAR} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{|D_i|}\sum_{d\in D_i} E_i(d)\right). \tag{23}$$

A lower MFR or MAR indicates better localization performance.

Top-k counts the number of bug versions for which at least one defective method appears within the first k ranked positions, with $k \in \{1, 3, 5\}$. It represents practical inspection budgets of one, three, and five methods and complements the average-rank metrics with an absolute localization-success count. SSFL and its ablation configurations use the same tie-aware expected-rank convention defined above. Values reproduced from prior studies retain the ranking conventions reported by their original authors.

3.4 Experimental Setup and Implementation

In this subsection, we detail the implementation of our **Semantic Spectrum-based Fault Localization (SSFL)** framework. The workflow consists of four sequential phases as follows.

Phase 1: Candidate Identification (Structural Filtering). This phase narrows the search space to a candidate pool $Susp_M$, defined as the set of methods exercised by at least one failing test in T_f. The rationale is that a defect can only be triggered by a method that is actually executed during a failure, so methods never reached by any failing test need not be considered. We obtain $Susp_M$ from the structural coverage information provided by the *Defects4J* coverage tool, and pass it on as the input to the subsequent semantic analysis.

Phase 2: Semantic Capture via Runtime Monitoring. The second phase focuses on monitoring the candidate methods in $Susp_M$ to capture their runtime outputs. Given that our subjects are Java programs, we utilize the **Byteman** Java agentRed Hat (2024) to automate this process.

Byteman Mechanism. Byteman operates by injecting Java bytecode into the application at runtime, allowing instrumentation without modifying the program's source code. It uses a rule-based language to specify trigger points (e.g., method entry, exit, or exception exit) and associated actions. When the program execution reaches these points, the Byteman agent intercepts the control flow to execute the defined monitoring logic before returning control to the application.

We configure Byteman to capture the return values of non-void methods at method exit. For methods returning primitive arrays (e.g., `double[]`, `int[]`), we employ Byteman's `BIND` syntax to explicitly declare the return type, enabling proper array-to-string conversion. For `void` methods, we capture the state of mutable parameters (arrays and objects) at method exit, while excluding immutable primitive-type parameters (e.g., `int`, `double`, `boolean`) that remain unchanged throughout execution. Methods that terminate via exceptions are handled by configuring an additional `AT EXCEPTION`

`EXIT` rule that records a special exception token. These monitoring rules are encoded into a `.btm` configuration file.

Writing these rules by hand for every candidate method is tedious, so we use Claude (Anthropic; the specific model version was not recorded) to assist with drafting them. Given each method's signature and return type as input, Claude produces the corresponding Byteman rule text, which we then spot-check manually before using it in the experiments. This semi-automated rule preparation is performed offline and only once per method, and therefore does not contribute to the per-version runtime overhead reported in RQ4. The complete generated and verified `.btm` rule files are included in the replication package referenced in the Data Availability Statement.

Phase 3: Semantic Spectrum Construction. The third phase parses the generated log files and constructs the **Semantic Spectra** using an automated Python script. The implementation first applies a fixed tokenization procedure to every captured value. Boolean values are mapped to single tokens (`true` $\rightarrow T$ and `false` $\rightarrow F$). Numerical values are represented by four significant digits, a scale token indicating the order of magnitude (A through F for positive exponents 10^1 through 10^6, a and b for negative exponents 10^{-1} and 10^{-2}, and O for 10^0), and a sign token ($\oplus$ or $\ominus$). For example, -49.7593 is represented as $4975b\ominus$. Arrays are tokenized element-wise using the rule for their element type; strings and other captured representations are decomposed symbol by symbol. The designated exception marker is treated as a single token. These fixed implementation choices are applied uniformly to all subject projects and bug versions.

The resulting token sequences are then aggregated into two frequency matrices: M^p is the Semantic Spectrum derived from passing test cases, and M^f is the Semantic Spectrum derived from failing test cases.

When a candidate method has passing-test outputs but no failing-test output ($M^p \neq \varnothing$ and $M^f = \varnothing$), matrix-based divergence cannot be computed. Such a method is therefore passed to the exception-stack procedure in Phase 4 rather than being silently excluded.

Phase 4: Divergence Quantification and Ranking. The final phase calculates the suspiciousness scores based on Semantic Spectra. For each candidate method, we first compute the **Bidirectional Normalized Euclidean Distance** ($Susp_{norm}$) between M^p and M^f, which quantifies the raw distributional difference. To address the saturation effect where $Susp_{norm}$ approaches 1.0 for many methods due to disjoint token-frequency distributions, we additionally compute the **Weighted Frequency Difference** (ΔW_{freq}), which captures changes in token-level weighted frequencies between passing and failing tests. Under the output-profile-aware adaptive combination strategy defined in Section 2.4.4, Chart and Time use the spectrum-divergence score, whereas Lang, Math, and Closure combine it with the coverage-based score through normalized max-fusion (CombMAX). All candidate methods are ranked in descending order of their suspiciousness scores, with higher scores indicating greater likelihood of containing the fault.

In our experimental workflow, Phases 3 and 4 are fully automated using algorithmic scripts, whereas Phase 2 is semi-automated: method signatures and return types are

supplied to Claude to draft the Byteman rules, after which the generated rules are manually spot-checked and deployed for each project version.

3.5 Results and Discussion

Evaluation protocol. The current SSFL implementation does not instrument constructors and therefore produces no ranked list for 23 constructor-residing faults. These versions are conservatively counted as not localized in all Top-k results, which retain the full denominator of 357 faults. MFR and MAR are computed over the remaining 334 versions; no arbitrary penalty rank is assigned where no ranking exists. The MFR/MAR comparisons therefore use a different evaluation coverage from baselines reported over all 357 faults and should be interpreted together with the Top-k results.

3.5.1 RQ1: comparison with Traditional SBFL

Table 8 presents the experimental results of SSFL compared to the best-performing SBFL formulae (including Ochiai Abreu et al. (2006), DStar Wong et al. (2014), and GP13 Yoo (2012)) in five projects in the Defects4J benchmark. For each project, we select the formula with the lowest MFR as the Best Baseline, representing the most favorable result achievable by existing hit-based techniques.

We use the MFR metric defined in Section 3.3 for this comparison because the available SBFL baseline data are reported in terms of MFR. MAR and Top-k, also defined in Section 3.3, are included in the comprehensive comparison in the following subsection.

Table 8 Mean First Rank (MFR) comparison of Semantic Spectrum-based Fault Localization (SSFL) with hit-based fault localization formulae on Defects4J. Lower MFR indicates better performance.

Project	Ochiai	DStar	GP13	Best Baseline	SSFL	Impr.
Lang	4.46	4.46	4.39	4.39 (GP13)	**1.73**	60.6%
Math	9.73	10.25	10.83	9.73 (Ochiai)	**3.41**	64.9%
Closure	71.64	71.63	95.49	71.63 (DStar)	**6.88**	90.4%
Time	15.96	18.65	22.81	15.96 (Ochiai)	**2.41**	84.9%
Chart	8.82	9.18	34.34	8.82 (Ochiai)	**1.80**	79.6%

Impr. $= (\text{MFR}_{\text{baseline}} - \text{MFR}_{\text{SSFL}})/\text{MFR}_{\text{baseline}} \times 100\%$

As shown in Table 8, SSFL consistently and substantially outperforms all hit-based SBFL formulae across all five Defects4J projects, achieving MFR reductions of 60%–90% compared to the best-performing baseline formula.

Among the five projects, the improvement on Closure is the most dramatic: SSFL reduces the MFR from 71.63 (DStar, the best baseline) to 6.88—a 90.4% reduction. The severe coverage ties inherent in hit-based formulae make it nearly impossible to distinguish the defective method from dozens of equally-scored candidates in a large project such as Closure. SSFL breaks these ties by exploiting the behavioral divergence in method output, dramatically narrowing the search space.

The improvement on Time is also particularly pronounced: SSFL reduces MFR from 15.96 (Ochiai) to 2.41, an 84.9% reduction. For Math, SSFL achieves an MFR of 3.41, representing a 64.9% reduction compared to Ochiai's 9.73. This demonstrates that SSFL's value-centric analysis is highly effective for numerical computation libraries, where defects often manifest as incorrect return values that are directly captured by the Semantic Spectrum.

For Lang and Chart, the improvements are also substantial, with MFR reductions of 60.6% and 79.6%, respectively, confirming the generalizability of our approach across different project types. The relatively modest improvement on Lang is partly attributable to its smaller candidate pool and less pronounced coverage tie problem compared to larger projects such as Closure.

3.5.2 RQ2: comparison with Learning-Based and LLM-Based Techniques

Table 9 presents a comprehensive comparison between SSFL and three representative learning-based fault localization techniques: DeepFL Li et al. (2019), which integrates suspiciousness scores from multiple FL families using deep neural networks; Grace Lou et al. (2021), which leverages graph neural networks over coverage representations; and HetFL Chen et al. (2024), a state-of-the-art heterogeneous graph-based technique. For HetFL, we report the cross-project setting, which trains on four projects and tests on the remaining one, as it better reflects real-world scenarios where historical fault data from the target project may not be available. All three baselines require project-specific or cross-project training data, whereas SSFL requires no training data whatsoever—it directly computes the behavioral divergence between passing and failing test outputs without any learning phase.

The results reveal two key findings. First, SSFL reports lower MFR values across all five projects, indicating that the defective method is ranked close to the top on average. At the overall level, SSFL attains an MFR of 4.25, compared with 7.85 for HetFL under the respective evaluation protocols; the MAR results, which we report for completeness, show a consistent trend (5.28 vs. 9.16). The improvement is most pronounced on Closure, where SSFL reduces MFR from 12–15 to 6.88, and on Time, where MFR drops from 12–16 to 2.41. This demonstrates that SSFL produces more stable and consistently high rankings, even for projects with large numbers of suspicious methods.

Second, SSFL also leads in Top-k counts at the overall level. SSFL localizes 172, 242, and 262 bugs at Top-1, Top-3, and Top-5, respectively, compared to HetFL's 151, 195, and 228. The advantage is particularly striking at Top-3 and Top-5, where SSFL surpasses HetFL by 47 and 34 bugs in absolute count. On a per-project basis, SSFL leads on Lang and Math across all metrics and achieves the highest Top-3 and Top-5 on every project. The single per-project metric on which SSFL trails a baseline is Top-1 on Closure (48 vs. DeepFL's 51); as discussed in Section 3.5.3, this reflects a deliberate recall-oriented trade-off of the max-fusion policy, which SSFL repays with decisive leads at Top-3 (67 vs. 65) and Top-5 (78 vs. 73) on the same project.

These results highlight a fundamental advantage of SSFL: by leveraging output value semantics rather than structural coverage patterns, SSFL captures diagnostic

Table 9 Comparison of SSFL with representative learning-based and large language model (LLM)-based fault localization techniques on Defects4J using Top-k, Mean First Rank (MFR), and Mean Average Rank (MAR). Lower MFR and MAR values indicate better performance.

Technique	Top-1		Top-3		Top-5		Total	MFR	MAR
	n	%	n	%	n	%			
Lang									
DeepFL	35	53.8%	40	61.5%	46	70.8%	65	2.72	3.38
Grace	32	49.2%	41	63.1%	46	70.8%	65	3.21	3.67
HetFL (cross)	37	56.9%	42	64.6%	46	70.8%	65	2.59	3.02
SSFL	**42**	**64.6%**	**51**	**78.5%**	**55**	**84.6%**	65	**1.73**	**2.11**
Math									
DeepFL	44	41.5%	57	53.8%	72	67.9%	106	5.32	6.15
Grace	40	37.7%	56	52.8%	69	65.1%	106	6.73	7.60
HetFL (cross)	46	43.4%	63	59.4%	76	71.7%	106	5.44	6.21
SSFL	**57**	**53.8%**	**86**	**81.1%**	**88**	**83.0%**	106	**3.41**	**3.96**
Closure									
DeepFL	**51**	**38.3%**	65	48.9%	73	54.9%	133	12.21	16.56
Grace	39	29.3%	57	42.9%	65	48.9%	133	15.14	20.16
HetFL (cross)	46	34.6%	61	45.9%	69	51.9%	133	14.58	19.22
SSFL	48	36.1%	**67**	**50.4%**	**78**	**58.6%**	133	**6.88**	**8.85**
Time									
DeepFL	8	29.6%	11	40.7%	14	51.9%	27	16.23	16.83
Grace	6	22.2%	10	37.0%	14	51.9%	27	12.16	13.08
HetFL (cross)	10	37.0%	13	48.1%	17	63.0%	27	11.87	12.14
SSFL	**12**	**44.4%**	**18**	**66.7%**	**20**	**74.1%**	27	**2.41**	**2.95**
Chart									
DeepFL	9	34.6%	14	53.8%	16	61.5%	26	4.41	5.02
Grace	8	30.8%	13	50.0%	15	57.7%	26	5.16	5.39
HetFL (cross)	12	46.2%	16	61.5%	20	76.9%	26	4.80	5.23
SSFL	**13**	**50.0%**	**20**	**76.9%**	**21**	**80.8%**	26	**1.80**	**2.00**
Overall									
DeepFL	147	41.2%	187	52.4%	221	61.9%	357	8.18	9.59
Grace	125	35.0%	177	49.6%	209	58.5%	357	8.48	9.98
HetFL (cross)	151	42.3%	195	54.6%	228	63.9%	357	7.85	9.16
FlexFL	164	46.5%	214	60.6%	236	66.9%	353	–	–
SSFL	**172**	**48.2%**	**242**	**67.8%**	**262**	**73.4%**	357	**4.25**	**5.28**

information without requiring project-specific model training. While learning-based techniques must rely on sufficient and representative training data to generalize effectively, SSFL operates on a per-version basis, analyzing the behavioral divergence of each method without any prior knowledge. This training-free property makes SSFL particularly well-suited for scenarios where historical fault data is scarce or unavailable, and eliminates the computational overhead associated with model training and hyperparameter tuning.

We also compare SSFL against recent LLM-based method-level fault localization, represented by FlexFL Xu et al. (2025) (IEEE TSE 2025), a two-stage framework that first narrows the search space using a combination of SBFL (Ochiai), information retrieval-based fault localization (IRFL; BoostN), HybridFL (SBIR), and an LLM-based agent, then leverages a second LLM agent to read the source code of candidate methods and refine the ranking. On the same five Defects4J (v1.0) projects, FlexFL reports Top-1 = 164, Top-3 = 214, and Top-5 = 236; these published overall results are included in the last block of Table 9. SSFL reports 172, 242, and 262 Top-1, Top-3, and Top-5 localizations, respectively, corresponding to 8, 28, and 26 additional localized faults. We note that SSFL relies on the runtime outputs of the existing test suite, whereas FlexFL additionally consumes bug reports and LLM-based source code comprehension, neither of which is always available in practice. This suggests that the output behavioral divergence captured by the Semantic Spectrum provides a competitive and complementary diagnostic signal, even when compared against approaches that aggregate multiple information sources including LLM reasoning. The two approaches exploit fundamentally different information dimensions—FlexFL leverages LLM code comprehension over source text, while SSFL captures runtime output value distributions—making them orthogonal and potentially complementary. Integrating SSFL as an additional input source in FlexFL's space reduction stage is a promising direction for future work.

To provide an intuitive, at-a-glance summary of the quantitative results in Tables 8 and 9, Figure 5 consolidates them into a single visualization, with the compared techniques grouped into four families: SBFL, machine learning (ML)-based, LLM-based, and our SSFL. Panel (a) plots the per-project MFR (on a logarithmic scale) of the SBFL and ML-based baselines against SSFL, showing that SSFL attains the lowest MFR on every project; panel (b) plots the overall Top-k percentages of the ML-based and LLM-based baselines against SSFL, showing that SSFL has the highest overall count at each k. Together the two panels condense the numerical evidence of both tables into a single, directly comparable view.

3.5.3 RQ3: Ablation Study

To disentangle the contributions of the two scoring signals and of the adaptive strategy that combines them, we evaluate three configurations: (i) *Spectrum-divergence*, which ranks the candidates of every version by S_{DW} (Eq. 18); (ii) *Euclid-Distance*, which ranks by S_{FP} (Eq. 19); and (iii) the full adaptive SSFL strategy of Section 2.4.4, which uses S_{DW} for Chart and Time and CombMAX for Lang, Math, and Closure. Both single-signal variants retain the $Susp_{norm}$ factor, so the comparison isolates the contribution of the second factor and of score fusion. Versions whose defective method is never executed by a passing test and is localized directly from the exception stack are counted identically under all configurations and therefore do not bias the comparison. Table 10 reports the Top-k counts.

In Table 10, "–" indicates that the standalone Euclid-Distance variant is not reported for Chart and Time, which use the Spectrum-divergence score in the full strategy. Under the tie-aware expected-rank convention, the standalone Euclid-Distance variant localizes 54 Math bugs at Top-1. Under an optimistic best-rank convention,

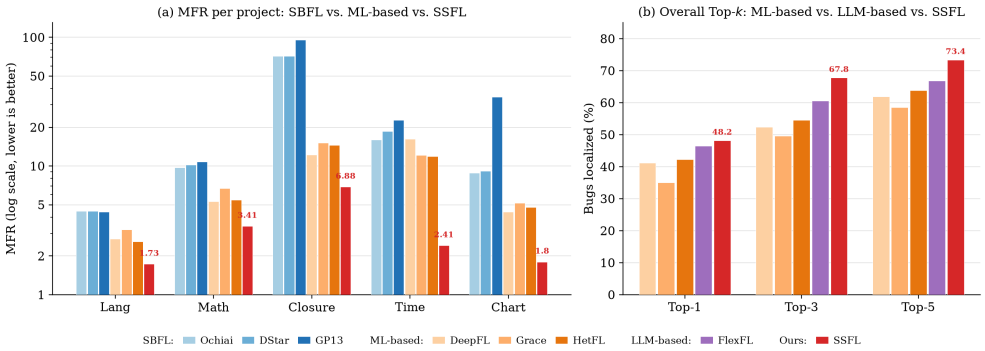


Fig. 5 Visual summary of Tables 8 and 9, with techniques grouped into four families: spectrum-based fault localization (SBFL), machine learning (ML)-based methods, large language model (LLM)-based methods, and Semantic Spectrum-based Fault Localization (SSFL). (a) Per-project Mean First Rank (MFR; log scale, lower is better) of SBFL and ML-based techniques versus SSFL. (b) Overall Top-k localization percentages of ML-based and LLM-based techniques versus SSFL. FlexFL is omitted from panel (a) as it reports no MFR under a comparable protocol; SBFL formulae are omitted from panel (b) as they provide no Top-k figures

Table 10 Ablation results for the adaptive combination strategy, reported as Top-k localization counts.

Project	Configuration	Top-1	Top-3	Top-5
Math (106)	Spectrum-divergence (S_{DW})	55	82	87
	Euclid-Distance (S_{FP})	54	76	81
	CombMAX (SSFL)	**57**	**86**	**88**
Lang (65)	Spectrum-divergence (S_{DW})	37	47	52
	Euclid-Distance (S_{FP})	43	52	54
	CombMAX (SSFL)	**42**	**51**	**55**
Time (27)	Spectrum-divergence (S_{DW})	12	18	20
	Euclid-Distance (S_{FP})	–	–	–
	Spectrum-divergence (SSFL)	**12**	**18**	**20**
Chart (26)	Spectrum-divergence (S_{DW})	13	20	21
	Euclid-Distance (S_{FP})	–	–	–
	Spectrum-divergence (SSFL)	**13**	**20**	**21**
Closure (133)	Spectrum-divergence (S_{DW})	37	66	74
	Euclid-Distance (S_{FP})	36	53	64
	CombMAX (SSFL)	**48**	**67**	**78**
Overall (357)	Spectrum-divergence (S_{DW})	154	233	254
	Adaptive combination (SSFL)	**172**	**242**	**262**

the count would be 62, but eight of these cases involve a two-way tie for first place because two methods share the same (P, F) profile. By contrast, none of the 57 Math bugs localized at Top-1 by the full SSFL strategy involves a tie for first place.

The results support three observations.

Neither signal dominates alone. The Euclid-Distance score wins Top-1 on Lang (43 vs. 37), whereas on Math its tie-aware Top-1 count is slightly lower than that of Spectrum-divergence (54 vs. 55); under optimistic best-rank treatment, the former would rise to 62 because eight two-way ties would be counted as first-place hits. It also performs less well at Top-3/Top-5 on Math (76/81 vs. 82/87) and across all three budgets on Closure (36/53/64 vs. 37/66/74). The Spectrum-divergence score is strongest at Top-3/Top-5 across these projects, but it concedes Top-1 on Lang. This asymmetry is consistent with the mechanism of Section 2.4.4. A coverage-anomalous fault has an extreme $F/(P+1)$ value and may top the list outright, whereas spectrum divergence can still place the fault at rank 2–3 when its signal is diluted, helping to explain the consistently high Top-3/Top-5 counts of S_{DW}.

The adaptive combination strategy benefits from complementary evidence. At the overall level, the full strategy improves on the uniform Spectrum-divergence configuration by +18 at Top-1, +9 at Top-3, and +8 at Top-5. Lang, Math, and Closure use the same deterministic CombMAX rule, which preserves strong evidence from either signal without a version-specific switch or threshold. On Math Top-1, the full strategy localizes 57 bugs without a first-place tie, compared with the tie-aware Euclid-Distance count of 54. The latter reaches 62 only under optimistic best-rank treatment that counts eight two-way ties as Top-1 hits.

CombMAX yields its largest project-level gain on Closure. On Closure, Spectrum-divergence and Euclid-Distance localize 37 and 36 faults at Top-1, respectively, whereas CombMAX increases the count to 48. Relative to the better single signal at each inspection budget, CombMAX adds 11 Top-1, 1 Top-3, and 4 Top-5 localizations. These results indicate that the two signals provide complementary high-ranking evidence on Closure and that max-fusion primarily improves the ordering of candidates near the top of the ranked list.

In summary, the ablation confirms that the two signals are complementary: using Spectrum-divergence alone instead of the adaptive SSFL strategy forfeits 8–18 localized bugs overall, depending on the inspection budget.

3.5.4 RQ4: Runtime Overhead Analysis

To assess the practicality of SSFL in real engineering settings, we measured its runtime cost on the five Defects4J projects and compared it against traditional SBFL. The experiments were conducted on a computer equipped with a 13th Gen Intel Core i5-13500HX processor, 16 GB of random-access memory (RAM), and 1 TB of storage. The software environment comprised a Windows host running Windows Subsystem for Linux 2 (WSL2) with Ubuntu 24.04.4 Long Term Support (LTS), Python 3.12.3, and OpenJDK 11.0.31.

The key to a fair comparison is to separate **shared overhead** from **SSFL-specific overhead**. Within the pipeline described in Section 3, collecting coverage data is a step that any SBFL tool must also perform—it likewise checks out the faulty version and runs the test suite to obtain coverage information. This cost is therefore not unique to SSFL. The only overhead SSFL genuinely adds arises from Byteman instrumentation, semantic spectrum construction, and suspiciousness computation. We note

that the generation of the Byteman rules is performed offline and only once, with LLM assistance, and is thus excluded from the per-version runtime overhead.

The SSFL-specific overhead is essentially constant across projects, at roughly 7 seconds per version: about 5.5 seconds for Byteman instrumentation (which writes the trace files), about 1.5 seconds for spectrum construction, and under 0.01 seconds for the suspiciousness computation itself. This cost is governed by the size of the runtime trace files (the captured method outputs) rather than by the project type; because Closure produces by far the largest traces—several million lines in some versions—the 7-second figure can be regarded as a conservative upper bound. Table 11 contrasts SSFL with traditional SBFL at both the per-version and whole-project scales.

Table 11 Runtime overhead of SSFL versus traditional SBFL. The shared overhead is coverage data collection (Phase 1); the SSFL-specific overhead is the sum of Byteman instrumentation, spectrum construction, and suspiciousness computation (Phases 2–4).

	Closure	Lang	Math	Time	Chart	Total
Shared overhead (s/version)	28.7	8.3	14.0	14.4	∼20	—
SSFL-specific (s/version)	7.0	7.0	7.0	7.0	7.0	—
SBFL total (min)	63.6	9.0	24.7	6.5	8.7	112.5
SSFL total (min)	79.1	16.6	37.1	9.6	11.7	154.1
Extra overhead ratio	+24%	+84%	+50%	+49%	+35%	+37%

As Table 11 shows, SSFL's additional cost over traditional SBFL ranges from 24% to 84%. Notably, this ratio runs inversely to the cost of coverage collection. Closure exhibits the lowest relative overhead (24%) precisely because its large test suite makes coverage collection the dominant cost, so the fixed surcharge is small in proportion; Lang exhibits the highest (84%) because its tests run quickly (about 8 seconds per version), leaving a small shared-cost baseline against which the fixed 7-second SSFL surcharge looms large.

At the aggregate level, across all five projects traditional SBFL spends approximately 112.5 minutes on coverage collection, whereas the full SSFL pipeline takes about 154.1 minutes—an additional 41.6 minutes, or approximately 37% more overall. Per version, coverage collection averages about 18.9 seconds (shared by SBFL and SSFL), SSFL adds about 7 seconds, and the complete SSFL pipeline averages about 25.9 seconds per version. Given that this modest surcharge accompanies the substantial gains in localization precision reported in RQ1 and RQ2, SSFL is well suited to practical debugging scenarios.

Finally, we stress that SSFL's principal bottleneck is not time but **space**. To capture method outputs, Byteman writes detailed trace files during test execution, and these can grow very large—for Closure, some versions produce trace files of several million lines. Although capping each call's output at 64 characters (see the implementation details in Section 3) alleviates this to some extent, the storage cost of the trace files may still become the dominant constraint when SSFL is scaled to larger codebases or longer test executions. Reducing this dependence on full trace files—for

instance through streaming processing or online statistics that avoid materializing the complete trace—is a promising direction for future work.

3.6 Threats to Validity

Internal validity concerns potential errors in the experimental setup that may affect the reliability of the results. First, the candidate list of suspicious methods is derived from parsing coverage XML files generated by the Defects4J framework. In rare cases, XML parsing errors may cause the candidate list to be incomplete, potentially missing certain defective methods. When a defective method is absent from the candidate list, it cannot be ranked by any technique, which may inflate MAR scores for versions containing multiple defects. However, this does not affect MFR, as MFR only measures the ranking of the first defective method encountered. Second, preparation of the Byteman instrumentation rules remains semi-automated: Claude drafts rules from method signatures and return types, but the inputs must be organized and the generated rules must be manually spot-checked and deployed for each project version. This process introduces the possibility of configuration errors and prevents full end-to-end automation of the experimental pipeline. To mitigate this, we cross-validated the captured outputs against expected method behavior for a randomly selected subset of versions. Third, the tokenization scheme involves design choices—such as the number of significant digits for numerical values and the elevated weight assigned to sign tokens—that were adopted based on preliminary experiments and applied uniformly across all projects, but different configurations could yield different rankings.

External validity concerns the generalizability of our findings. Our experiments are conducted exclusively on five Java projects from the Defects4J benchmark. While these projects are widely used in fault localization research and cover diverse application domains (compiler, mathematics library, date/time processing, charting, and language utilities), the performance of SSFL on software written in other programming languages (e.g., C/C++, Python) or on significantly larger industrial codebases remains an open question. Additionally, the current SSFL implementation does not instrument constructors; the treatment of the affected versions is specified in the evaluation protocol.

Construct validity concerns whether the evaluation metrics adequately reflect practical debugging effectiveness. We adopt Top-k, MFR, and MAR, which are standard metrics in the fault localization literature. However, these metrics model an idealized scenario where a developer inspects the ranked list linearly from the top. In practice, developers may leverage additional context—such as code familiarity, commit history, or IDE navigation—that is not captured by the ranking alone. Furthermore, these metrics evaluate localization at the method level, which may not directly correspond to the statement-level granularity that developers typically work with during debugging.

4 Related Work

We organize prior work by the source of diagnostic information that each technique exploits: structural coverage collected during test execution, the code comprehension

capabilities of LLMs, and runtime behavioral data. A further line of IRFL research matches the text of bug reports against source code; since SSFL consumes no textual artifacts, we discuss this direction only where it appears as a component of hybrid techniques such as SBIR Motwani and Brun (2023) and FlexFL Xu et al. (2025). Viewed through this lens, the essential difference of SSFL lies not in its ranking machinery but in the information it consumes: rather than recording which code was executed, what the code looks like, or how a failure is described in natural language, SSFL records what data the code actually produces at runtime.

4.1 Coverage-Based Fault Localization

The longest-standing family of techniques derives suspiciousness from structural coverage. Classic spectrum-based fault localization (SBFL) correlates the execution of program elements with test outcomes through statistical formulae Wong et al. (2016), with Ochiai Abreu et al. (2006), DStar Wong et al. (2014), and GP13 Yoo (2012) among the most widely used. Despite extensive study, Zou et al. (2021) showed that standalone SBFL techniques on the Defects4J benchmark Just et al. (2014) are fundamentally limited by the coverage tie problem Xu et al. (2011), where multiple entities receive identical scores. To mitigate this, SBIR Motwani and Brun (2023) aggregates SBFL and IRFL ranked lists via Monte Carlo rank aggregation without requiring training data.

Prior work in the *Journal of King Saud University – Computer and Information Sciences* has investigated three complementary directions for multiple-fault localization. PSOMFL formulates localization as a particle-swarm search over candidate fault distributions Cao et al. (2023b); ACOMFL constructs a search graph from coverage information and applies ant-colony optimization with genetic operations Cao et al. (2023c); and a parallel-localization approach uses Chameleon clustering to partition failing execution traces into fault-focused clusters Cao et al. (2023a). These approaches optimize or partition structural execution evidence, whereas SSFL introduces method output values as an additional behavioral source of diagnostic information.

A second generation of techniques replaces the statistical formula with a learned model while retaining coverage as the core input. Zhang et al. (2017) first applied deep neural networks to fault localization, followed by CNN-FL Zhang et al. (2019), which learns fault patterns from coverage matrices. DeepFL Li et al. (2019) integrates suspiciousness scores from SBFL and mutation-based fault localization with code complexity and text similarity features. Graph neural network approaches have proven particularly effective: Grace Lou et al. (2021) represents test-entity coverage as a graph and uses gated graph neural networks for listwise ranking, while DepGraph Rafi et al. (2024b) extends this with call dependency and code evolution information, and HetFL Chen et al. (2024) constructs a heterogeneous graph aggregating coverage, code change histories, and method call dependencies. Other notable contributions include Meng et al. (2022) on deep semantic features, TS-FL Zhang et al. (2024) on teacher–student knowledge transfer, and Zhang et al. (2023) on context-aware representations; a recent study Rafi et al. (2024a) further highlights the importance of benchmark data quality in evaluating these techniques.

For classical SBFL and for many learning-based extensions, structural coverage remains a core information source: it records *which* code was executed, whereas the additional structural, historical, or textual features still do not directly record *what values* a method produced. Consequently, identical coverage profiles remain difficult to distinguish unless an added information source separates the tied entities. SSFL addresses this limitation through runtime output-value evidence rather than through another transformation of the coverage spectrum.

4.2 LLM-Based Fault Localization

The rapid advancement of LLMs has opened a new paradigm for fault localization. Rather than relying on statistical formulae or trained neural networks, LLM-based approaches leverage the code comprehension and reasoning capabilities of pretrained language models to directly analyze source code and bug-related information.

AutoFL Kang et al. (2024) is among the first to bring LLM-based FL to the project level. It equips GPT-3.5/GPT-4 with function calls that allow the LLM to autonomously navigate a software repository—retrieving class structures, method signatures, and code snippets—to localize buggy methods, improving method-level Top-1 accuracy by up to 233.3% over non-LLM baselines on 798 real-world bugs in Java and Python. LLMAO Yang et al. (2024) takes a fundamentally different approach by fine-tuning lightweight bidirectional adapters on pretrained LLMs to predict buggy lines *without any test coverage information*, demonstrating that LLMs can perform fault localization even in test-free scenarios.

Subsequent work has explored multi-agent architectures to tackle the complexity of project-level FL. SoapFL Qin et al. (2025) (IEEE TSE 2025) employs a multi-agent system that models FL as a three-phase process—comprehension, navigation, and confirmation—where specialized agents collaborate to iteratively refine suspicious method rankings. FuseFL Widyasari et al. (2024) combines SBFL results, test execution outcomes, and LLM-generated code descriptions to guide step-by-step reasoning for explainable fault localization. FlexFL Xu et al. (2025) (IEEE TSE 2025) proposes a two-stage framework that first narrows the search space using SBFL, IRFL, HybridFL, and an LLM agent, then leverages a second LLM agent to read candidate method source code and refine the ranking. FlexFL demonstrates that even lightweight open-source LLMs such as Llama3-8B can achieve competitive method-level performance, outperforming AutoFL and SoapFL with GPT-3.5.

Despite this remarkable progress, LLM-based approaches may depend on proprietary or computationally expensive models and exhibit non-determinism that requires repeated runs to stabilize results. Although hybrid frameworks can consume bug reports, test outcomes, coverage, and source code, they do not explicitly model how the distributions of method output values differ between passing and failing executions. SSFL is therefore complementary: it provides an orthogonal diagnostic signal—the behavioral divergence of method outputs—that could serve as an additional input to enhance LLM-based frameworks.

4.3 Runtime Behavior-Based Fault Localization

A complementary line of research exploits runtime behavioral information beyond structural coverage. Yan et al. (2023) model fault propagation context to refine suspiciousness scores, while Vancsics et al. (2022) investigate function call frequencies as an alternative spectrum. SmartFL Zeng et al. (2022) is the most closely related to our work: it abstracts program values into binary correctness labels and performs Bayesian inference over a dynamic dependency graph. However, SmartFL and SSFL differ fundamentally. SmartFL operates at the statement level and reasons about whether values *could be* correct through probabilistic inference, requiring 205 seconds per fault on average. SSFL operates at the method level and directly observes *what values are actually produced*, comparing output token frequency distributions between passing and failing tests via lightweight Byteman instrumentation. SmartFL's binary abstraction discards the magnitude and distribution of output values—precisely the information that SSFL's frequency matrices and divergence metrics capture. The two approaches are complementary: SmartFL models intra-method value propagation, while SSFL models inter-test output distributions.

Our work takes a different approach from the above categories. SSFL directly captures the *output values* produced by methods during execution and quantifies behavioral divergence between passing and failing tests, exploiting a diagnostic dimension—the semantic content of method outputs—that is orthogonal to coverage-based and LLM-based techniques and finer-grained than the runtime signals used in prior behavior-based work. Although an LLM was used offline to assist in drafting instrumentation rules, SSFL requires neither model training nor online LLM reasoning to compute suspiciousness rankings; the rankings themselves are derived deterministically from the runtime evidence of each program version.

5 Conclusion

In this paper, we proposed Semantic Spectrum-based Fault Localization (SSFL), a novel approach that shifts the fault localization perspective from structural coverage to output value semantics. By monitoring the runtime outputs of methods and transforming them into frequency matrices through a unified tokenization process, SSFL captures the behavioral divergence between passing and failing test executions—a diagnostic signal that is fundamentally different from and complementary to traditional coverage-based information. We introduced two complementary metrics to quantify this divergence: the Bidirectional Normalized Euclidean Distance, which measures the raw distributional difference between the passing and failing semantic spectra, and the Weighted Frequency Difference, which captures changes in token-level weighted frequencies. Together, these metrics support suspicious-method ranking, including cases in which structural coverage provides limited discriminative information.

Evaluation on five projects from the Defects4J benchmark provides empirical evidence of the effectiveness of SSFL on the studied subjects. Compared with classical SBFL formulae such as Ochiai, DStar, and GP13, SSFL reports MFR values that are 60%–90% lower across the five projects under the evaluation protocol described in

Section 3.5. Compared with three representative learning-based techniques—DeepFL, Grace, and HetFL—the Top-k results provide the primary basis for comparison because all techniques report these metrics. SSFL localizes 172, 242, and 262 bugs at Top-1, Top-3, and Top-5, respectively, compared with 151, 195, and 228 for HetFL, the strongest learning-based baseline overall. Thus, SSFL localizes 21, 47, and 34 additional bugs at Top-1, Top-3, and Top-5, respectively. SSFL also reports an overall MFR of 4.25, compared with 7.85 for HetFL under the respective evaluation protocols. Compared with FlexFL, a representative LLM-based technique that leverages bug reports, SBFL results, and LLM reasoning, SSFL reports 8, 28, and 26 additional Top-1, Top-3, and Top-5 localizations (172/242/262 versus 164/214/236). Beyond localization accuracy, SSFL is training-free and deterministic, requires no bug reports or online LLM reasoning, and derives its rankings directly from runtime evidence in the existing test suite. Its semantic spectrum captures a diagnostic dimension that is complementary to structural coverage and source-code reasoning, suggesting its potential both as a standalone technique and as an additional signal for hybrid fault localization. These findings indicate that runtime output semantics provide complementary diagnostic information beyond traditional structural spectra.

Several limitations of the current work warrant discussion. First, the current SSFL implementation does not instrument constructors and therefore cannot rank constructor-residing faults. Second, methods returning multi-dimensional arrays or complex opaque objects (such as AST `Node` objects in the Closure project) may not produce semantically meaningful `toString()` output, limiting the effectiveness of the Semantic Spectrum for such methods. Third, faults in void methods that produce no observable return value or mutable parameter-state change cannot be detected by the current implementation. Finally, the runtime trace files generated by Byteman can occupy substantial storage space, which may pose practical challenges when scaling to larger projects.

Future work will focus on several directions. First, we plan to turn the current semi-automated Byteman rule preparation into an end-to-end pipeline that automatically extracts method signatures, generates and validates rules, and deploys them for semantic spectrum construction and large-scale evaluation across all Defects4J versions. Second, we intend to explore hybrid approaches that combine SSFL's semantic spectrum with structural coverage information or LLM-based reasoning—for example, integrating SSFL as an additional input source in frameworks such as FlexFL—potentially leveraging the complementary strengths of multiple diagnostic dimensions. Third, we will investigate techniques for enriching the output representation beyond the current tokenization scheme, such as serialization or reflection-based state extraction, to capture richer semantic information from complex objects. Finally, we aim to explore the integration of SSFL with deep learning-based fault localization by using semantic frequency matrices as training samples for model training and defect identification. Unlike raw coverage vectors, semantic matrices encode richer behavioral information, potentially enabling models to learn more discriminative fault patterns. However, this direction requires a sufficiently large collection of labeled semantic matrices to train effectively—a requirement that cannot be met by the current Defects4J

benchmark alone. Scaling to larger datasets, such as Defects4J 2.0 or industrial codebases, will be a prerequisite for this learning-based extension.

Statements and Declarations

Funding. Funding information is provided in the separate title page to preserve double-anonymous review.

Competing interests. The authors have no relevant financial or non-financial interests to disclose. The funders had no role in the study design; data collection, analysis, or interpretation; preparation of the manuscript; or the decision to submit the work for publication.

Author contributions. Author contribution information is provided in the separate title page to preserve double-blind review.

Data availability. The replication package, including the experimental data, scripts, per-bug localization results, and complete LLM-generated Byteman rule files, is available through Google Drive at https://drive.google.com/drive/folders/1C5apXYfBbFudjJGLqQGUyARCNFBkbaqf?usp=drive_link. For reliable access and downloading, we recommend opening the link in Microsoft Edge or Google Chrome. A persistent DOI for the archived replication package will be provided after acceptance.

Code availability. The analysis scripts and instrumentation rules are included in the replication package described above.

Ethics approval and consent to participate. Not applicable.

Consent for publication. Not applicable.

Abbreviations

SSFL	Semantic Spectrum-based Fault Localization
SBFL	Spectrum-Based Fault Localization
IRFL	Information Retrieval-based Fault Localization
LLM	Large Language Model
MFR	Mean First Rank
MAR	Mean Average Rank
AST	Abstract Syntax Tree

References

Abreu R, Zoeteweij P, van Gemund AJC (2006) An evaluation of similarity coefficients for software fault localization. In: The 12th Pacific Rim International Symposium on Dependable Computing (PRDC). IEEE, pp 39–46

Cao H, Chu Y, Zhao C, et al (2023a) Software multi-fault localization via Chameleon clustering in parallel. Journal of King Saud University – Computer and Information Sciences 35(8):101676. https://doi.org/10.1016/j.jksuci.2023.101676

Cao H, Wang F, Deng M, et al (2023b) Software multiple-fault localization using particle swarm optimization via genetic operation. Journal of King Saud University – Computer and Information Sciences 35(4):21–35. https://doi.org/10.1016/j.jksuci.2023.02.023

Cao H, Wang F, Deng M, et al (2023c) Multiple fault localization based on ant colony algorithm via genetic operation. Journal of King Saud University – Computer and Information Sciences 35(8):101668. https://doi.org/10.1016/j.jksuci.2023.101668

Chen X, Sun T, Zhuang D, et al (2024) HetFL: Heterogeneous graph-based software fault localization. IEEE Transactions on Software Engineering 50(11):2884–2905

Fox EA, Shaw JA (1994) Combination of multiple searches. In: The Second Text REtrieval Conference (TREC-2), pp 243–252

Google Cloud (2025) Service control global outage: Incident report for 12 june 2025. Google Cloud Service Health. Available online: https://status.cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW, accessed: 26 July 2026

Just R, Jalali D, Ernst MD (2014) Defects4J: A database of existing faults to enable controlled testing studies for java programs. In: The International Symposium on Software Testing and Analysis (ISSTA). ACM, pp 437–440

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

Li X, Li W, Zhang Y, et al (2019) DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp 169–180

Lou Y, Zhu Q, Dong J, et al (2021) Boosting coverage-based fault localization via graph-based representation learning. In: The 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 664–676

Meng X, Wang X, Zhang H, et al (2022) Improving fault localization and program repair with deep semantic features and transferred knowledge. In: The 44th International Conference on Software Engineering (ICSE), pp 1169–1180

Motwani M, Brun Y (2023) Better automatic program repair by using bug reports and tests together. In: The 45th IEEE/ACM International Conference on Software Engineering (ICSE), pp 1225–1237

Qin Y, Wang S, Lou Y, et al (2025) SoapFL: A standard operating procedure for LLM-based method-level fault localization. IEEE Transactions on Software Engineering 51(4):1173–1187. https://doi.org/10.1109/TSE.2025.3543187

Rafi MN, Chen AR, Chen THP, et al (2024a) Exploring data cleanness in Defects4J and its influence on fault localization efficiency. In: The IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, pp 386–387

Rafi MN, Kim DJ, Chen AR, et al (2024b) Towards better graph neural network-based fault localization through enhanced code representation. Proceedings of the ACM on Software Engineering 1(FSE):1937–1959

Red Hat (2024) Byteman: A bytecode manipulation tool for java. https://byteman.jboss.org/

Sarhan QI, Beszédes A (2022) Effectively combining risk evaluation metrics for precise fault localization. Mathematics 10(21):3924

Vancsics B, Horváth F, Szatmári A, et al (2022) Fault localization using function call frequencies. The Journal of Systems and Software 193:111429

Widyasari R, Ang JW, Nguyen TG, et al (2024) Demystifying faulty code with LLM: Step-by-step reasoning for explainable fault localization. In: The IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 568–579

Wong WE, Wei T, Qi Y, et al (2008) A crosstab-based statistical method for effective fault localization. In: The 1st International Conference on Software Testing, Verification, and Validation (ICST). IEEE, pp 42–51

Wong WE, Debroy V, Gao R, et al (2014) The DStar method for effective software fault localization. IEEE Transactions on Reliability 63(1):290–308

Wong WE, Gao R, Li Y, et al (2016) A survey on software fault localization. IEEE Transactions on Software Engineering 42(8):707–740

Wong WE, Gao R, Li Y, et al (2023) Software fault localization: An overview of research, techniques, and tools. In: Handbook of Software Fault Localization Foundations and Advances. Wiley, p 1–117, https://doi.org/10.1002/9781119880929.ch1

Xu C, Liu Z, Ren X, et al (2025) FlexFL: Flexible and effective fault localization with open-source large language models. IEEE Transactions on Software Engineering 51(5):1455–1471. https://doi.org/10.1109/TSE.2025.3553363

Xu X, Debroy V, Wong WE, et al (2011) Ties within fault localization rankings: Exposing and addressing the problem. International Journal of Software Engineering and Knowledge Engineering 21(6):803–827

Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: The 30th IEEE International Conference on Software Maintenance

and Evolution (ICSME). IEEE, pp 191–200

Yan Y, Jiang S, Zhang Y, et al (2023) A fault localization approach based on fault propagation context. Information and Software Technology 160

Yang AZH, Le Goues C, Martins R, et al (2024) Large language models for test-free fault localization. In: The 46th IEEE/ACM International Conference on Software Engineering (ICSE), pp 154–165

Yoo S (2012) Evolving human competitive spectra-based fault localisation techniques. In: The 4th International Symposium on Search Based Software Engineering (SSBSE), Lecture Notes in Computer Science, vol 7515. Springer, pp 244–258

Zeng M, Wu Y, Ye Z, et al (2022) Fault localization via efficient probabilistic modeling of program semantics. In: The 44th International Conference on Software Engineering (ICSE), pp 958–969

Zhang J, Yue L, Zheng L, et al (2024) TS-FL: Software fault localization based on teacher-student network. In: The 2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, pp 599–604

Zhang Z, Lei Y, Tan Q, et al (2017) Deep learning-based fault localization with contextual information. IEICE Transactions on Information and Systems 100(12):3027–3031

Zhang Z, Lei Y, Mao X, et al (2019) CNN-FL: An effective approach for localizing faults using convolutional neural networks. In: The 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp 445–455

Zhang Z, Lei Y, Mao X, et al (2023) Context-aware neural fault localization. IEEE Transactions on Software Engineering 49(7):3939–3954

Zou D, Liang J, Xiong Y, et al (2021) An empirical study of fault localization families and their combinations. IEEE Transactions on Software Engineering 47(2):332–347