# Fine-grained Tracing for Performance Anomaly Diagnosis of Serverless Functions

RUNAN WANG, Department of Computing, Imperial College London, UK
GUANGBA YU, School of Computer Science and Engineering, Sun Yat-sen University, China
GIULIANO CASALE, Department of Computing, Imperial College London, UK
PENGFEI CHEN, School of Computer Science and Engineering, Sun Yat-sen University, China
ANTONIO FILIERI*, Department of Computing, Imperial College London, UK

Serverless function compositions subject to unpredictable faults are challenging to evaluate for root-cause analysis. Even though distributed tracing provides observations at multiple levels of granularity for troubleshooting, excessive code instrumentation increases the tracing overheads in terms of both computation and storage. Therefore, developers face the challenge of where and how to instrument serverless functions to maximize the likelihood of locating faults based on tracing data while minimizing tracing overhead and costs.

In this paper, we propose a methodology to instrument an application with code-level tracing to infer the location of faults, taking into account constraints in terms of the maximum cost of the instrumentation and testing. We encode the tracing probe placement based on the control flow graph of the application and devise heuristics-based tracing data collection strategies to relate possible probe placements with their ability to locate a fault. Then we train novelty detection models to identify the internal anomalies and present an enhanced global search algorithm that automatically computes a probe placement with optimal fault localization ability versus cost. Experimental results show high performance in locating single and multiple faults with over 90% recall score for up to 15% latency anomalies, with minimal instrumentation overhead.

Additional Key Words and Phrases: Serverless Functions, Tracing, Code Instrumentation, Anomaly Detection, Optimization

**ACM Reference Format:**
Runan Wang, Guangba Yu, Giuliano Casale, Pengfei Chen, and Antonio Filieri. 2026. Fine-grained Tracing for Performance Anomaly Diagnosis of Serverless Functions. 1, 1 (August 2026), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Distributed tracing has emerged in recent years as a technique to achieve observability in end-to-end applications with less effort and lower costs than traditional logging and monitoring tools [20]. Differently from monolithic applications, the source code of individual serverless functions is usually more focused and succinct with the coordination logic among the functions embedded in the code. The code-level analysis produces an ideal level of abstraction that emphasizes interaction among functions without including many low-level details appearing in the monolithic applications. Thus,

Authors' Contact Information: Runan Wang, runanwang@outlook.com, Department of Computing, Imperial College London, UK; Guangba Yu, yubg5@mail2.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, China; Giuliano Casale, g.casale@imperial.ac.uk, Department of Computing, Imperial College London, UK; Pengfei Chen, chenpf7@mail.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, China; Antonio Filieri, a.filieri@imperial.ac.uk, Department of Computing, Imperial College London, UK.


Manuscript submitted to ACM

in the context of serverless computing, distributed tracing can be used to extract the dependencies between different serverless functions and improve the detection of anomalies arising in their composition [24]. The technique leverages code instrumentation to record runtime data at varying levels of granularity, ranging from monitoring end-to-end request paths to observing individual code portions within a serverless function. Some instrumentation frameworks and libraries, such as OpenTelemetry [19], offer developers the ability to easily inject tracing commands in their codebase across a large range of programming languages.

Sampling-based tracing is a common approach in distributed tracing, which selectively collects tracing data instead of capturing each transaction in the system. However, from the perspective of developers, fine-grained tracing is particularly valuable for detailed performance issue diagnosis and a complete understanding of system interactions. First, fine-grained distributed tracing can provide detailed insights to the developers with comprehensive data, e.g., capturing the exact sequence of operations with their latencies and dependencies, which helps in understanding the root cause of issues. Second, fine-grained tracing offers precise metrics on how long each operation takes, which can be critical for performance tuning and optimization with profiling data. However, fine-grained distributed tracing is quite challenging for serverless functions.

Tracing probes are code snippets that specify which parts on execution should be monitored, thus defining the logical structure and content of the trace data records – called *spans*. A developer can design a distributed tracing that delivers observability into the performance of an application at the desired level of granularity for each of its parts. However, deciding on the appropriate placement of instrumentation probes may not be trivial. On the one hand, as code-level instrumentation of distributed tracing is tightly related to the internal logic and implementation of functions, there is still a lack of methods for developers to automatically instrument their code with tracing probes without manual operations and expert knowledge. On the other hand, distributed tracing comes with its own cost due to coding and testing time, as well as latency overheads in production. Additional computation cost is also induced by collection, processing, and storage of the new tracing data [8]. Thus, a *decision problem* arises concerning where to instrument the code and how many probes should be injected into the application.

To automatically compute a probe allocation that maximizes the chances of finding the location of faults that manifest as increased latency, while minimizing the instrumentation overhead, we propose a methodology in two phases.

First, we explore the probe allocation space to assess how instrumenting different possible locations impacts our ability to localize injected performance faults. Because the space of possible probe allocations is typically too large to be exhaustively explored, we introduce a novel heuristic that exploits the control flow graph of the application to iteratively decide where to explore next in order to gather the most information without exceeding a maximum budget of computational resources allowed for the exploration. For each explored probe placement, we run systematic performance tests to collect execution traces and use novelty-based anomaly detection methods to attempt to locate the injected fault from the collected traces.

Second, we formulate an optimization problem to decide which probes to retain in production in order to maximize the probability of locating a fault with a (near-)minimal set of probes and, therefore, minimal instrumentation latency overhead and costs. We use the traces collected during the previous phase to infer the ability of a probe placement choice to locate a fault. Primarily, we devise a tailored branch and bound search strategy that exploits the structure of the control flow graph and specialized meta-heuristics to sift through the otherwise intractably large search space efficiently.

In preliminary experiments, we abstract the anomaly behaviors inside functions with single and multiple delays to simulate the internal faults that impact the performance of serverless functions. The experimental results show good

accuracy on the different sensitivity levels of internal faults with over 90% recall score for up to 15% latency anomalies, and the enhanced search method achieves better efficiency compared to the original branch and bound strategy in locating the anomalies with minimal instrumentation overhead.

In summary, the main contributions are:

- A CFG-based encoding of tracing probe placement at the code level for fine-grained observations.
- Heuristics-based strategies to initialize the probe locations for tracing data collection.
- A global search with the branch-and-bound algorithm to elicit the best tracing probe placement for anomaly diagnosis, enhanced with search and bounding strategies to achieve fast convergence.

The rest of this paper is structured as follows. Section 2 presents the background knowledge for this paper. Section 3 provides an overview of the proposed method, and Section 4 presents the code-driven probe placement encoding and heuristics-based trace data collection. Section 5 demonstrates the novelty detection-based models for anomaly identification. In Section 6, we illustrate an enhanced branch-and-bound search method. Section 7 evaluates the proposed methodology with serverless tracing data. Section 8 briefly presents the state of the arts, and section 9 discusses the threats to the validity of this work. Section 10 draws the conclusion of this paper.

## 2 Background

In this section, we briefly recall necessary background concepts and a motivating example to intuitively qualify the trade-off of optimal instrumentation for diagnosing anomalies.

### 2.1 Distributed tracing.

Distributed tracing provides observations of requests for services that propagate through distributed computing environments. It is critical to complex cloud-native applications, for example, in the context of microservice [12, 27] and serverless architectures [24] for health checks, performance issue diagnosis and debugging. Distributed tracing is enabled by metadata context propagation to collect informative data about the monitored system.

*Trace* and *span* are core concepts in distributed tracing to understand the behaviors of applications. A trace provides a high-level description of how the request passes through various services that can communicate with each other in an application. A complete execution path of each transaction is constructed by a trace with all operations needed to be executed. A unique identifier defines each trace and consists of one or a group of spans. A span is a representation of a logical operation involved with processing a request or transaction. Spans can expose the execution metadata of current working units, such as the starting and ending timestamps, the execution status, operation names, etc. Spans can also be nested to represent constituent sub-operations that enable an arbitrarily fine-grained capture of the orders of processed procedures. The relationship between nested spans can be depicted in a hierarchical parent-child structure, where a parent span is broken down into (possibly overlapping) segments by its children spans.

A trace starts with a root span referring to the initial operation of the transaction, and it can be conveyed as a tree structure to correlate the composed spans with context propagation.

Tracing data can be generated with either black-box approaches or annotation-based systems [25]. Black-box distributed tracing tools take the instrumentation control from the users and offer tracing agents to be set up on their infrastructure. In comparison, annotation-based tracing schemas rely on the instrumentation of application code with common libraries to sniff and preserve customized tracing information.

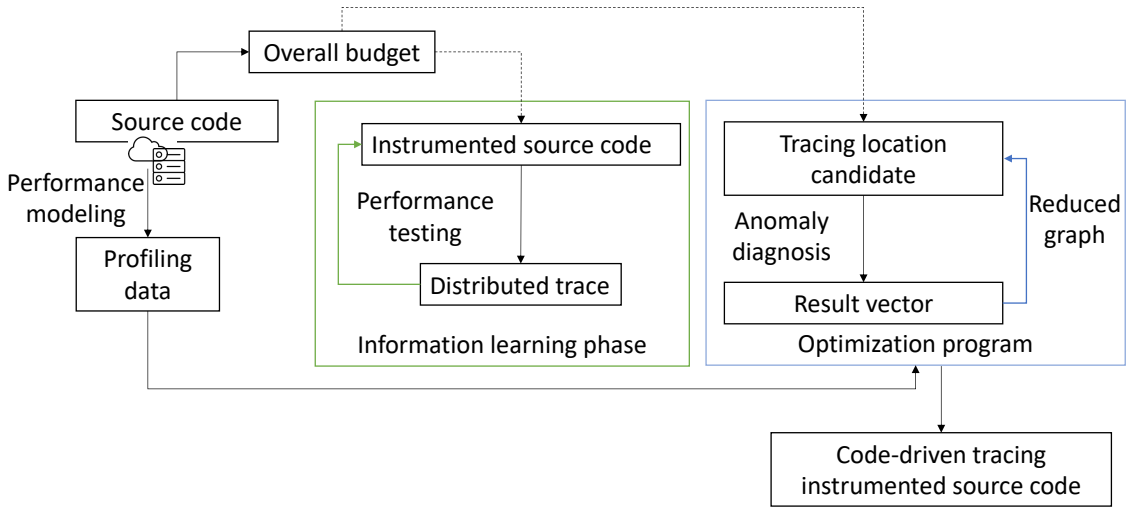



Fig. 1. Overview of the proposed methodology

### 2.2 Static analysis and control flow graph.

Reasoning about the structure and features of application code makes it easier to understand the internal logic and decision actions behind program executions. Static analysis is widely adopted to infer information about a program without actually executing the program [17]. Compared to dynamic analysis [3], static analysis provides a fast way to examine large codebases for the presence of possible behavior patterns, e.g., corresponding to security vulnerabilities. By not executing the code, static analysis reasons mainly on structural aspects extracted from the code, which usually represent an overapproximation of the possible behaviors of the application, i.e., what the application may potentially do, which may be a superset of what it will actually do in a specific execution context. For example, possible control flows can be statically extracted via code analysis to represent the order in which the procedures in the program may be executed. In static analysis, the code is usually abstracted into an intermediate representation, like the abstract syntax tree (AST) [22], from which control flows are reconstructed.

All the possible paths (a superset of the feasible paths) that could be executed at runtime can be represented in a directed graph notation as a control flow graph (CFG). A CFG represents how the evaluation of conditional statements (e.g., branches and loops) determines the next code block to be executed. CFGs depict the dependencies between code blocks and all the possible execution orderings, subject to the decisions at conditional nodes.

## 3 Methodology overview

The ultimate goal of our method is to decide the locations where instrumentation probes should be injected into the production code to minimize the overhead and cost of monitoring while achieving high fault localization capabilities.

To achieve this goal automatically and efficiently, without relying on human expertise, we propose a systematic experimentation and optimization process involving three parts: 1) construction of an extensive model of the performance profile of the application in normal operating conditions, i.e., assuming no faults, 2) inject faults and efficiently collect monitoring traces that can help ascertain the fault localization capability of different instrumentation locations, and 3)

optimize the set of instrumentation probes to embed in the production code to minimize the overhead for the users while retaining as much fault localization ability as possible. Figure 1 gives an overview of our proposed methodology.

### 3.1 Code abstraction and normal operation

A critical insight underlying our methodology is the abstraction of the functions composing the application into their CFG [17]. We assume that the code of (most) serverless functions is accessible either from the cloud or from a private server and preliminarily compute its aggregated CFG that captures all the possible executions of the application. While a CFG is usually an overapproximation of the possible behaviors (i.e., it may also include execution paths that may never actually execute with the inputs received by the application), it is efficient to compute statically from the source code. It provides a compact structure to reason about the possible execution paths within the application. In particular, as we will elaborate in the next section, the branches in the CFG partition all the possible executions in segments, such that each execution path is uniquely identified by the sequence of segments it exercises.

To complete a reference model of the regular operation of the application, which will be later used for anomaly detection, we perform an initial systematic performance test. We profile the current version of the application to collect trace data without faults to estimate the execution time of code blocks on different paths exercised by a reference workload. Typically, intense performance testing for profiling purposes should be conducted on a shadow production deployment – i.e., resembling the execution resources of the production environment to collect realistic performance metrics while not affecting the quality of service experienced by external customers.

### 3.2 Code-driven tracing data learning

This phase (green box in Figure 1) consists of injecting faults in the application and learning the fault localization power of different instrumentation choices based on the collected execution traces.

To collect such traces, we need to decide how to instrument the code. This instrumentation has different goals than the final instrumentation of the production code our approach will produce as output: we aim at collecting the most informative tracing information that will later help us decide the minimal set of probes to retain in production. Therefore, we will not try to minimize the number of probes; instead, we aim to collect traces useful for discriminating at different levels of granularity the location of the injected faults.

Because each instrumentation and injected fault requires repeating the performance tests, it is usually intractable to exhaustively try all the possible instrumentation options. Additionally, as we shall demonstrate throughout the paper, a relatively small number of locations provide generally enough data to detect an anomaly and localize the corresponding fault within reasonably small portions of code.

We instead propose in Section 4 an iterative exploration of possible probe locations to evaluate based on a heuristic search guided by the CFG structure. To prevent an excessive use of computational resources and time for the testing, we require the user to set an overall budget constraint $M$ that represents the cost (e.g., in dollars) of the total resources available for this phase. These resources typically include the additional CPU time directly used by the instrumentation probes and tracing operations and the cost of generating the test workload.

In abstract terms, the problem we aim to solve can be summarized in the following optimization program

$$\begin{aligned} \max_{n,k} \quad & \log n + \log k \\ \text{s.t.} \quad & k\, u_{app} + n\, u_{probe} \leq M \\ & n, k \geq 1 \end{aligned} \tag{1}$$

where $n$ is the number of probes injected into the code, and $k$ is the number of executions of the baseline application. $u_{app}$ is the cost of a run of the un-instrumented application, including the cost of generating the test requests, and $u_{probe}$ is the average cost of a probe. The objective function is concave and equivalent to maximizing the product $nk$.

The optimization program (1) is just a conceptual formalization of the goal of this phase. In practice, the overall budget is constrained across multiple iterations, which will be discussed in the next section. As mentioned above, in this phase, we aim to maximize the number of probes. On the contrary, we want to use the entire budget – without exceeding it – to gather as many traces as possible while deciding the instrumentation to test at each iteration based on the information we collected before and a search heuristic guided by the CFG.

### 3.3 Optimal probe placement

The last phase of our method (blue square in Figure 1) uses the set of traces $\boldsymbol{d}$ collected in the learning phase and an indication of the maximum number of probes $n$ used to collect them to produce the final probe placement to be retained in production.

Contrary to the learning phase, whose main goal was exploration and information gathering, this phase aim at minimizing the set of probes in the final instrumentation while preserving the most anomaly detection and fault localization capabilities possible. This optimization problem relies only on the set of traces $\boldsymbol{d}$ and does not require additional performance tests. We formulate it in Equation (2):

$$f(n, \boldsymbol{S}, \boldsymbol{d}) = score(\boldsymbol{S}, \boldsymbol{d}) - cost(n, \boldsymbol{S}) \tag{2}$$

where $\boldsymbol{S} = (S_1, \ldots, S_m)$ denotes the space of possible location combinations to inject tracing probes, where each $S_i$ represents a set of code locations to instrument and $i \neq j \implies S_i \neq S_j$. The detailed encoding of $S_i$ is discussed in Section 4.1. $\boldsymbol{d}$ is the dataset of traces collected from the previous phase and $n$ the maximum number of probes used during learning, which is used to limit the maximum number of probes enabled in each $S_i$. The identification *score* and *cost* functions relate to the expected fault localization capability and overhead of the final instrumentation and will be discussed in Section 5.

## 4 Code-driven tracing data learning

In this section, we present our CFG-based heuristic for the systematic exploration of the probe placement space to collect traces from which we will estimate the fault localization power of different probe locations. To inform the decision problem of where to instrument the code, we first extract the function structure as a control flow graph (CFG) via static code analysis. A CFG can be represented with a graph notation, $CFG = (\boldsymbol{N}, \boldsymbol{E})$, where $\boldsymbol{N}$ and $\boldsymbol{E}$ denote the nodes and edges, respectively. Nodes in the CFG represent possible locations in the code where probes can be placed, while edges represent partial orderings of the locations executed during a run corresponding to branch instructions in the code.

The CFG of each serverless function can be generated by applying static analysis to the source code. To construct the complete CFG for the application, the orchestration is required to connect each single CFG with their calling relationships. This can be achieved by either exploring metadata or utilizing trace context propagation to identify the dependencies between functions during the performance testing phase.

To align distributed tracing data with the CFG structure, we assume that when a probe is placed at a location $l$ within a function, the corresponding span starts when the location is executed and ends when the function returns. Such spans are nested under the root span that monitors that function.

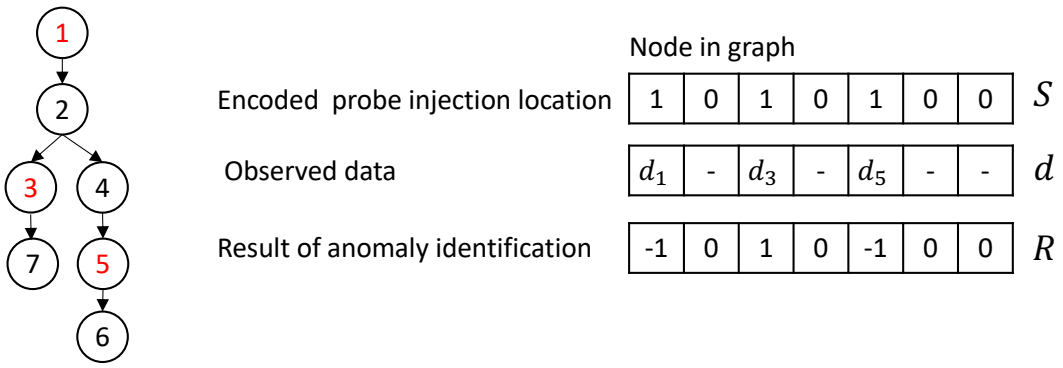



Fig. 2. Example of encoding of trace probe location with the underlying CFG

### 4.1 Code-level tracing probe encoding

Figure 2 shows an example encoding of probe locations from a CFG. The left-hand side represents an example control flow, where node 1 indicates the entrypoint of the function, and nodes 6 and 7 are possible return points. After node 2, the execution may branch towards either 3 or 4, depending on the inputs. We highlight in red the nodes in the example CFG where probes are placed. The top vector on the right-hand side shows a binary encoding of the instrumented location, where $S_i = (1, 0, 1, 0, 1, 0, 0)$ indicates that probes are placed at locations 1, 3, and 5. The observed data $d_1$, $d_3$, and $d_5$ represent the execution time for the spans starting with the probes at the corresponding locations and ending with the return of the function.

Anomaly identification in this work is based on a novelty detection method that tests the observed data against the expected values under normal operating conditions and produces a vector $R$ that assigns $-1$ to the data classified as an anomaly and 1 as data matching the expectation. 0 represents infeasible classification for execution segments not instrumented for the current set of observations. We will detail our novelty detection method in Section 5.

### 4.2 Heuristics-based data collection

The set of traces gathered during each iteration of performance testing depends on the probes used to instrument the code for that iteration. Because the number of possible locations may be intractably large and the process may be repeated for multiple fault injection, it is usually intractable to instrument every location of the code within a reasonable overall computational budget $M$.

Instead, we exploit the structure of the CFG to devise a heuristic probe placement to narrow down as much as possible the possible locations of an injected fault. We achieve this by injecting probes at the successors of each branch node. When a probe is injected at the location corresponding to a node of the CFG, we assume its monitoring span to extend until the function returns, i.e., it measures the execution time from the node to the first return node encountered during the execution.

We define $\boldsymbol{\sigma} = \{\sigma_1, \sigma_2, \ldots\}$ as a set of all CFG nodes that are successors of a branch. For example, in Figure 3, the blue nodes of the CFG on the left are included in $\boldsymbol{\sigma}$ since they are the successors of branch nodes – namely, of 2 and 4.

Intuitively, each branch splits the execution paths that reach it in two disjoint subsets. We can thus use them as natural classification predicates to identify the subgraph of a CFG where a fault is located. However, not all splits carry

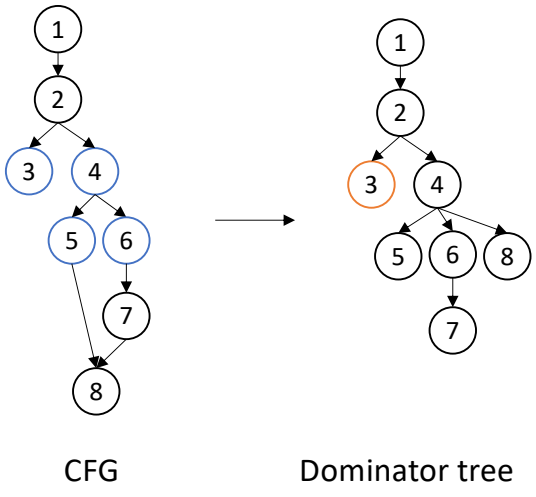



Fig. 3. Example of CFG and corresponding dominator tree

the same localization power. Consider the CFG on the left-hand side of Figure 3. If a fault has been injected in the code under node 3 (successor of 2), placing a probe on node 3 will very specifically localize the fault. If a fault is instead placed, e.g., under node 5, instrumenting node 4 will provide less specific information about the location of the fault (which could be in any node $\geq 4$. In other words, the fewer further execution splits of the execution are possible after a branch successor, the more specific the localization information obtainable instrumenting that successors are.

We can formalize this intuition by constructing the dominator tree of the CFG. A dominator tree is defined as a directed graph structure where each node immediately dominates its successors [5]; a node $n_i$ dominates a node $n_j$ if every path from the entrypoint of the CFG to $n_j$ must go through $n_i$. Given a CFG, the corresponding dominator tree $dTree$ can be efficiently constructed using the Lengauer-Tarjan algorithm [11]. We can then sort the nodes in $\sigma$ in ascending order based on the maximum distance from a leaf in $dTree$. For example, node 3 and node 5 have a distance of zero, while node 4 has a distance of two, as expected from the discussion above.

Putting it all together, let us assume a fraction $0 < p < 1$ of the overall learning budget $M$ is allocated for the current iteration, which allows placing a maximum of $n$ probes, then:

(1) if $|\sigma| = n$, we instrument all the locations of nodes $\sigma$
(2) if $|\sigma| > n$, we instrument the locations of the first $n$ nodes in $\sigma$, after sorting them according to their maximum distance from a leaf in the dominator tree of the CFG, as discussed above
(3) $|\sigma| < n$, we instrument all the locations of nodes in $\sigma$ and then select $n - |\sigma|$ additional location to instrument based on two secondary heuristics described below.

### 4.3 Residual budget allocation.

If $|\sigma| < n$, the budget constraints allows to instrument an additional set of $r = n - |\sigma|$. Recall that our goal in this phase is to gather as much information as possible, provided we do not exceed the budget. We thus consider selecting $r$ additional nodes either 1) randomly from the CFG or 2) using a bisection strategy on a sample of collected paths that revealed an anomaly.

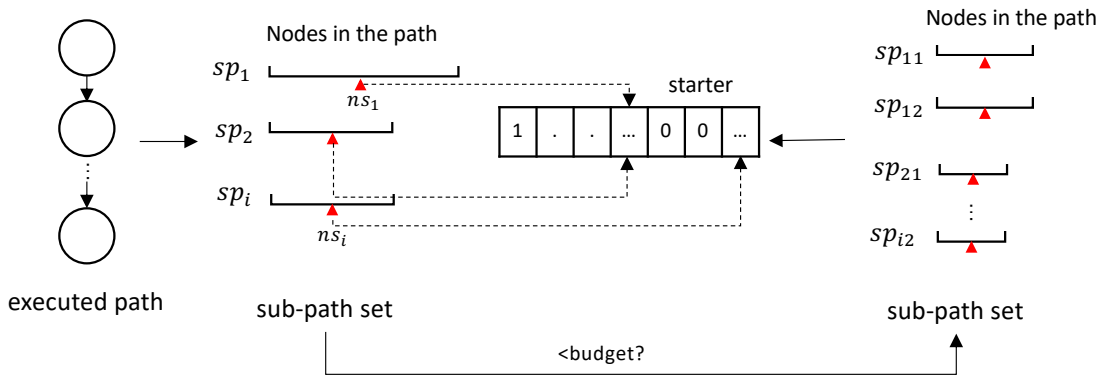



Fig. 4. Bisection sampling on feasible paths

The bisection strategy works as follows. Assume in a previous iteration, we detected an anomaly along an execution path $\omega$ through the nodes of the CFG. We can partition $\omega$ into sub-paths $sp = (sp_1, sp_2, \dots)$ based on the branch successors that were instrumented, i.e., each $sp_i$ is a sequence of non-instrumented nodes between elements of $\boldsymbol{\sigma} \cup \top$, where $\top$ represent return statements of the function. Then, we can iteratively find the middle of each segment $sp_i$ and add an instrumentation probe at its location. This process is akin to a binary search for the location of a fault within each segment and is exemplified in Figure 4. We compare these two strategies experimentally in Figure 5, and discuss the results in the evaluation section.

### 4.4 Budget distribution across iterations.

The conceptual model in the optimization program (1) does not prescribe a specific distribution of the overall learning budget $M$ across iterations. In fact, the heuristics described in this section also apply in case a single iteration is conducted. Nevertheless, it may not be easy to estimate the costs of instrumentation probes $u_{probe}$ and of performance testing of the baseline application $u_{app}$. For example, the cumulative CPU time consumed by a probe statically placed at a specific code location depends on how often such location is exercised by the test workload (e.g., the number of iterations of a loop). A more conservative strategy to avoid exceeding the overall budget limit $M$ would be to allocate a fraction $0 < p < 1$ of the available budget to the next iteration. After the iteration, both the available budget can be reduced by the actual amount used, and the estimates of $u_{app}$ and $u_{probe}$ can be refined with the new observations.

### 4.5 Pseudocode.

Algorithm 1 summarizes the heuristic instrumentation and trace collection process described in this section. Notice that we always include a probe at the entrypoint of a function (node 1; line 1), which will account for the total execution time, i.e., until return. The function `heuristic` at line 8 also includes the chosen secondary node selection heuristic for the situations where $n_i$ is larger than the number of branch nodes $\boldsymbol{\sigma}$ in $\boldsymbol{N}$. The algorithm implemented by the function `detectAnomaly` at line 11 will instead be described in the next section.

**Algorithm 1** Heuristic trace data collection

**Input:** $CFG = (\boldsymbol{N}, \boldsymbol{E})$,
$M$ overall budget limit,
$0 < p \leq 1$ fraction of budget per iteration,
$u_{app}$ and $u_{probe}$ cost estimates
**Output:** $\boldsymbol{d} \leftarrow$ Collected traces

Initialize $S \leftarrow (1, 0, \ldots, 0)$
Initialize $R \leftarrow (0, 0, \ldots, 0)$
Initialize $r \leftarrow M$
Initialize $\boldsymbol{d} \leftarrow \emptyset$
**while** $r > 0$ **do**
  $r_i \leftarrow r \cdot p$
  $n \leftarrow estimatedMaxProbes(r_i, u_{app}, u_{probe})$
  $S \leftarrow heuristic(\boldsymbol{N}, R, n)$
  $\boldsymbol{d_i} \leftarrow$Collect observed data with $S$
  $\boldsymbol{d} \leftarrow \boldsymbol{d} \cup \boldsymbol{d_i}$
  $R \leftarrow detectAnomaly(S, \boldsymbol{d})$
  $r \leftarrow r -$ actual cost of the iteration
**end while**
**return** $\boldsymbol{d}$

## 5 Anomaly identification

We use a novelty detection strategy to localize the likely location of a fault leading to a performance anomaly. This strategy compares the performance metrics associated with a node in the CFG under normal operation conditions – i.e., no-fault injected – with its metrics after a fault has been injected. The performance metric we focus on is the execution time, where instrumenting a node $i$ corresponds to estimating the time between the execution of the node and the function's return. After a fault has been injected, or it later occurs in production, its localization can be formulated as a novelty detection problem to recognize new patterns or outliers that deviate significantly from the profiling data.

### 5.1 Feature extraction.

Let **P** be the set of traces through the CFG exercised during performance testing. Similarly to the procedure described in Section 4.2 for the bisection heuristics, we can see a path $\omega \in \mathbf{P}$ as the concatenation of segments $(sp_1, sp_2, \ldots)$ that share at most one instrumented node. The execution time of a segment of an execution trace can be computed as difference of the execution times computed at its extremes: recall that the instrumentation of a node spans from the node to the function's return and that the entrypoint of a function is always instrumented. Different paths may share segments.

Let $i$ and $j$ be the start and the end nodes of a segment $sp$. The dataset of traces $\boldsymbol{d}$ computed during the tracing data learning phase can be processed into a set of tuples $(i, j, time(i) - time(j))$. We assume that for each segment exercised during the trace learning phase, the corresponding time duration under normal conditions (no fault injected) has also been estimated.

### 5.2 Novelty detection models.

Using the performance data taken under normal conditions and baseline, novelty detection can be framed as a one-class classification (OCC) [10] classification problem. We evaluate four different methods for this task: distance-based k-means

clustering (*kmeans*) [26], kernel-based one-class support vector machines (*ocsvm*) [9], isolation forest (*iforest*) [14] and deep learning-based autoencoders (*AE*) [16]. Each of these models is trained on the same dataset of segment performance under normal conditions, and we use the following thresholds $\eta$ to decide whether an estimated duration for a segment should be considered an anomaly:

(1) for the *kmeans* models, we calculate the distance between each data point to the centroid of its cluster and set the threshold as the $\eta^{th}$ percentile of the distance.
(2) for *ocsvm* and *iforest*, we compute the distance of each training point to the decision boundary and the average path length across isolation trees, respectively. The threshold value is defined with the $\eta$ interquartile ranges of the maximum and minimum quartile of the distances and path lengths.
(3) for trained *AE*-based models, training loss $\mathcal{L}$ and reconstruction error ($\epsilon$) measured with the mean squared error is used to decide the anomaly label for test data by comparing $\epsilon > \mathcal{L} + \eta$.

### 5.3 Anomaly identification scores.

The result of anomaly detection can be formalized a vector $R$ whose entries assign to the corresponding nodes of the CFG the values -1, 1, or 0 if the execution time from the node to the return of the function is classified as anomaly (-1), normal (1), or the node cannot be classified due to lack of observations (0); the latter case corresponds to a non-instrumented node (or a node never exercised during performance testing, which could signal a limitation in the workload generation).

If a node is classified as abnormal, the fault is likely to be located along a path originating from the node. Looking at the partial order among instrumented nodes (and the corresponding path segments), it is often possible to narrow down the location of a fault. For example, consider again the CFG and anomaly detection result $R$ in Figure 2. The instrumented nodes are 1, 3, and 5. Because node 1 is anomalous but node 3 is normal, the fault is likely to be located along the path between node 1 and node 6. Because node 5 is also anomalous, the most likely location of the fault is between nodes 5 and 6.

The maximum localization resolution can be obtained only when a node is classified as anomalous, and all its immediate successors are also instrumented and classified as normal. In this case, the fault is likely related to the operation performed at the code location corresponding to the anomalous node. However, excessively rewarding such high resolution in the search process may lead to overfitting on instrumenting around the locations where faults have been injected during the trace collection phase. This may be desirable in domains where the developers can confidently restrict the possible causes of a performance anomaly (e.g., invocations of known expensive remote functions).

However, one may prefer a wider coverage of the CFG, albeit with possibly lower resolution, as allowed by the limit on the instrumentation overhead. For this purpose, we classify a node as *suspicious* if it *may be* the location of a fault, as opposed to (strictly) *anomalous*, i.e., flagged as anomalous and whose successors have also been instrumented and flagged as normal. In the example of Figure 2, nodes 1 and 5 would be suspicious. We can then refine the `score` function in Equation (2) as the weighted sum:

$$score(R) = w_a \sum_i AS_a(R) + (1 - w_a) \sum_i AS_s(R) \tag{3}$$

where $AS_a(R)$ counts how many nodes can be identified as strictly anomalous with a candidate probe placement $S$ and $AS_s(R)$ counts how many nodes can be identified as suspicious, given the same probe placement. The weight $0 \leq w_a \leq 1$ allows for a trade-off of the resolution of the detection versus coverage of the CFG. The goal of the optimization phase will thus be to infer from the dataset of traces collected in the previous phase which probe placement will maximize

score while minimizing the instrumentation cost. In the next section, we present our strategy to efficiently solve such an optimization problem.

## 6 Enhanced search method for probe location inference

Given the dataset of traces collected during the iterative trace learning phase, the optimization problem of deciding a probe placement that minimizes overhead and maximizes anomaly identification can be solved by search-based methods. Using the anomaly detection score defined in Equation (3), the original optimization problem in (2) can be rewritten as:

$$\underset{S \in \boldsymbol{S}}{\operatorname{argmax}} f(S, R) \tag{4}$$

$$f(S, R) = score(R) - u_{probe} \sum_{i} [S \neq 0] \tag{5}$$

i.e., we aim at finding the minimum-size set of probes that maximizes the detection score, based on the desired resolution vs coverage trade-off.

For a CFG with $n$ nodes, the search space of possible probe placements has cardinality $|\boldsymbol{S}| \approx O(2^n)$. Optionally, such a space may be reduced by bounding the cost component in Equation (5), but it would, in general, remain intractable for exhaustive search. Instead, we propose in the remaining of this section a branch and bound strategy enhanced by exploiting the structural information in the CFG.

**Branching exploration.** The search strategy starts from the initial probe allocation, which instruments only the entry point of the CFG. At each branching step, the unexplored children of currently instrumented nodes are considered for additional probe allocations. Without any bounding strategy, this basic exploration process would search through all the possible probe allocations

in a breadth-first order through the CFG nodes, which is usually an intractably large search space.

**Tabu meta-heuristic.** The information in the CFG can be used to build a graph-based Tabu list [1] to prevent the repetitive exploration of nodes through the search process. The core idea is to combine the information in the anomaly detection error with the dominance of nodes in the CFG. Suppose the $j$-th element of the anomaly detection vector $R$ for an explored probe placement is flagged as normal ($R[j] = 1$). All nodes dominated by node $j$ should themselves be normal and, therefore, are added to the Tabu list ($TL$) to avoid exploring nodes that cannot contribute to increasing anomaly scores – recall that a probe in node $j$ accounts for the execution time of an execution path from $j$ to a return of the function, thus a delay at any point along any such paths would also be observed by the probe in $j$. Nodes added to $TL$ are excluded from future probe allocation exploration, thus incrementally reducing the search space and accelerating the convergence of the process.

**Bounding strategy.** Besides incrementally pruning the search space with the Tabu list, we bound the search exploration by keeping track of the current local optimal scores at any explored node. At each iteration $i$ of the search process, the possible child branches to explore are computed starting from the current probe allocation candidate $S_i$. The function $f(\cdot, \cdot)$ (Equation (5)) of each child is then compared with the score of its parent, and if lower, the child is excluded from further branching. This strategy helps to bound the exploration, focusing on children that can improve the value of the current partial allocation.

**Enhanced branch-and-bound search algorithm.** Algorithm 2 summarizes our branch-and-bound strategy. The initial candidate $S_0$ corresponds to the allocation of a probe to the entrypoint of the CFG: $S_0 = (1, 0, \dots)$ (line 5). We push this candidate to the queue $\boldsymbol{\nu}$ of candidates to explore (line 6) and register it as the initial best candidate $\boldsymbol{\theta}$. The

**Algorithm 2** Enhanced branch-and-bound search

**Input:** $CFG \leftarrow$ control flow graph
$n \leftarrow$ maximum number of probes to place
$\boldsymbol{d} \leftarrow$ collected observation traces dataset
**Output:** $\boldsymbol{\theta} \leftarrow$ Tracing probe placement vector

1: Initialize $\boldsymbol{\nu} \leftarrow queue()$
2: Initialize $\boldsymbol{\theta} \leftarrow S_0$
3: Initialize $best \leftarrow -\infty$
4: Initialize $TL \leftarrow \emptyset$
5: Initialize $S_0 \leftarrow (1, 0, \ldots, 0)$
6: $push(\boldsymbol{\nu}, S_0)$
7: **while** $\boldsymbol{\nu}$ is not $\emptyset$ **do**
8: $S_i \leftarrow pop(\boldsymbol{\nu})$
9: $R_i \leftarrow detectAnomaly(S_i, \boldsymbol{d})$ // novelty detection
10: $TL \leftarrow tabu(CFG, R_i, TL)$ // update Tabu list
11: $LB \leftarrow f(S_i, R_i)$ // current node value
12: **if** $LB \geq best$ **then**
13: $best \leftarrow LB, \boldsymbol{\theta} \leftarrow S_i$
14: **end if**
15: $\boldsymbol{C} \leftarrow child(CFG, S_i, n, TL)$ // generate candidates
16: **for** $C_j \in \boldsymbol{C}$ **do**
17: $R_j \leftarrow detectAnomaly(C_j, \boldsymbol{d})$
18: **if** $f(C_j, R_j) \geq LB$ **then**
19: $push(\boldsymbol{\nu}, C_j)$
20: **end if**
21: **end for**
22: **end while**
23: **return** $\boldsymbol{\theta}$

iterative search process is within the loop at line 7. The first candidate $S_i$ is popped from the queue $\boldsymbol{\nu}$, the novelty detection algorithms are used to compute $R_i$, and the Tabu list is updated. The value of the candidate $S_i$, $f(S_i, R_i)$, is computed and stored as the lower-bound $LB$ to be used to bound the exploration of possible child branches from $S_i$; if $LB$ exceeds the current $best$ known value, we update the best-known value and the corresponding candidate (line 13). Children of $S_i$ to be explored are generated at line 15, provided the budget $n$ is not exceeded and excluding nodes in the Tabu list $TL$. Each child is evaluated according to the objective function $f(\cdot, \cdot)$ and only those with a value greater or equal to their parent's $LB$ are pushed into the working queue (loop at line 16). When no more candidates are left to explore, the loop exits, and the algorithm returns the best-explored probes allocation $\boldsymbol{\theta}$.

## 7 Evaluation

### 7.1 Experimental setup

We deployed an OpenFaaS [18] serverless platform on our local Kubernetes cluster, with two master nodes and ten worker nodes. Each node has an 8-core 2.10GHz CPU, 16GB memory, and runs with Ubuntu 18.04 OS. For each serverless function, we instrument the trace logic based on the OpenTelemetry framework [19], which is a vendor-neutral open-source Observability framework for instrumenting, generating, collecting, and exporting trace data. We

Table 1. Dataset under evaluation

| | Dataset | Fault injection level | Number of traces |
|---|---|---|---|
| Single fault | *SF*30 | 30% | 2399 |
| | *SF*15 | 15% | 2417 |
| | *SF*3 | 3% | 2399 |
| Multiple faults | *MF*30 | 30% for each fault | 2419 |
| | *MF*15 | 15% for each fault | 2373 |
| | *MF*3 | 3% for each fault | 2463 |

use the OpenTelemetry Collector [7] to collect traces of functions and persist them in the Grafana Tempo [1]. We generate function workloads based on the Locust[2] framework from one separate node in the same cluster.

We created an image-resizing serverless function on OpenFaaS to evaluate the proposed prototype. The CFG of the resizing function includes 12 CFG nodes with sequential and branching execution orders, where the tracing data is referred to as *RS-Data*. The summary of the dataset under evaluation is presented in Table 1. Both single fault and multiple faults are taken into account to verify the performance with 6 datasets as shown in Table 1. We also conduct experiments with a serverless workflow involving sequential serverless function calls. This workflow comprises an image processing function and a deep learning-based image classifier, utilizing the pre-trained MobileNet model with PyTorch and running with ONNXRuntime from [23]. The CFG of the serverless workflow is reconstructed from the CFG of each individual function and connected according to the calling order of the workflow topology, including 19 nodes in total. The workflow dataset is denoted as *RS-WF-Data*. Different levels of faults are injected into the function to simulate the internal anomalies, where we consider [30%, 15%, 3%] of the original function mean execution times as the synthetic latency anomalies.

**Evaluation metrics.** We focus on evaluating the effectiveness and accuracy of the probe placement result rather than the instrumental overhead since it is negligible in the experimental analysis of the proposed methodology on the serverless function.

To evaluate the performance of the proposed method, all the training data (without internal faults) are labeled with 1, and the testing data are labeled as -1 for anomalies and 1 for normal data. The ground truth of the best probe placement locations for each dataset in Table 1 is calculated based on the identification index in Equation (5). To ensure that the identification index for comparison has positive values, we scale the score in Equation (5) by multiplying it by 3500.

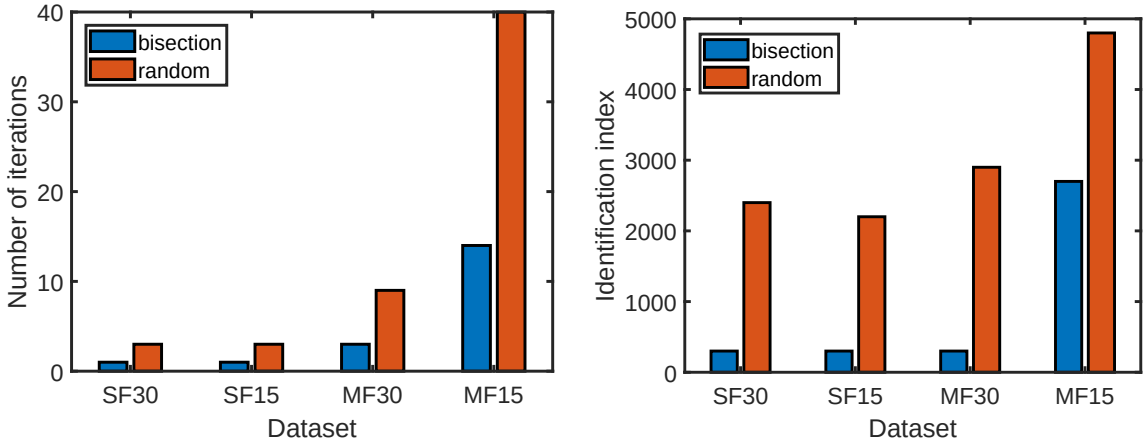
We use the following metrics in the experimental comparison. For the performance of the tracing probe placement, we compare the number of *iterations* to obtain the solution for efficiency and the *identification index* to show effectiveness. We also specify a *relative score* to evaluate the performance of the probe placement inference to the ground truth among 4 datasets as

$$rs = \frac{f^{'} - f}{f} \tag{6}$$

where $f^{'}$ is the identification index of the solution from search algorithms, and $f$ is computed from the best locations in the ground truth.

In the analysis of novelty detection with different sensitive levels, *recall*, which is known as sensitivity to identify positive samples (abnormal observations) is applied.

[1] https://grafana.com/docs/tempo/
[2] https://locust.io/




(a) Iterations (b) Identification index

Fig. 5. Comparison results on heuristics-based starter strategies

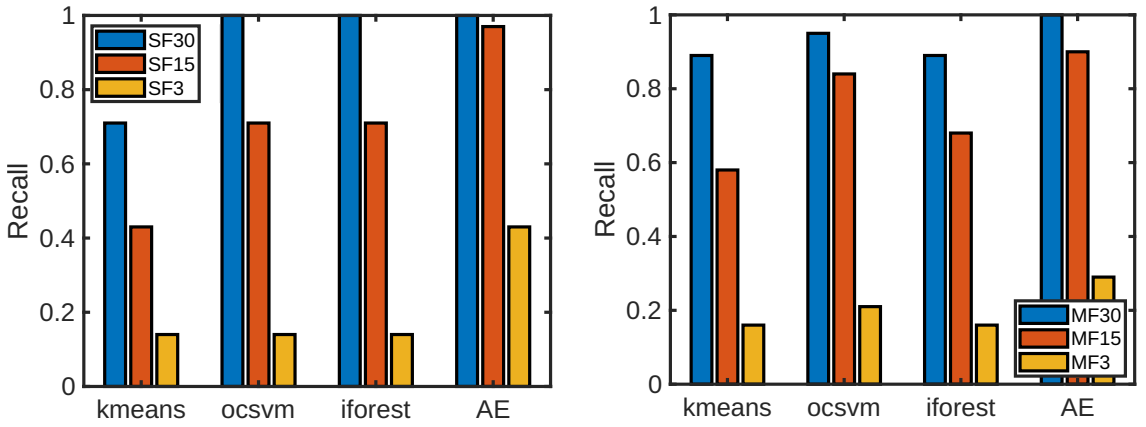
In the following subsection, we analyze the experimental results in relation to the following research questions: 1) How do different strategies for heuristic generation of probe locations impact efficiency and diagnostic performance? 2) How do novelty detection methods perform at different sensitivity levels? 3) How does the proposed method with enhanced BB search perform under different budget provisions?

### 7.2 Result analysis

First, we compare the performance of different heuristic probe location generation strategies. Next, we conduct experiments with different novelty detection models on different levels of internal faults to assess the sensitivity of anomaly identification. Finally, we analyze the experimental results for code-level probe inference with the enhanced BB algorithm under two demanding conditions: limited budget and sufficient budget.

*7.2.1 Strategies for heuristic generation of probe locations.* In Section 4.2, we propose random selection-based and static bisection-driven strategies to generate heuristic starting locations for tracing data collection at the code level. We here compare the number of iterations of the optimization program to converge and the anomaly identification index value with different heuristics in Figure 5. It can be seen in Figure 5a that with static generation strategy (blue bars), the optimization program terminates with fewer iterations compared to the random selection on all the datasets under evaluation. It only takes 1 iteration to complete the program for $SF15$ and $SF30$ with the bisection-based generation of probe locations. This is because no more search candidates can be generated based on the current tracing locations and observations by following the strategies in the enhanced search algorithm.

However, within the budget, static generation of the heuristic starting locations cannot reach the same identification index value as the random heuristics, as shown in Figure 5b. While across different datasets, the random-selection-based method achieves higher values than bisection-based generations, indicating better tracing probe placement to locate the faults. This is because the solution from the optimization program based on different heuristic strategies not only depends on the graph structure but also relates to the actual fault location in the functions. Static generation limits the exploration of the underlying CFG. However, random selection is more flexible, as it increases the opportunities to hit




(a) Single fault (b) Multiple faults

Fig. 6. Detection results for different OCC models. AE-based models achieve the best recall score among all OCC models.

the possible locations of anomalies. Intuitively, for simple CFG structures like sequential executions, it may be efficient to generate the tracing location points for data collection with static heuristics to evaluate sub-paths iteratively. In comparison, random selection-based generation could be a better choice for functions with complex execution orders (e.g., multiple branches and loops) to involve more randomness when exploring the whole CFG.

*7.2.2 Sensitivity analysis of novelty detection models.* Synthetic faults injected into the functions can be set at different sensitivity levels to verify the effectiveness of different novelty detection models in Section 5. We conducted numerical experiments to simulate single and multiple faults on the tracing data without internal faults to control the sensitivity level. The sensitive evaluation is based on testing data where there is no limitation of the budget, and all observed data is available to show the comprehensive detection ability.

In Figure 6, we take both single and multiple faults with different sensitivity levels into consideration. First, it can be seen from 6a that with 30% increase of delay in the function, except *kmeans*, the other trained models achieve a high recall score close to 1, providing promising outcomes of the detection for single fault injection. While the performance of *kmeans* is also adequate with a recall value of 0.75 on *SF*30. For multiple faults in Figure 6b, with 30% level, all the trained models obtain good performance on the recall score no less than 80%. Among all trained models, *AE*-based algorithms outperform with a recall value 0.99 on 30% level for both single and multiple-fault injection.

When the synthetic fault is at 15% and 3% level, the difference in the recall value among all models is noticeable. In Figure 6a, *AE*-based method outperforms others with a recall value 0.97 on 15% injected faults. While all four presented methods cannot ensure correct identification of the anomalies at 3% level, this appears reasonable since the injected faults are too subtle compared to the original execution times. By computing the statistics of the dataset, the standard deviation of tracing dataset without faults is greater than 3% level of faults injection, indicating that 3% increase of the function execution time may come from the system noise. Thus, in practice, abnormal behaviors cannot be captured at this level. In Figure 6b, multiple faults exist in the functions with 3% increase of the latency, *AE*-based model slightly improves recall value compared to other methods. Consequently, the neural network-based model is more sensitive to detecting different levels of anomalies in our problem.

Table 2. Compared results of enhanced BB algorithm with limited budget

| | | SF30 | SF15 | MF30 | MF15 |
|---|---|---|---|---|---|
| Iterations | Original | 10 | 14 | 20 | 30 |
| | Enhanced | 1 | 3 | 10 | 17 |
| Identification index ($10^3$) | Original | 3.3 | 3.3 | 2.9 | 2.0 |
| | Enhanced | 3.3 | 3.3 | 2.9 | 2.0 |
| Relative score | Original | 1 | 1 | 0.63 | 0.63 |
| | Enhanced | 1 | 1 | 0.71 | 0.71 |
| Probes | Both | 2 | 2 | 3 | 4 |

Table 3. Compared results of enhanced BB algorithm without budget constraint

| | | SF30 | SF15 | MF30 | MF15 |
|---|---|---|---|---|---|
| Iterations | Original | 10 | 14 | 77 | 317 |
| | Enhanced | 1 | 3 | 9 | 145 |
| Identification index ($10^3$) | Original | 3.3 | 3.3 | 4.8 | 4.4 |
| | Enhanced | 3.3 | 3.3 | 4.8 | 4.4 |
| Relative score | Original | 1 | 1 | 1 | 0.96 |
| | Enhanced | 1 | 1 | 1 | 0.96 |
| Probes | Both | 2 | 2 | 4 | 6 |

*7.2.3 BB result analysis.* Two budget scenarios are considered to assess the enhanced BB algorithm for tracing probe location inference. To reduce the impact of randomness in the heuristic starter generation, we run 20 times the optimization programs to take the average results. In the following experiments, we apply *AE*-based novelty detection models and random generation of heuristic starting locations on data collection (in section 4.2) to evaluate the performance of the enhanced BB. We assume that there is an extreme constraint on the budget specified for the total cost of obtaining the instrumentation location to identify the faults. Within the budget, it is possible not to locate each fault, but the outputs of the BB methods are the best locations within the budget to place the probes. We also conduct another experiment without budget limitation to allow the BB-based methods to explore the underlying CFG of the functions fully.

**Result Analysis of RS-Data.** First, we compare the number of iterations to obtain the probe placement inference from the optimization program. We can learn from Table 2 that under a limited investigation budget, compared to the original BB without any searching and bounding strategies, the enhanced BB can achieve fast convergence by saving 65% of the number of iterations on average. For $SF30$ and $SF15$ with a single fault, the enhanced search can obtain the probe placement solution within 3 iterations. It is important to emphasize that the size of the underlying BB trees for different datasets is also related to the specific fault locations, except for the number of faults.

When there is no limitation of budgets, all the nodes in the CFG can be observed, although this is usually infeasible in practice. Compared to the results in Table 2, the iteration values remain the same for $SF30$ and $SF15$, indicating that the limit budget is enough to obtain the probe placing locations based on our method. The iterations of the multiple-fault dataset grow compared to the values under the budget limit since more observations are available when expanding the BB trees during searching. The enhanced BB can still efficiently save over 50% iterations (a minimum of 54% for $MF15$ and a maximum of 88% for $MF30$) compared to the original algorithm, as shown in Table 3. Thus, the enhanced features with graph dependencies can accelerate the optimization process to locate the function faults.

Table 4. Compared results of BB algorithm with a budget constraint on RS-WF-Data

| | | SF30 | SF15 | MF30 | MF15 | DMF30 | DMF15 |
|---|---|---|---|---|---|---|---|
| Iterations | Original | 13 | 13 | 18 | 20 | 17 | 22 |
| | Enhanced | 9 | 9 | 8 | 11 | 8 | 9 |
| Identification index ($10^3$) | Original | 2.8 | 2.8 | 3.5 | 3.5 | 3.5 | 2.8 |
| | Enhanced | 2.8 | 2.8 | 3.5 | 3.5 | 3.5 | 2.8 |
| Relative score | Original | 1 | 1 | 0.62 | 0.62 | 0.62 | 0.5 |
| | Enhanced | 1 | 1 | 0.62 | 0.62 | 0.62 | 0.5 |
| Probes | Both | 2 | 2 | 3 | 3 | 3 | 3 |

Table 5. Compared results of BB algorithm without budget constraint on RS-WF-Data

| | | SF30 | SF15 | MF30 | MF15 | DMF30 | DMF15 |
|---|---|---|---|---|---|---|---|
| Iterations | Original | 29 | 29 | 100 | 52 | 199 | 105 |
| | Enhanced | 3 | 3 | 20 | 8 | 53 | 21 |
| Identification index ($10^3$) | Original | 2.8 | 2.8 | 5.6 | 5.6 | 5.6 | 2.8 |
| | Enhanced | 2.8 | 2.8 | 5.6 | 5.6 | 5.6 | 2.8 |
| Relative score | Original | 1 | 1 | 1 | 1 | 1 | 0.5 |
| | Enhanced | 1 | 1 | 1 | 1 | 1 | 0.5 |
| Probes | Both | 2 | 2 | 5 | 5 | 5 | 3 |

Next, we turn to a brief discussion on the identification index and the relative score with different budgets. Among Table 2 and 3, two BB search algorithms obtain the same identification index for single-fault datasets $SF15$ and $SF30$ with a relative score 1. The results suggest that the inferred tracing probe locations correctly capture the internal anomalies as the best placement in the ground truth. For $MF30$, the relative score of the sufficient budget improves to 1 compared with the value in Table 2. Besides, the identification index of $MF15$ also increases over 2 times, and the relative score is upgraded to 96% by capturing all internal anomalies. It can be inferred that without budget limitation, more observations are provided to identify the anomalies and suspicious nodes that lead to a higher identification index. Note that the final placement solutions are dependent on the performance of novelty detection models on different datasets. This can be explained by the fact that when all anomalies are detected in $MF15$, the relative score is not equal to 1 due to the inclusion of false positive labels produced by the detection models. These false positives increase the number of probes in the placement inference, which affects the comparison with the best solution in the ground truth.

By comparing the number of probes in the inferred solution to a fully distributed tracing (via instrumentation of all 12 nodes in the CFG under evaluation), the solutions from either search algorithm in Table 3 reduce the instrumented nodes to provide the inference for fault localization.

**Result Analysis of RS-WF-Data.** For the experiments with the serverless workflow, we also simulate internal anomalies in different functions by considering the same levels of faults as *RS-WF-Data*. In this scenario, we first compare the enhanced BB search results to the original method with a limited instrumentation budget. The comparison results of iterations to obtain the probe placement presented in Table 4 demonstrate that the enhanced BB consistently outperforms in reducing over 50% of the number of iterations across all 6 datasets of the serverless workflow, compared to the number of iterations with the original search strategies. This aligns with the results observed in *RS-Data* for the single serverless function. Table 5 shows the results if we can allocate a sufficient instrumentation budget to locate the internal faults. We can observe that, compared to the iteration values in Table 4, there is a notable increase in the

number of iterations. This increase is attributed to the expansion of the search space, which resulted in the inclusion of more observations from feasible probe locations. However, the enhanced BB strategies can still achieve significant savings in the number of iterations required to generate probe locations for internal faults, with a minimum of 80% reduction. Moreover, when comparing the iterations between *RS-Data* and *RS-WF-Data* in Table 3 and Table 5, it can be observed that, despite a larger number of CFG nodes in the serverless workflow for *RS-WF-Data*, it requires fewer iterations to locate faults when a sufficient budget is available. This is attributed to the fact that probe placement results are influenced not only by the number of nodes in the given CFG but also by the topology of the internal function calls.

Then, we discuss the experimental results on the identification index and the relative score for the serverless workflow datasets. In both Table 4 and Table 5, it can be initially observed that the identification index and the relative score of the original strategy and enhanced BB search remain consistent. This indicates that the improved strategies can ensure the same level of fault detection abilities as the original strategy. However, the identification index for multiple faults with a limited budget consistently maintains low values among all datasets in Table 4. These results show the inadequacy of the provided budget, which is insufficient for accurately locating internal faults. When analyzing the relative scores in Table 5, it becomes apparent that, apart from *DMF15*, all other scores reach 1 by identifying all potential faults using the probes obtained from the BB search. While for the results of *DMF15*, we suspect that the low detection performance may be linked to the accuracy of the novelty detection models with the specific probe locations.

In summary, the overall results of BB search for anomaly diagnosis are tightly related to the locations and number of faults in the serverless functions. Multiple faults and complex execution orders in the CFG complicate the problem of tracing probe inference. Thus, more budgets are required to produce the optimal solutions for anomaly diagnosis at probe locations. At the same time, the enhanced algorithm takes less time to provide the same identification results under both demanding constraints.

## 8 Related work

From the perspective of serverless developers, how to automatically achieve fine-grained troubleshooting for internal faults of serverless functions still lacks investigation in the state of the arts.

Customized tracing and profiling enable developers and users to monitor the system at their preferred level and focus on areas of interest. However, instrumentation overhead consistently presents a problem, impacting both the end-user experience and the efficiency of the programs. In [2], the authors state that it is infeasible to enable all possible instrumentation all the time for a function due to the unacceptable overhead. Several works aiming at reducing the instrumentation overheads, for example, In [4], Ball *et al.* propose a set of optimal algorithms enhancing efficiency by reducing counter numbers and trace file sizes for program profiling and instruction tracing, to reduce the overhead of profilers. The proposed approaches improve efficiency in profiling and tracing within individual programs by optimizing the placement of counters and tracing instrumentation, which is particularly beneficial for analyzing performance and identifying bottlenecks within single applications or specific components of a system. However, serverless applications often present unique challenges that may require more advanced and robust techniques. Unlike traditional applications, serverless architectures involve functions that interact dynamically and frequently across various services and components. To effectively monitor and optimize these applications, it is crucial to implement comprehensive tracking and performance monitoring methods. These techniques should be capable of capturing detailed performance metrics and transaction data as requests pass through the different functions and services. There requires methods that enable developers to understand the complex interactions within the serverless environment, identify potential performance issues, and ensure that the application operates efficiently and reliably.

Troubleshooting and faults localization are more challenging for serverless functions due to the limitation of accessibility and control of the infrastructures for developers [24]. Manner *et al.* propose to use log data for fault detection and resolution of serverless functions via semi-automated monitoring and debugging [15]. However, logs and alert messages are difficult to customize and analyze at the code level to track the procedure executions within and across functions. In [13], GammaRay is introduced to capture the causal dependencies for debugging and optimization of serverless applications on AWS Lambda. The advantage of this proposed methodology is that it requires no intervention from the developers to record the performance data. Emre *et al.* present an automated instrumentation framework based on calling context trees for workflow-centric tracing, aiming at the data of the requests processed within and among workflows with low overhead in [2]. In [6], the authors evaluate several distributed tracing approaches and present a developer-driven tracing method. They focus on the integration of distributed tracing in serverless applications. This work improves observability for both developers and platform providers.

While most existing research on serverless troubleshooting primarily focuses on observations at the function level, there remains a significant research gap in terms of providing fine-grained insights within serverless functions. Current methods often aggregate data at a higher level, which may not capture the details of the internal workings of each function. As a result, developers are left with limited visibility into the specific areas within a function where faults or performance issues occur. Addressing this gap would involve developing more sophisticated techniques for automatic instrumentation and monitoring within serverless functions, enabling a deeper analysis of their behavior and facilitating more effective debugging and optimization.

To obtain the required visibility of executions, customized tracing and monitoring can be applied to collect and correlate performance metrics and trace data with instrumentation [20]. However, the quality and frequency of instrumentation are critical to the resulting observation. There are studies on the automatic instrumentation of tracing at different levels. In [8], the proposed method provides dynamic instrumentation of tracing within machines and across clusters. In [21], Popa and Oprescu introduce data flow analysis into the codebase to obtain the instrumentation points for data-centric distributed tracing. Although these studies address the issues of instrumentation with only expert knowledge, there are limited works to our knowledge that provide code-driven instructions for developers for fine-grained instrumentation of spans for serverless functions, especially on code-level troubleshooting.

## 9 Threats to Validity

In this section, we identify and discuss potential threats to the validity of our methodology and results.

- Assumption of offline information gathering: Our methodology proposed is based on the assumption of the ability to conduct offline information gathering with the same configurations as the production environment. However, this may not always be feasible or applicable, potentially limiting the generalizability of the methodology to other serverless platforms.
- Synthetic fault injection: Injecting sleep functions into different locations of the source code to simulate synthetic increases in latency may introduce bias and oversimplify real latency increase by internal anomalies due to the blocking nature of sleep functions. The increased latency through the sleep function can intentionally create synthetic internal faults to assess the performance of diagnosis. It could be possible to use wait functions that allow for performing other tasks while waiting for specific operations to complete, which may provide a more realistic simulation of performance anomalies and reducing bias in the experimental results.

- Limited experimentation scope: Conducting controlled experiments with fully accessible source code may limit the scope of validation and effectiveness demonstration, potentially overlooking the performance faults originating from IO or network bottlenecks.
- Required inputs of instrumentation cost and overall budget: Assuming that instrumentation cost can be measured and specified by developers may oversimplify the realistic development and management of serverless applications, potentially leading to an inaccurate representation of the optimization process. In practice, the optimization program can be reformulated under realistic constraints that can be measured, such as resource consumption to adapt the current solution to the necessary scenarios.

## 10 Conclusions

We have presented a code-level tracing methodology for internal anomaly diagnosis of serverless functions via distributed tracing. We exploit the code-driven encoding of tracing probe locations to detect the faults with heuristics-based data collection at a fine-grained level driven by static code analysis. Path-based novelty detection models are trained with the profiling data to detect anomalies inside serverless functions. We then proposed an enhanced search-based algorithm to obtain the instrumentation locations for tracing probes at the code level for anomaly diagnosis. The experimental results show that the instrumentation overheads can be reduced with our enhanced search methods and keep the same good accuracy to locate the anomalies.

Possible future directions can target improving the robustness of anomaly diagnosis for the changes in the underlying production environments of the serverless functions. The tracing data can suffer data shifts due to different resource provisions and platform configurations. Potentially, data-driven adaptive learning methods may contribute to the generalization and robustness of code-level anomaly diagnosis.

## References


[1] A. Amuthan and K. Deepa Thilak. 2016. Survey on Tabu Search meta-heuristic optimization. In *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*. 1539–1543.

[2] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 165–170.

[3] Thoms Ball. 1999. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 216–234.

[4] Thomas Ball and James R. Larus. 1992. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 59–70.

[5] Sandrine Blazy, Delphine Demange, and David Pichardie. 2015. Validating dominator trees for a fast, verified dominance test. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*. Springer, 84–99.

[6] Maria C Borges, Sebastian Werner, and Ahmet Kilic. 2021. Faaster troubleshooting-evaluating distributed tracing approaches for serverless applications. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 83–90.

[7] OpenTelemetry Collector. 2019-2024. https://opentelemetry.io/docs/collector/

[8] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–35.

[9] Maryamsadat Hejazi and Yashwant Prasad Singh. 2013. One-class support vector machines approach to anomaly detection. *Applied Artificial Intelligence* 27, 5 (2013), 351–366.

[10] Shehroz S Khan and Michael G Madden. 2014. One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review* 29, 3 (2014), 345–374.

[11] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141.

[12] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27 (2022), 1–28.

[13] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. 2018. Tracking causal order in aws lambda applications. In *2018 IEEE international conference on cloud engineering (IC2E)*. IEEE, 50–60.

[14] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[15] Johannes Manner, Stefan Kolb, and Guido Wirtz. 2019. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems* 34 (2019), 99–104.

[16] Andrew Ng et al. 2011. Sparse autoencoder. *CS294A Lecture notes* 72, 2011 (2011), 1–19.

[17] F. Nielson, H.R. Nielson, and C. Hankin. 2015. *Principles of Program Analysis*. Springer.

[18] OpenFaaS. 2024. https://docs.openfaas.com/.

[19] OpenTelemetry. 2019-2024. https://opentelemetry.io/

[20] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.

[21] Nicolae Marian Popa and Ana Oprescu. 2019. A data-centric approach to distributed tracing. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 209–216.

[22] Khirulnizam A Rahman and Md Jan Nordin. 2007. A review on the static analysis approach in the automated programming assessment systems. (2007).

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[24] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *Comput. Surveys* 54, 11s (2022), 1–32.

[25] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[26] Douglas Steinley. 2006. K-means clustering: a half-century synthesis. *Brit. J. Math. Statist. Psych.* 59, 1 (2006), 1–34.

[27] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW 2021*. ACM, 3087–3098.