Online Diagnosis of Microservices Based Applications Via Partial Digital Twin

Sourav Das, Jit Gupta, Krishna Kant
Computer and Information Sciences Temple University, Philadelphia, USA
{sourav.das | Jit.Gupta | Kkant}@temple.edu

***Abstract*—The increasing adoption of DevOps and microservices paradigm in enterprise software has not only brought about several benefits but also a host of challenges. In particular, such applications may suffer from frequent misconfigurations and other faults that need to be diagnosed promptly. In this paper, we propose a novel framework centered around the concept of Partial Digital Twin (PDT) to enable isolated testing without disrupting the main infrastructure. Our framework employs a systematic methodology to replicate the chain of dependencies relevant to reported faults, facilitating efficient root cause analysis. Additionally, we introduce a hierarchical categorization of faults leading to the selection of tests tailored to specific fault types. Through empirical evaluation, we demonstrate that the median number of tests required to diagnose a misconfiguration using our approach is only 7% above the ideal scenario.**

***Index Terms*—Microservices, partial digital twin, root cause analysis**

I. Introduction

The DevOps transformation of IT services is fueling a radical change in how cloud services are conceptualized, designed, and implemented [1]. The emerging microservices paradigm, aided by service-mesh to manage multiple instances of a microservice, is increasingly replacing the traditional monolithic designs [2]. The popular idea now is to decompose a large service into relatively independent microservices, each of which can be independently developed and maintained. This is facilitated by mechanisms to limit coupling between them such as asynchronous calls, weak consistency, lock-free operation, circuit breaker, etc. Each microservice typically runs in its own container environment, and the *service-mesh* architecture allows the number of running microservice instances to be changed dynamically based on the offered load. Such transparent instancing also allows for concurrently running multiple versions of a microservice simultaneously (e.g., an older proven version and a newer test version) along with means to direct workload to specific instances.

Unfortunately, splitting a sequential application into a large number of microservices creates a large distributed system out of a centralized one. The dynamic, heterogeneous instancing and features like lock-free shared data access further increase the conceptual complexity of the application and hence the chances of concurrency, synchronization, and other faults. Furthermore, the Continuous Integration/Continuous Deployment (CI/CD) processes associated with DevOps result in continual updates, reconfiguration, and instancing of microservices, which substantially increases misconfiguration faults.

Thus an online diagnosis mechanism that can quickly root-cause the problems becomes essential. Because of the typical small sizes of microservices, automated root-causing is needed only at the level of microservices and their interactions, and this is a focus of this paper. Deeper analysis such as identifying incorrectly set individual configuration variables or program bug identification is not within the scope of this paper and may be tackled by other mechanisms, including manual checking.

Microservices diagnosis has been explored in several prior works but is typically concerned with offline analysis and visualization of collected logs [3] [4]. The intent is to proactively find potential problems. Zhou et al. [5] discusses various tools for such diagnosis in a microservices environment. Recent works often use machine learning algorithms for such diagnosis [6]. Van Aken et al. [7] presents a machine learning technique to predict configuration-induced failures in microservices environments.

To effectively solve the microservices level root cause analysis problem, we propose a novel framework that implements the concept of a ***Partial Digital Twin*** (PDT). Unlike a digital twin (DT) that replicates the entire system at certain level of abstraction [8], a PDT temporarily replicates only a part (i.e., some microservices and the data they use). The PDT provides a controlled environment that replicates a part of the production system without disturbing the mainstream infrastructure [9] services. Testing in this isolated environment ensures that potential faults or failures discovered during analysis do not impact end users or business operations. An example is running test transactions that modify tables, which would be highly undesirable to do in real databases. Instead, we can reproduce specific states or conditions in the digital twin to observe and analyze faults. The contributions of this paper include:

1) We demonstrate how the PDT can be implemented efficiently by using Istio's microservices orchestration and logging features.
2) We propose a hierarchical fault categorization to create a systematic diagnosis procedure for various microservices faults.
3) Our experiments show that the proposed automated method nearly matches ideal diagnosis performance.

The rest of the paper is organized as follows. Section II explains the microservices architecture and configuration pa-

This research was supported by NSF grant CNS-2011252.

rameters, Section III elaborates on the framework design for fault diagnosis, Section IV explains the evaluation of our proposed framework, and Section V provides a conclusion.

II. Microservice Environment and Service Mesh

A. Microservices and Configuration Parameters

Modern microservice-based applications decompose functionality into independent, loosely coupled services, which form a well defined path, represented as a DAG (directed acyclic graph) and known as a service graph.

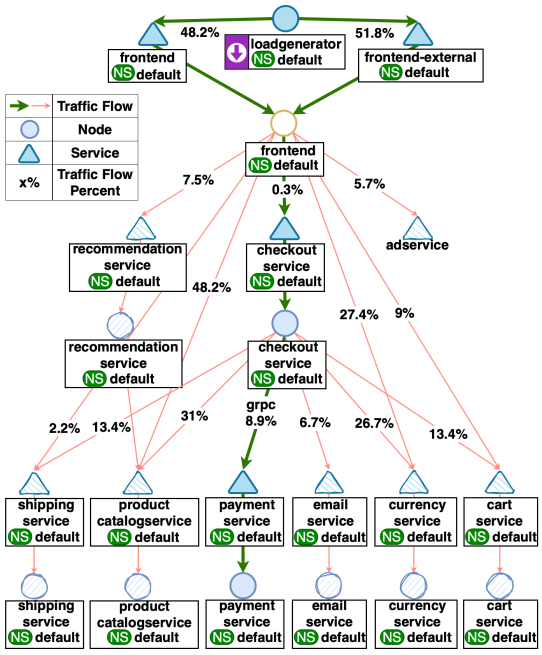

Fig. 1: Sample DAG for a request using Istio

Fig. 1 shows such a graph for the service mesh of a microservices based shopping application [10] that we have used for this paper. The green arrows depict a sample transaction, with requests facing contention at infrastructure (individual servers) and platform (communication channels) levels. This contention, coupled with the service processing times influence the performance during the journey through the graph [11].

TABLE I: Test-based Configuration Parameters for Microservices

Attribute Type	Configuration Parameters
Network	DNS Resolution, Network Latency, Network Routing
Security	Access Control, API/JWT Keys, Authentication
Infrastructure	Service Discovery, Istio Config. (Dest. Rules, Request Routing, mTLS Encryption), Port Accessibility, Payload, Response Time
Database	DB Connectivity, DB Grants, DB Query Execution, Maximum File Size, Caching, Cron Jobs Scheduling
Data	Data Formatting, Invalid Input Handling, Environment Type Config.

Kubernetes [12] or K8s, a container orchestration platform, offers several configuration management options like ConfigMaps and Secrets. It facilitates the management of configuration data separately from the application code. Istio extends Kubernetes' capabilities by providing traffic management, security, and observability features. A ReplicaSet in K8s is a fundamental controller that ensures a specified number of pod replicas are running at any given time.[1]. K8s maintains the desired state of the application by managing pod instances. If a pod crashes or is deleted, the ReplicaSet automatically creates a new pod to replace it. Within the K8s environment, infrastructure parameters like "replicas" and "ingress" play a critical role in managing load and ensuring stability. In addition to microservices themselves, there are several configuration parameters (CPs) associated with the service mesh that concern how microservices are instantiated and invoked. Our diagnosis is focused on detecting problems in these CPs. For example, DestinationRules in Istio defines policies that apply to traffic intended for a specific service, such as load balancing, connection pool settings, etc. Table I shows the functionalities of various resources that are usually specified via a set of CPs.

B. Service Mesh

Microservice architectures are commonly described as collections of loosely coupled services designed by independent development teams. Real-world deployments often integrate a substantial number of services created by external entities. These external services can be categorized as either sidecar services or core services [13].

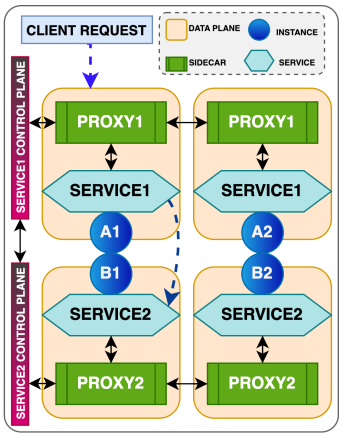

Fig. 2: Service Mesh Architecture

Sidecar services encapsulate common functions essential for microservices. It acts as an intermediary by intercepting all external requests for configuration management, logging etc. For example, when "SERVICE1" needs to communicate with "SERVICE2", it sends its request to its local sidecar proxy instead of directly contacting the target service (Fig 2). Two primary types of sidecars are (a) Service Mesh and (b) API Gateway. In contemporary cloud-native architectures, service meshes have emerged as a pivotal solution for managing communication between microservices. We chose Istio [14] to conduct this research, , although any other service-meshes such as Linkerd [15] could also be used. Both K8s and Istio provide several logging tools that we will be using in our work. We have used Prometheus [16], a service monitoring tool, to collect metrics from all services in the system, providing insights into performance and resource utilization. Analyzing these metrics can help identify potential bottlenecks or anomalies that might be causing faults. Distributed tracing techniques like Jaeger [17] allow tracing requests across multiple services. This helps developers identify the path taken by a request to pinpoint the service where a fault occurred.

[1]A pod is the smallest unit of deployment in K8s and represents a single instance of a running process.

III. Conducting Diagnosis Using PDT

A. Building and Maintaining PDT in Istio

In this section, we discuss the details of constructing the partial digital twin (PDT). PDT provides reproducibility of the environment, which is crucial in a live environment where conditions continuously change. PDT supports such reproducibility but can also effectively deal with the impacts of configuration/code changes by allowing for specific microservice instances to be represented in the PDT. In particular, tests on new or old instances can run in PDT while the production system processes transactions with its current settings or CI/CD related changes [18]. Furthermore, running tests in a production environment during episodes of congestion or heavy load may result in unexpected consequences including the impact on normal transactions or failed tests due to the load [19]. Note that a full digital twin (FDT) of a large enterprise system is essentially impractical – in addition to requiring enormous amounts of extra resources, it will also likely be bogged down by the synchronization traffic. In contrast, an on-demand PDT can allow for isolated testing without the need for substantial resources. The key to using our proposed mechanism successfully is to keep the PDT as small as possible and yet include the parts most relevant for the root cause analysis. We assume that the microservices system uses a service-mesh for dynamic scaling, and we exploit this capability for constructing PDT as well. We use Istio features to launch, idle, and destroy microservices instances that are intended to be part of the PDT, without the need for any specialized infrastructure.

We build PDT with the help of Istio features for replication of microservices and redirection of workload queries to various microservice instances. Because of the expense of replicating large parts of the enterprise system, we build PDT incrementally and dynamically based on the diagnosis needs. This involves two aspects: (a) Selecting the subset of microservices for which we want to create PDT instances, and (b) The data to be used for running the test queries in the PDT. For (a), we maintain a cache so that the microservices that most often are involved in diagnosis have their instances present, while others may be evicted depending on the cache size maintained. This is achieved by maintaining a hash map and a doubly linked list. The former provides $O(1)$ access time to cache entries, while the latter maintains the order of access, with the most recently accessed items being moved to the front and the least recently accessed items to the end. The real databases used by the queries may be quite large and it is not possible to replicate them in the PDT or synchronize them. Therefore, we use a mechanism that we call as *query transformation* for (b), which effectively reduces the data requirements of the query. This is essential since the PDT cannot afford to maintain huge amounts of data contained in large relational tables. Instead, we judiciously compress the size of large tables, and alter the queries on the fly to refer to only the limited data range stored in PDT. In general, it can be very complex to achieve such reduction while retaining the query semantics. This is particularly true in the context of joins where we would like to maintain similar levels of join selectivity. Our current solution is based on an intimate knowledge of the tables involved, and not intended to be fully automated. However, to reduce handpicking of data, we employ two techniques: *a) stratified sampling* and *b) aggregation*. For large tables, we apply stratified sampling, where data is divided into homogeneous subgroups (strata) based on important attributes like time ranges, transaction amounts, or customer demographics. In cases where random sampling is more appropriate (e.g., in datasets where the distribution of values is uniform), we employ this method to ensure that the reduced dataset does not introduce bias. For queries that involve summary statistics (e.g., order totals by period), we use pre-aggregated tables to store condensed representations of the data. For example, if a query targets all transactions over $100 in the last five years, the query transformation mechanism redirects it to an aggregated table that only stores orders for high-value transactions over a reduced time period, such as the last two years.

The implementation involves the deployment of object-relation mapping models of the database entities as separate services. We then implement in-memory caching using a Redis cache server. As shown in Fig. 4, we determine the most frequently used queries through empirical analysis by identifying the most invoked microservices and, hence backtracking to the type of query triggered. Specifically, we compute the access frequency f_i for each microservice i and its associated queries q_i, where $f_i = \frac{\text{Number of invocations of microservice i}}{\text{Total number of invocations of all microservices}}$. Firstly, we find all frequently called microservices, and then enumerate all the queries. Queries with the highest frequencies are selected, and the cache is updated accordingly to reflect these priorities.

We are using a Redis cache server [20] for our experiments. It does not have a predefined size; instead, Redis provides a configurable parameter to specify how long to maintain a query. When a query encounters problems, we match this query against the queries present in the cache. The principle of query matching is the exact match of keys between the query request and cache entries. For instance, if user profiles are cached with a key like user:x, a query to fetch user "10" would use the command GET user:10. Redis performs a hash-based lookup to find this key and returns the associated value if it exists and has not expired. The SCAN command is then used with a pattern like user:* to match multiple keys. If no matching query exists, we insert it in the cache, else we increment the frequency of the matching query.

B. Framework of PDT based Fault Diagnosis

Our diagnosis is triggered by a reported problem chosen from a predefined set of 8 fault types (Table II), typically from users and administrators, but could also come from automated monitoring systems. It results in launching a diagnosis session with the help of PDT. The human-reported problems may be vague or may incorrectly state the problem. This invariably requires a dialog to gather adequate pertinent information to enable a focused diagnosis procedure. Although, we do

not delve into it, the rapid progress in the large language model (LLM) area could enable suitably fine-tuned LLMs to substitute people for holding such a dialog.

Essentially, we are considering two things. Firstly, a ticket is generated with the reported fault. The next step deals with subset identification of the microservices that need to be tested in the PDT. The identification of microservices conclude the first part of the framework. Once, they are identified, it is then checked if they are already present in the PDT or not. If they are not present, they are replicated. The next step is to run a series of hierarchical tests on the replicated set of microservices in the PDT environment. The selection of tests is done using a two step approach. First, a neural network is built to implement Zero Shot Learning (ZSL) [21], which categorizes objects by leveraging semantic information. In our case, it outputs the probability of relevance associated with each fault and test pair as shown in Fig 3. The second part is to determine the order of running the tests. That is done by executing the tests in a specific order that captures all the necessary attributes of a specific reported fault (Fig 5). For example, if we consider a Service Unreachable test, the tests that are going to be executed are {B, R-W, I-Q} or {B, I-Q} depending on the outcome of the tests. The test IDs mentioned above are explained in Table V. We execute the tests unless the misconfiguration is detected. If they are all used up without finding the misconfiguration, we rerun the subset indentification step and add more services again for testing. Further explanation is provided later in Section III-D.

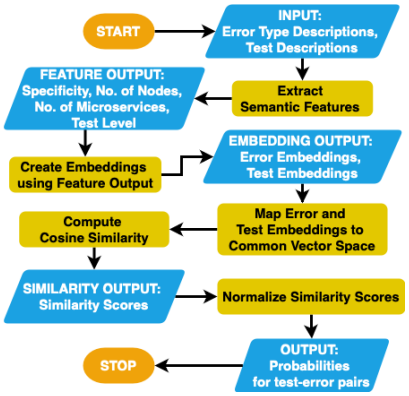

Fig. 3: Algorithm Overview

C. Fault Types and Tests

To effectively diagnose issues in a microservices architecture, it's crucial to categorize fault types and design corresponding tests. We have categorized faults into primarily 8 types (Table II) on the basis of commonly reported faults in the industry [22] and have come up with 26 unit tests (Table III) that can assist us in finding misconfiguration related faults. Faults such as UN (Unreachable Network) and SN (Slow Network) can be investigated by simulating network failures or latency to assess application behavior under these conditions. For US (Unreachable Service) and SS (Slow Service), tests can include service unavailability to identify issues with specific features or service response times. DE (Data Error) and DC (Data Corruption) require tests that validate data integrity and consistency, including checks for accurate data representation and error handling. UA (Unauthorized Access) can be addressed with access control tests to ensure proper user permissions. Finally, BA (Blocked Access) can be tested by implementing scenarios that simulate restricted access and examining how the application handles blocked resources.

TABLE II: Reported Symptoms and Fault Classes

Code	Fault Type	Symptoms Leading to User Report
UN	Unreachable Network	Unable to load the application or parts of it.
SN	Slow Network	Experiences delays in loading content.
US	Unreachable Service	Specific features not working with errors/timeouts.
SS	Slow Service	Certain actions taking longer than usual to complete.
DE	Data Error	Incorrect/inconsistent data displayed within the application.
DC	Data Corruption	Errors during data processing, or data loss.
UA	Unauthorized Access	Access denied when using previously accessible feature/data.
BA	Blocked Access	Unable to access parts of the application due to restrictions.

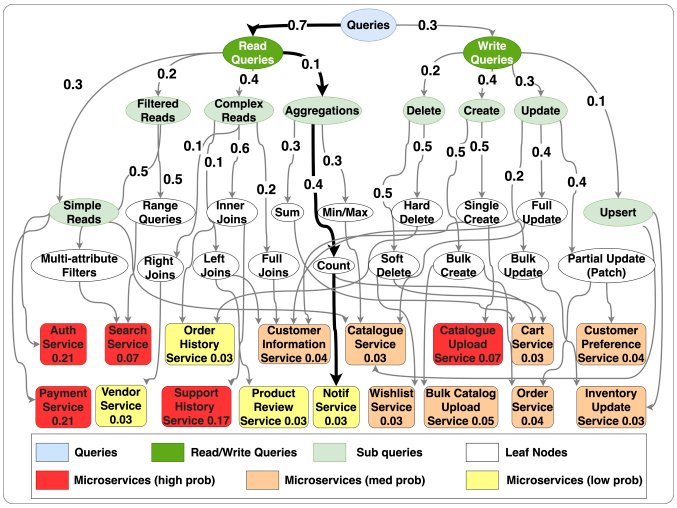

Fig. 4: Query Probabilities

D. Diagnosis Algorithm

Effective diagnosis requires that the detailed logging is switched on all the time so that the log data can be accessed whenever a problem is reported. This is currently true with most microservices systems since the overhead of detailed logging is rather small [23]. For simplicity, this paper deals with the common case of a single fault being reported at any time. While multiple simultaneous faults are possible, the complex procedures to handle them are beyond the scope of this paper. All the assumptions have been detailed in Table IV for clarity. Our diagnosis procedure includes two parts discussed below.

1) Identifying Microservices for Replication in PDT: In this section we present a heuristic method, shown in Algorithm 1 to determine the subset of microservices, denoted by $\mathcal{M}_{\text{to\_add}}$. The algorithm establishes the chain of dependent microservices $\mathcal{M}$ corresponding to the reported fault $\mathcal{E}$ based on the notion of a *fault-score*, computed by Algorithm 2, and iteratively adds the one with the highest score to set $\mathcal{M}_{\text{to\_add}}$

TABLE III: A-Z Unit Test Descriptions

ID	Commands to Run	Pass/Fail Conditions
A	nslookup <domain>	1 iff successful DNS resolution takes <1s.
B	ping -c 4 <domain>	1 iff ICMP latency to destination domain is <100ms in 4 attempts.
C	traceroute <domain>	1 iff destination is reachable.
D	ping <service_url>	1 iff ping to the target succeeds.
E	curl -I <service_url>	1 iff expected HTTP code.
F	curl -I -u <user>:<pass> <service_url>	1 iff 200 for auth, 403 for unauth.
G	curl -H "Authorization: Bearer <token>" <api_url>	1 iff 200 for valid, 401 for invalid.
H	curl -I -u <user>:<pass> <auth_url>	1 iff 200 for valid, 401 for invalid.
I	kubectl get svc <service_name>	1 iff service listed and accessible.
J	kubectl get destinationrule <rule_name> -o yaml	1 iff rule correct.
K	curl <istio_route_url>	1 iff correct service instance.
L	kubectl get peerauthentication -n istio-system	1 iff mTLS enabled and enforced.
M	kubectl top pods	1 iff CPU, memory <80%.
N	kubectl get pods -n <namespace>	1 iff all pods running.
O	netcat -zv <host> <port>	1 iff port open.
P	curl -X POST -d @payload.json <service_url>	1 iff correct response.
Q	curl -w %time_total <service_url>	1 iff <500ms.
R	nc -zv <db_host> <db_port>	1 iff reachable in <1s.
S	mysql -u <user> -p -e "SHOW GRANTS FOR <user>@<host>;"	1 iff correct grants.
T	mysql -u <user> -p -e "SELECT * FROM <table>;"	1 iff expected results.
U	upload_test_file.sh	1 iff upload <100MB.
V	curl -I <cached_url>	1 iff cache hit (X-Cache: HIT).
W	crontab -l — grep <job_name>	1 iff job scheduled correctly.
X	python data_formatting_test.py	1 iff data matches schema.
Y	python invalid_input_test.py	1 iff handles gracefully, appropriate error.
Z	kubectl config view	1 iff configurations correct.

TABLE IV: Assumptions in the implementation

No.	Assumption Description
1	**Stateless:** Microservices (MS) are stateless, with no session information retained between calls.
2	**Cascading:** Configuration changes may affect dependent MS.
3	**Query Transformation:** Queries are correctly transformed between environments without adding bias.
4	**Single Fault:** Only one fault occurs at a time within the system.
5	**No Intermittence:** Faults persist until resolved, with no intermittent behavior.
6	**Identifiability:** All misconfigurations are definitively identifiable.

if not already present in it. Each such iteration also runs the diagnosis procedure, which continues until either the fault's root cause is determined or the algorithm stops prematurely. The success rate of the algorithm or *diagnosis accuracy*, defined as the fraction of the cases where root-cause analysis terminates with correct root-cause, is an important measure that we report in the experimental results.

Algorithm 1 begins by initializing an empty list $\mathcal{M}_{\text{to\_add}}$ to store microservices that need to be added. It then enters a loop to identify candidate microservices $\mathcal{C}$ that are not currently in the LRU cache $\mathcal{A}$ but are called by at least one microservice in $\mathcal{A}$. If no such candidates exist, the loop breaks. Otherwise, it computes a fault-score for each candidate microservice using a scoring function. The computation of the fault-score in Algorithm 2 uses performance metrics obtained from Prometheus, such as *historical error rate* (HER), *historical call frequency*

Algorithm 1 : Replicating Microservices in the PDT

Require: $\mathcal{G} = (M, E)$: Call graph,
$\mathcal{E}_c$: Class of the reported problem,
$\mathcal{A}$: LRU cache of available microservices,
C_m: Set of callers for a microservice m,
error_rates: Historical error rates,
call_count: Historical call frequency

$\mathcal{M}_{\text{to\_add}} \leftarrow []$
while true **do**
  $\mathcal{C} \leftarrow \{m \in M \mid \exists c \in C_m \cap \mathcal{A}$ and $m \notin \mathcal{A}\}$
  if $(\mathcal{C} = \emptyset)$ **break**
  $S \leftarrow \{\}$
  for all $m \in \mathcal{C}$ **do**
    $S[m] \leftarrow$ scoring_function$(m, \mathcal{E}_c,$ error_rates, call_count$)$
  end for
  $m^* \leftarrow \arg\max_{m \in \mathcal{C}} S(m)$; $\mathcal{M}_{\text{to\_add}}$.append$(m^*)$
end while
while $\mathcal{M}_{\text{to\_add}} \neq \emptyset$ **do**
  new_addition $\leftarrow \mathcal{M}_{\text{to\_add}}[0]$
  if $|\mathcal{A}| <$ CACHE_SIZE **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{$new_addition$\}$; $\mathcal{M}_{\text{to\_add}}$.remove(new_addition)
  else
    evict $\leftarrow$ LRU$(\mathcal{A})$; $\mathcal{A} \leftarrow (\mathcal{A} \setminus \{\text{evict}\}) \cup \{$new_addition$\}$
    $\mathcal{M}_{\text{to\_add}}$.remove(new_addition)
  end if
end while
return $\mathcal{M}_{\text{to\_add}}$

(HCF), and *transaction call graph* (TCG), services exhibiting anomalous behavior are identified and considered for the next stage of analysis. The HER measures the frequency of faults for a given microservice over time, and the HCF captures how often the microservice is called. In computing the score from these measures, we use exponential smoothing over successive fault episodes and also introduce penalties to the fault rates based on a predefined threshold value. The candidate with the highest fault-score, m^*, is selected and appended to $\mathcal{M}_{\text{to\_add}}$.

Then, for each microservice in this list, it attempts to add it to the cache $\mathcal{A}$. If the cache has space (i.e., the size of $\mathcal{A}$ is less than the predefined `CACHE_SIZE`), the microservice is added directly. If the cache is full, the least recently used (LRU) microservice is evicted from $\mathcal{A}$, and the new microservice is added in its place. This eviction policy ensures that the most frequently and recently used microservices remain in the cache, optimizing the cache's performance. The process continues until all microservices in $\mathcal{M}_{\text{to\_add}}$ have been processed. Finally, the updated list $\mathcal{M}_{\text{to\_add}}$, now containing the optimal set of replicated microservices, is returned. This greedy approach ensures that the most critical microservices, as determined by their usage patterns and error rates, are available in the cache.

2) Selecting Tests to run for determining misconfigured CP: Test selection can be defined as the process of determining a set of unit tests from a list of 26 tests (Table III) whose purpose is to diagnose the misconfiguration in the replicated set of microservices $\mathcal{M}_{\text{to\_add}}$ obtained from Algorithm 1. As a prerequisite, we use ZSL to train a neural network to extract abstract features from the test-fault relationship [21]. Then, we calculate the weighted sum of the probabilities of the mapped features to determine the relevance of each test to a fault. Test selection begins by defining semantic features

Algorithm 2 : Computing Fault-Score of a Microservice

Require: $microservice$: The microservice to be scored,
$problem\_class$: The class of the reported problem,
$historical\_error\_rates$: A dictionary of historical error rates,
$historical\_call\_rate$: A dictionary of historical call rate,
$decay\_factor$: A factor to give more weight to recent data,
$error\_threshold$: A threshold for score penalty,
α and β: Coefficients for score calculation
Ensure: A score for the microservice
1: call_count ← HCF.get(microservice, 0)
2: error_rate ← HER.get(microservice, 0)
3: last_update_time ← time_of_last_update.get(microservice, 0)
4: t_{current} ← current_time()
5: recent_factor$_t$ ← $\gamma \times \text{HCF}_t + (1-\gamma) \times \text{recent\_factor}_{t-1}$
6: ncc ← call_count/max_call_count
7: ner ← error_rate/max_error_rate **if** (error_rate > error_threshold) penalty ← 1.5 **else** penalty ← 1.0
8: fault-score ← $(\alpha \times ncc + \beta \times ner \times$ recent_factor $\times$ penalty
9: **return** $score$

for both reported faults and diagnostic tests. Each fault and test is characterized by a set of numerical features, capturing the relevance to each other. For example, for a timeout fault, doing a DNS check first is more relevant than doing database configuration checks. The training data is constructed by combining these features for all possible pairs of faults and tests, with labels indicating whether each test is relevant to diagnosing a specific fault.

To enable ZSL, we encode the test and fault pairs into a high-dimensional vector using the popular Word2Vec [24] [25] model. These vectors encode semantic relationships between tests and faults for which we use 4 semantic features: (a) *Specificity*, which measures the number of fault types a test is relevant to, (b) *Number of nodes* to which the fault may be propagated, (c) *Number of microservices* to which the fault may be propagated, and (d) *Test level*, which denotes the detail provided by the test. For instance, a 'ping' test provides basic connectivity data, while a 'traceroute' test offers detailed network path insights, thus occupying a higher level. Let's consider the fault types and tests in our example: "Network Unreachable" and "ping." When we use embeddings, we convert these phrases into vectors. These vectors are essentially points in a multi-dimensional space where the distance between points reflects their semantic similarity. For instance, "network unreachable" and "ping" might be represented as vectors that are close to each other in this space because both relate to network connectivity issues. By calculating the cosine similarity between the vectors, we determine how closely related these concepts are. Finally, we get the relevance scores as the output as shown in Figure 3.

We built a table of relevance scores for the 26 diagnostic tests ("A-Z") and 8 fault types. Due to lack of space, only some entries are shown in Table V. The flowchart in Fig. 5 shows how to determine the order of test execution pertaining to each fault type (denoted by the grey boxes). Depending on the outcome of the tests, we follow the green (meaning success) or red (meaning failure) arrows to proceed to one of the five *attribute types* (in blue). Note that each attribute type consists of a set of tests and is executed sequentially on the

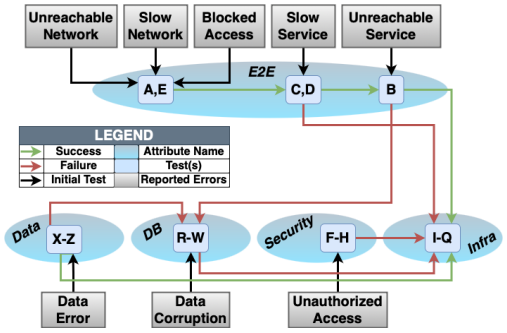

Fig. 5: Test Flowchart

TABLE V: A-Z Hierarchical Testing Suite

Test Categorization	Network		Service		Data		Access	
Test ID	**UN**	**SN**	**US**	**SS**	**DE**	**DC**	**UA**	**BA**
E2E								
(B)	0.99	0.97	0.85	0.92	0.00	0.00	0.41	0.99
Infrastructure								
(I)	0.37	0.00	0.00	0.00	0.00	0.00	0.00	0.49
.....	..	..	..	..	..	..	..	..
(Q)	0.87	0.93	0.22	0.78	0.11	0.02	0.21	0.25
Database								
(R)	0.53	0.49	0.61	0.00	0.79	0.88	0.00	0.00
.....	..	..	..	..	..	..	..	..
(W)	0.56	0.84	0.95	0.98	0.00	0.00	0.00	0.00

basis of the determined relevance score. For example, if we consider faults of type "**US**", the tests executed are {B, R-W, I-Q} or {B, I-Q} depending on the outcome of the tests.

IV. Evaluation

A. Experimental Setup

Our experimental environment consisted of a K8s cluster deployed on a set of virtual machines provisioned with the following specifications: each VM was equipped with 16 vCPUs, 8 GB of RAM, and 64 GB of SSD storage. The cluster was composed of 10 worker nodes and 3 master nodes to ensure high availability and scalability. The Istio sidecar proxy was injected into each microservice pod to facilitate service mesh functionalities. A representative set of microservices was designed to mimic a typical enterprise application [10]. The architecture included 26 microservices, each deployed as a separate Docker [26] container within the K8s cluster. The dependency chains were intentionally constructed to reflect real-world complexity, with an average of 5 dependencies per microservice. The longest chain in the setup comprised 9 microservices. To validate the diagnosis procedure, faults were systematically injected into the microservices. Specifically, 386 fault scenarios were built to target various faults. 260 of them were labeled as System A and the remaining 126 as System B, as shown in Fig. 6(c). These numbers were chosen to see the behaviour of the algorithm under different workloads. These faults were injected at random intervals to ensure comprehensive testing of the algorithm's robustness. The Redis cache server was deployed within the K8s cluster, with a memory allocation of 16GB.

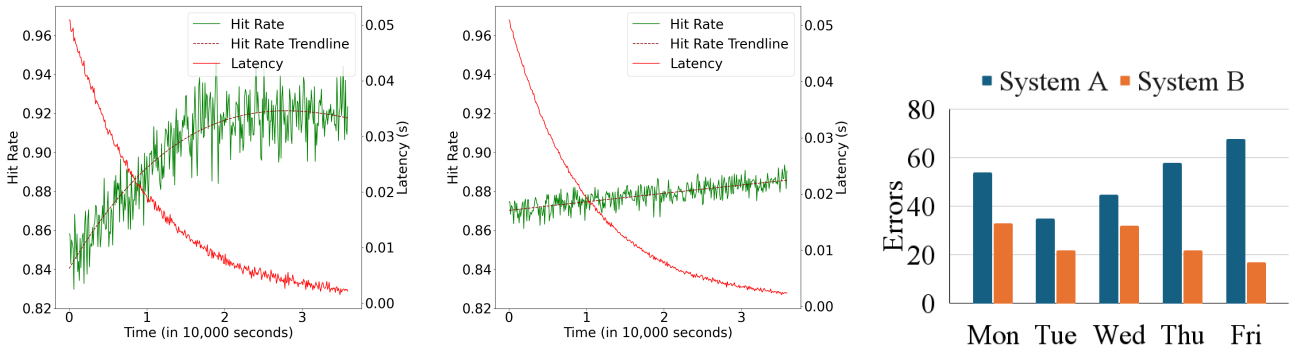

Fig. 6: (a) Caching perf. of Workload A, (b) Caching perf. of Workload B, (c) Distribution of faults reported

B. Experimental Results

Fig. 7 shows the performance of the algorithm as a measure of steps taken. We see that the mean number of steps taken is approximately 22% above the ideal case for all fault types combined. The average median is about 7% above the ideal case. Note that we have considered the ideal route (Fig. 5) to comprise a maximum of 1 test (highest score) from every attribute box, which adds up to form the ideal number of tests for each fault type. Each fault type displays distinct performance characteristics. For instance, faults such as "**SN**" and "**SS**" show a relatively higher number of steps taken, with a noticeable deviation from the ideal steps. This could be attributed to the complexity and variability inherent in diagnosing network-related issues. On the other hand, faults like "**US**" and "**DC**" exhibit a smaller gap between the actual steps taken and the ideal steps, suggesting that these fault types are more straightforward to diagnose within the proposed framework. Additionally, the number of incorrect reports, represented by the dotted line, provides insight into the algorithm's accuracy. A higher number of incorrect reports for faults like "**UN**" suggests that these cases are not only more complex but also more prone to diagnostic faults. The observation holds true even when multiple application instances are tested for misconfiguration diagnosis, later shown in Fig. 12. This emphasizes the need for further refinement in the algorithm to enhance its accuracy and reduce false positives or negatives. This could also indicate the need for further categorization of fault types. For the scoring logic in Algorithm 2, we performed a grid search to determine optimal values of α and β by considering the percentage of successful diagnoses with each combination. We varied both from 0.0 to 1.0 in steps of 0.1 and discovered optimal values as $\alpha = 0.7$ and $\beta = 0.3$, which were used subsequently. The values are assumed to be constants. Fig. 8 shows the error rate of microservices obtained from the K8s metrics as a function of time in 1000 sec units. As expected, the error rate goes down, but at a diminishing rate and eventually plateaus. This behavior simply shows that diagnosing frequently occurring misconfigurations can reduce the number of error reports. (This is somewhat optimistic since in reality, the CI/CD related changed may continue to bring in or expose new problems, but we do not consider that aspect in this paper.)

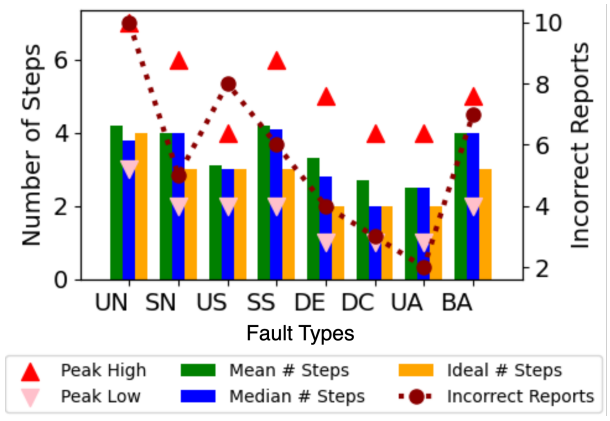

Fig. 7: #Steps taken to diagnose misconfiguration using proposed diagnosis framework vs ideal scenario

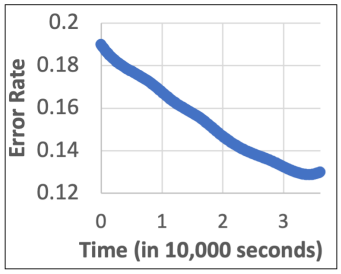

Fig. 8: Error rate vs. time

Next, we evaluate the performance of a Redis caching layer under two scenarios(Fig 6(c)). From our experimentation, we derive the hit rate and access latencies for both as shown in Figs. 6(a) and 6(b). System A achieved an average hit rate of 91%, compared to 86% for B. This shows that the cache is more effective under higher demand, serving a larger proportion of requests directly from memory. Latency averaged 3 milliseconds in both systems. However, the access latency values are more consistent for system B compared to system A. A low access latency indicates that the time taken to retrieve the data from the Redis caching layer is minimal, reducing the overall overhead compared to fetching the data from slower storage. The accuracy of the model also showed a

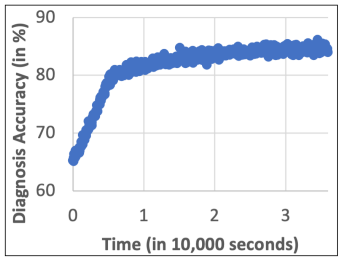

Fig. 9: Accuracy vs. time

positive growth and plateauing at a high value of 85% (Fig. 9).

The results above consider only a single e-commerce application. We next configured two such applications in our Istio implementation, one focusing on browsing (i.e., read heavy or I_r) and the other focusing on purchase (i.e., write heavy or I_w). In instance I_r, the queries result in 90% read operations, whereas in instance I_w, the queries result in 80% write/update operations. We programmed the load generator to issue a total of 10,000 requests over a 10-hour period to each application.

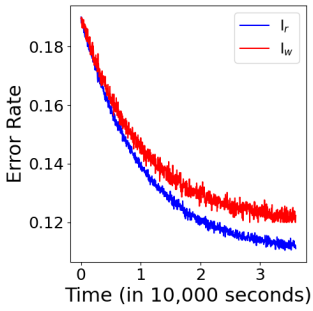

Fig. 10: I_r & I_w error rates

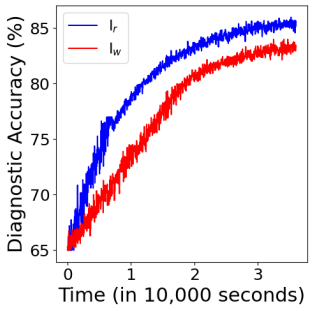

Fig. 11: I_r & I_w accuracy

Figs. 10 and 11 demonstrate the error rates for both I_r and I_w. The superior results for I_r is directly a result of fewer updates which makes diagnosis more difficult. This can be seen from Fig 12 which shows the number of tests required for each fault class for the two cases. It is seen that more tests are required in the write-heavy situation. This signifies the tradeoff between the number of tests required to diagnose a misconfiguration and the accuracy of the algorithm. Moreover, the previous findings from Figs. 8 and 9 remain consistent here as well.

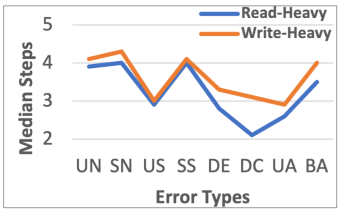

Fig. 12: Median #steps required for successful diagnosis

V. Conclusions

In this paper, we proposed a framework for performing online diagnosis of microservices triggered by an observed or reported malfunction. It uses the notion of a dynamic partial digital twin of the system to enable testing without perturbing the production system. Our empirical evaluation shows that the median number of tests required to diagnose misconfigurations is only 7% above the idealized scenario. In the future, we will focus on refining the fault categorization process and tackling a wider range of microservices faults.

References

[1] M. Krey, "Devops adoption: challenges & barriers," 2022.

[2] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in Proc. of ICITST, pp. 318–325, IEEE, 2016.

[3] M. Cinque, R. Della Corte, and A. Pecchia, "Microservices monitoring with event logs and black box execution tracing," *IEEE transactions on services computing*, vol. 15, no. 1, pp. 294–307, 2019.

[4] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting anomalies in microservices with execution trace comparison," *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[6] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pp. 19–33, 2019.

[7] D. Van Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Bilien, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proceedings of the VLDB Endowment*, vol. 14, no. 7, pp. 1241–1253, 2021.

[8] B. R. Barricelli, E. Casiraghi, and D. Fogli, "A survey on digital twin: Definitions, characteristics, applications, and design implications," *IEEE access*, vol. 7, pp. 167653–167671, 2019.

[9] A. Raghunandan, D. Kalasapura, and M. Caesar, "Digital twinning for microservice architectures," in *ICC 2023 - IEEE International Conference on Communications*, pp. 3018–3023, 2023.

[10] "GitHub - GoogleCloudPlatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC. — github.com." https://github.com/GoogleCloudPlatform/microservices-demo. [Accessed 02-08-2024].

[11] A. Bhardwaj and T. A. Benson, "Kubeklone: a digital twin for simulating edge and cloud microservices," in *Proceedings of the 6th Asia-Pacific Workshop on Networking*, pp. 29–35, 2022.

[12] D. Rensin, *Kubernetes*. O'Reilly Media, Incorporated, 2015.

[13] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225, IEEE, 2019.

[14] R. Sharma and A. Singh, "Getting started with istio service mesh," *Retry Requests*, p. 205, 2020.

[15] T. Fromm, "Performance benchmark analysis of istio and linkerd." https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/, May 2019.

[16] M. Yang and M. Huang, "An microservices-based openstack monitoring tool," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 706–709, IEEE, 2019.

[17] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[18] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[19] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 345–355, IEEE, 2017.

[20] J. Carlson, *Redis in action*. Simon and Schuster, 2013.

[21] F. Pourpanah, M. Abdar, Y. Luo, X. Zhou, R. Wang, C. P. Lim, X.-Z. Wang, and Q. J. Wu, "A review of generalized zero-shot learning methods," *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 4, pp. 4051–4070, 2022.

[22] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nassab, "On the nature of issues in five open source microservices systems: An empirical study," in *Evaluation and Assessment in Software Engineering*, pp. 201–210, 2021.

[23] M. T. R. Z. Alves, "Identifying logging practices in open source microservices projects,"

[24] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[25] "Google Code Archive - Long-term storage for Google Code Project Hosting. — code.google.com." https://code.google.com/archive/p/word2vec/. [Accessed 05-08-2024].

[26] I. Docker, "Docker," *lnea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.