Digital Twin and LLM Assisted Online Diagnosis in Microservices Based Enterprise Systems

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

***Abstract*—In this paper we explore a Partial Digital Twin (PDT) framework for online diagnosis of faults in Microservices (μSs) based systems. The microservices environment is highly dynamic and suffers from configuration problems due to frequent changes driven by CI/CD; therefore, quick online diagnosis is crucial but has not been addressed in the literature. We have designed the following mechanisms: (1) a conversational LLM assisted interface to automatically generate the initial "trouble-ticket" based on the problem(s) encountered by the user, (2) integration of tests for several key μSs patterns to diagnose distributed transaction failures, and (3) ensuring robustness of fault diagnosis scheme. Our methodology leverages the service mesh capabilities for μSs to dynamically reconstruct transactional contexts while incorporating a hierarchical testing strategy. Experimental results demonstrate that the framework requires only 8% more tests on average beyond the theoretical optimal for complex transaction chains. The LLM component shows an initial fault categorization accuracy of 92%, enabling precise reproduction of failure scenarios in the PDT environment without disrupting the production system.**

***Index Terms*—microservices, digital twin, root cause analysis, saga pattern, conversational AI**

I. Introduction

Modern cloud-native architectures increasingly rely on service oriented architecture (SOA) [1] and DevOps practices to achieve scalability and rapid iteration [2] which are well supported by using the concept of *service-mesh* along with μSs. However, the distributed nature of these systems introduces critical diagnostic challenges: 63% of production outages now stem from transaction coordination failures in service meshes, while 41% involve persistent data inconsistencies across sharded databases [3]. Traditional monitoring tools struggle with these issues due to the temporal nature of distributed transactions and the combinatorial complexity of microservice interactions [4].

In this paper, we explore online testing to find the root cause of the faults as and when they are reported by end users, developers, administrators, etc., still referred to as *users* for convenience. Often the reported problems affect only a few services/users, and thus we want to conduct online diagnosis without any significant impact on the production system. Towards this goal, we introduce the notion of "Partial Digital Twin" (PDT) for diagnosis purposes. A PDT essentially creates a very limited copy of the most relevant μSs and their data so that it is possible to run the tests in relative isolation. This is facilitated by the "service mesh" concept integral to μSs paradigm [5]. Such isolation not only avoids impact on the production system but also allows for testing with changes to configuration parameters or even to the code version, which wouldn't be allowed in the production system.

A crucial aspect in online diagnosis is the opportunity for an automated online dialog with the user by using the emerging large language models (LLMs) to better understand the issue and thus create an accurate *trouble ticket* used to guide the diagnosis. This helps to reduce the number of tests needed for root cause analysis. Also, the microservice paradigm is typically defined in terms of a number desired patterns, many of which are intended to enhance parallelism often at the cost of safety and robustness. Thus concurrency bugs can occur much more frequently than in traditional contexts. These patterns also involve specialized configuration parameters whose improper settings could lead to subtle problems. Thus, specialized online tests to uncover problems related to them become essential in μSs based systems. In particular, data consistency is often the most crucial issue in μS systems with SAGA based management of database (DB) transaction consistency being the most prevalent use case. SAGA is a way to manage distributed transactions by chaining local operations together. If one step fails, it runs compensating actions to undo previous steps, ensuring data stays consistent. Thus, we make the following 3 contributions in this paper:

1) Design of an *LLM conversation agent* that guides users through symptom description via structured dialogues, extracting reproduction parameters for trouble-ticket construction with 92% accuracy.
2) We demonstrate the practical advantages of our partial digital twin (PDT) implementation, specifically highlighting its ability to substantially reduce memory and storage overhead.
3) We implement detailed mechanisms to explore violation of SAGA patterns and consistency violations across multiple storage tiers (cache, DB, message queues) using differential analysis of storage layer snapshots.

Since there is no comparable (online) diagnosis mechanism in the literature, we evaluate the effectiveness of our methodology in 3 ways: (a) Number of tests required for diagnosis against the (unrealizable) ideal case where there are no wasted tests (i.e., every test brings us closer to determining the root cause), (b) Diagnosis gain by including tests for specialized μSs patterns such as SAGA, and (c) Accuracy of trouble ticket

This research was supported by NSF grant CNS-2011252

978-3-903176-75-1 ©2025 IFIP

generation and its impact on number of tests required.

We chose a medium sized, 26 μSs online shopping platform for evaluation [6]. In all cases, the average number of tests required for diagnosis were only about 1-2 more than for the ideal diagnosis. After the inclusion of saga-specific tests, the median diagnosis time decreased from 8.2 to 2.1 minutes and persistence testing reduced undetected data inconsistencies by 73%. The LLM interface itself was very effective, as it yielded a trouble ticket with 92% accuracy, as it engages the user in a dialog to clarify the initial complaint which could be quite vague. The PDT implementation also maintains operational isolation where 98% of tests do not impact live transactions.

The rest of this paper is organized as follows: Sec. II talks about the related work. Sec. III includes some basic background on μSs, misconfigurations, and PDT. Sec. IV describes the design of the LLM. Sec. V details on the PDT environment. Sec. VI presents quantitative results across multiple fault categories, followed by conclusions in Sec. VII.

II. Related Work

Although μS diagnosis has been extensively explored, nearly all of the work concerns offline diagnosis, which involves continuously collecting activity logs and analyzing those offline [7]. In a highly dynamic DevOps driven environment an online diagnosis (based on reports of problems encountered) and repair also become essential. Traditional methods, often relying on rule-based systems or statistical machine learning models, face challenges in managing the dynamic interactions and sheer volume of telemetry data in cloud-native environments [8]. While these techniques have contributed to anomaly detection and fault localization, they are limited by the need for large-scale labeled datasets or system-specific models [9]. Recent advancements in Artificial Intelligence for IT Operations (AIOps) have seen the rise of intelligent AI agents and the application of LLMs to automate complex operational tasks, including root cause analysis (e.g., RCACopilot [10]), fault detection, and incident response (e.g., IRCopilot [11]). These LLM-driven approaches demonstrate significant potential in processing unstructured operational data and generating explanatory narratives, moving towards more autonomous Site Reliability Engineering (SRE).

A critical and pervasive source of failures in μSs stems from misconfigurations, which are notoriously difficult to diagnose due to complex interdependencies [12]. While LLMs are beginning to address these challenges, showing promise in configuration validation (e.g., Ciri [13]) and even automated remediation (e.g., LLMSecConfig [14]), they introduce new concerns such as hallucinations, non-determinism, and the need for robust safety mechanisms to prevent unintended system state changes [15]. Furthermore, while LLMs have shown capabilities in log analysis and generating components of incident information [16], the direct integration of misconfiguration identification with the automated generation of complete and actionable incident tickets remains an area with significant scope for improvement. Our work aims to bridge this gap by leveraging LLMs to not only detect misconfigurations but also to seamlessly translate these findings into structured, actionable incident tickets which could then be exploited to run the most relevant tests and thereby diagnose the problem more quickly.

III. Background

A. Key Elements of Microservices Architecture

The μSs paradigm decomposes applications into independent, loosely coupled services, each of which focuses on a single functionality or encapsulates a single resource for its exclusive management. Typically both the user applications and infrastructure services (e.g., consistency/synchronization and authentication services) are implemented as μSs [17]. Each resource (e.g., a DB table, shared file, etc.) is also wrapped within an owner μS which provides read/write services for the resource. The microservice paradigm is often described in form of desired "patterns" [18]. One important pattern is SAGA, which is used to ensure consistency across updates to multiple databases. It is an alternative to the classical "two-phase commit" (2PC) [19] mechanism, and requires every transaction to follow the same update order. It uses *semantic locking* where the orchestrator sets "pending-update" flags for DB items, and all μSs need to explicitly avoid using unupdated data. Such a mechanism is both inelegant (compared with 2PC) and prone to mistakes, and thus likely to reduce robustness. In case of unexpected interference, the orchestrator runs a compensating transaction to correct data values and the affected μS must retry the operation. Other important patterns are asynchronous calls to μSs and weak consistency of updates, both of which can lead to subtle errors.

B. Managing μSs with Service-Mesh

Microservices can be scaled up by creating instances running in different containers and potentially working on different data ranges (e.g., DB shards), and managed collectively by a *service-mesh* (Fig. 1). Here we have two instances of the service in two different containers (on the same or different servers) and the desired instance is selected by the ingress gateway. The instances need not be identical which allow multiple version to run concurrently. The two dominant open-source, service-mesh implementations are Istio and Linkerd [20] with former being more capable. Both support a layer-7 SDN architecture[1] per μS, where the data plane is implemented via a *sidecar proxy* in each instance.

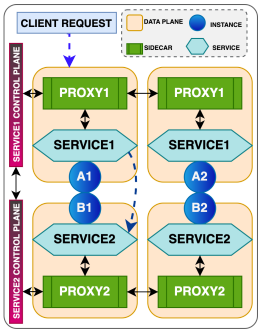

Fig. 1: Service Mesh

The proxy acts as an intermediary for the microservice by intercepting all interactions (e.g., calls to/from other μSs, storage and network IO, etc.) Two primary types of sidecars

[1]SDN (Software-Defined Networking) separates control from the underlying hardware and allows software-based management of network.

are (a) Service Mesh and (b) API Gateway. The proxy inserts itself in the data path transparently (e.g., by terminating the TCP/TLS connections), and acts on the packets via rule-matching. There is also a centralized *Service Control Plane* (SCP) that provides the rules for engaging the instances, their configuration, and handles traffic redirection in case of an instance failure. The proxy connects to the physical L2/L3 networking infrastructure via a virtual switch. Finally, *ingress/outgress gateways* apply entry/exit policies to the traffic. Ingress policies can enable load balancing across the instances. Istio provides several logging and monitoring tools. For example, we used Prometheus [21] to collect μS metrics.

C. Misconfiguration Issues in μSs Systems

The emphasis on parallelism in μS patterns and the use of settable configuration parameters for them tend to be detrimental to consistency. For example, in an e-commerce platform where Redis caching is misconfigured with an excessively long TTL, the users may see outdated product prices or inventory levels long after updates were made. Similarly, if a MongoDB cluster's read preference is set to secondary without proper awareness of replication lag, customers might place orders based on stale product availability data, leading to overselling [22]. In payment processing systems, overly aggressive saga timeout settings could prematurely trigger compensation logic, causing completed transactions to be erroneously rolled back. For a Cassandra cluster tuned for "ONE" consistency might return different results to concurrent queries [23], while a misconfigured Kafka consumer with an insufficient idempotency window could process duplicate messages [24], creating phantom orders. Istio's "circuit breaker" thresholds, if improperly calibrated, might interpret normal latency spikes as failures and isolate healthy μSs.

D. PDT Based Online Root Cause Analysis

To support arbitrary but isolated online testing, we explore the concept of a PDT —a lightweight, on-demand replica of the most relevant μSs and data used for their online fault diagnosis. This approach is easily supported by the instance replication capability of service mesh. The tailored testing sequences, informed by hierarchical fault categorization, can localize misconfigurations with high accuracy while maintaining low overhead. While we explored the PDT concept in our prior work in a simple setting [25], we now add several components to make it into a comprehensive, online fault localization mechanism in μSs mesh environment.

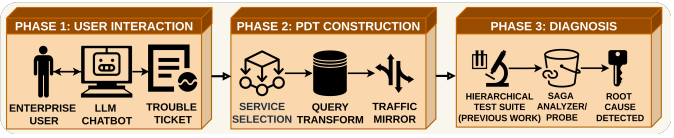

Fig. 2: Overview of framework to report and diagnose faults

Fig. 2 depicts our overall architecture which operates through three distinct phases: (i) Enterprise users interact with an LLM-powered chatbot that conducts structured dialogues to extract symptom details and automatically generates trouble tickets with 92% accuracy. We use a fine-tuned Mistral-7B model that classifies faults into eight categories through conversational refinement; (ii) PDT Construction focuses on building the partial digital twin environment through three key components: service selection, query transformation, and traffic mirroring. In our previous work [25] we showed how the system selects the most relevant microservices by considering historical error rates, call frequencies, and transaction call graphs. It then applies query transformation techniques to create compressed data subsets that maintain statistical properties of the original datasets. Traffic mirroring via Istio ensures that the PDT remains synchronized with production traffic while operating in complete isolation from the live system. This foundational approach already demonstrated efficiency, requiring only about 2 more tests than the theoretical optimal for complex transaction chains; (iii) Diagnosis encompasses the actual fault localization process through a hierarchical test suite, SAGA analyzer, and PPE. The system executes targeted diagnostic tests ordered by relevance scores determined through zero-shot learning (ZSL), while the SAGA analyzer reconstructs distributed transaction workflows from Istio trace data to identify transaction coordination failures. The PPE validates data consistency across multiple storage tiers (caches, databases, message queues), and detects the root cause through systematic execution of the test suite.

IV. LLM Based Initial Fault Classification

A. LLMs for Trouble Ticket Generation

The emergence of LLMs has opened new frontiers in automating and assisting the fault diagnosis process. Unlike static rule-based systems, LLMs can parse unstructured problem descriptions provided by users, ask contextually relevant follow-up questions, and iteratively refine their hypotheses. This capability helps in designing an LLM agent that can hold a conversation with the user to clarify the problem being encountered, obtain additional relevant details, and ultimately generate an accurate trouble ticket, similar to or better than the one that is traditionally generated manually. The generated trouble ticket can contain the following elements:

- Problem summary and classification (e.g., network-level failure in a microservice).
- Contextual information gathered through the conversation (user environment, service name, timestamps).
- Confidence level of fault classification.

Such automation significantly reduces mean time to detect (MTTD) and mean time to resolve (MTTR) by bridging the gap between problem detection and actionable response. By maintaining conversational logs and user interactions, the chatbot system creates an auditable trail of diagnostic reasoning, which can be invaluable for compliance and even future analyses. The integration of AI-driven conversational agents with incident management systems (IMS), such as Jira, can further enhance operational efficiency. Our trouble ticket generation comprises the following key components:

- User Interaction Layer: A chatbot interface designed to gather natural language fault reports.
- Classifier: Predicts fault labels with a fine-tuned LLM.
- Follow-up Question Engine: Dynamically generates relevant probing questions based on predicted fault category.
- Trouble Ticket Generator: Compiles structured reports for issue tracking systems (e.g., ServiceNow, Jira).

This modular design ensures adaptability and allows each component to be evaluated or replaced independently.

B. Designing LLM Agent

The LLM component operates in two stages: an interactive dialog stage to clarify symptoms followed by a stage where a crude classification is generated by the LLM agent. Based on the types of faults typically encountered, we use the following fault types for this classification: UN (Unresponsive Network), SN (Slow Network), US (Unreachable Service), SS (Slow Service), DE (Data Error), DC (Data Corruption), UA (Unauthorized Access), and BA (Blocked Access). The LLM is trained to output exactly one of these labels given an input query. We utilize Mistral-7B-Instruct-v0.1, a causal language model optimized for instruction-following tasks. To adapt it for fault classification, we perform parameter-efficient fine-tuning using the Low-Rank Adaptation (LoRA) technique. The training dataset is structured in JSONL format, with each record containing a user-generated natural language fault report (instruction) and its corresponding fault class abbreviation (output). The fine-tuning dataset contains instruction-output pairs specifically designed for parameter-efficient fine-tuning of the Mistral model using LoRA. It includes variations like 'Payment fails with timeout'→'SS', 'Wrong inventory count displayed'→'DE'. Instruction tuning is facilitated through prompt formatting, where each input is encapsulated in a structured template. A representative example is as follows:

```
<s>[INST] <<SYS>>
  You are a technical support engineer. Classify the fault
      type from user reports. Return only the correct fault
       abbreviation from: [UN, SN, US, SS, DE, DC, UA, BA].
<</SYS>> <fault report> [/INST] [<label>]</s>
```

It is passed through a tokenizer augmented with special tokens corresponding to each fault label. The model is resized accordingly to accommodate these new tokens. Fine-tuning is configured with LoRA parameters set to a rank of 64, scaling factor of 128, and a dropout rate of 0.05. Target layers for LoRA injection include the query, key, value, and output projections in the transformer attention mechanism [**?**]. The optimizer used is `paged_adamw_32bit`, with a cosine learning rate scheduler and a warmup ratio of 0.03. Training is conducted over 3 epochs using a gradient accumulation strategy to simulate larger batch sizes in constrained environments.

The performance of LLMs in classification tasks is highly sensitive to prompt design. In our system, instruction tuning is employed to align the model's behavior with the classification objective. Prompts are explicitly formatted to include system-level role instructions and label constraints. Token-level control is exerted by injecting task-specific tokens such as [UN], [DC], etc., directly into the tokenizer's vocabulary. These tokens are used both during training and inference, enabling the model to predict labels as discrete token completions. The instruction format minimizes generation entropy and constrains the output space, reducing hallucination and enhancing model reliability. Upon predicting a fault class, the chatbot initiates a follow-up dialogue to verify or refine the classification. For instance, if the predicted class is UN (Unreachable Network), the system asks questions such as "Are you unable to access the entire application or specific parts of it?", "Does refreshing or restarting the app help?", and "Can you access other websites on the same network?". Each class is associated with a curated question set, stored in a structured format and selected dynamically. These follow-up interactions emulate the diagnostic process of a human expert and help in narrowing down the root cause more effectively.

Algorithm 1 shows the overall algorithm for user conversation and trouble ticket generation. The BuildPrompt function encodes the conversation history into a structured format with system instructions. We include special tokens for fault labels and prompt the model to ask yes/no or multiple-choice questions. The loop ends when the classifier's confidence exceeds a threshold or no new information is gained. At the end, a fine-tuned BERT encoder model (or the last step of the decoder) predicts the final fault label. This hybrid design combines the LLM's ability to conduct natural conversation with the precision of an encoder classifier. The module correctly classified 92% of the samples, significantly higher than the BERT-only baseline. We can formalize the classification subtask as mapping a conversation context C to a label y. Let the dialog be a sequence $C = (u_1, q_1, u_2, q_2, \dots)$ of user answers u_i and system questions q_i. We train an **LLM** M to model $P(q_i \mid C)$ for generating the next question, and an **encoder** E to compute $P(y \mid C)$. The pipeline alternates between:

- **Question Generation**: $q_t = \text{argmax}_q \, P(q \mid C_{t-1}; M)$.
- **User Response**: Append the actual user reply to C.
- **Classification**: After T turns, predict $y^* = \text{argmax}_y \, P(y \mid C_T; E)$.

This design allows the model to iteratively focus the user on the most relevant symptom questions (e.g., "Is the microservice returning error 503 or 504?") before final categorization.

V. Construction of PDT and Its Use in Diagnosis

This paper builds on our initial PDT design in [25], which we describe briefly before detailing the SAGA analysis which is the main contribution of this paper.

A. Overview of Diagnosis Procedure

To build PDT we exploit Istio features to replicate a limited number of μSs and portions of databases used by them. To

[2] **BuildPrompt:** Constructs the LLM prompt by combining: (a) a static role description, (b) a truncated or summarized conversation history to fit context window limits, and (c) optional diagnostic guidance if available.

[3] **ConvergenceReached:** Returns `True` if either: (1) `MAX_TURNS` is reached, (2) LLM confidence is high and sufficient information is available (via a custom heuristic), or (3) the user indicates completion (via keywords like "done", "no more info", etc.).

Algorithm 1 Fault Reporting and Diagnosis with LLM

```
conv_hist ←; MAX_TURNS ← 5; LLM_CONF_THRES ← 0.8;
turn_count ← 0; curr_llm_pred ←
NULL; curr_llm_conf ← 0.0; prev_llm_guidance ← ""
repeat
    turn_count ← turn_count + 1
    prompt ← BuildPrompt²(conv_hist, prev_llm_guidance)
    llm_response ← Mistral.generate(prompt)
    conv_hist.append(llm_response)
    user_input ← GetUserInput()
    conv_hist.append(user_input)
    if turn_count ≥ 3 and (turn_count mod 3 = 0 or turn_count =
    MAX_TURNS - 1) then
        (curr_llm_pred, curr_llm_conf) ← Mistral.predict(conv_hist)
        if curr_llm_conf < LLM_CONF_THRES then
            prev_llm_guidance ← "The system is uncertain about the diag-
            nosis. Ask more targeted questions to differentiate between " +
            GET_TOP_UNCERTAIN_LABELS(curr_llm_pred)
        else
            prev_llm_guidance ← "The system is leaning towards " +
            curr_llm_pred + ". Consider questions to confirm/refine details."
        end if
    end if
until ConvergenceReached³(conv_hist, turn_count,
MAX_TURNS, curr_llm_pred, curr_llm_conf, user_input)
final_label ← Mistral.predict(conv_hist)
return final_label
```

identify μSs to replicate we assign a *fault-score* for each μS based on the analysis of performance metrics obtained from Prometheus, such as *historical error rate* (HER), *historical call frequency* (HCF), and *transaction call graph* (TCG). 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. Then starting with the μS that encountered the problem initially, we follow the chain of μSs invoked and add the ones with highest fault score to the PDT cache. Each round of addition is followed by the diagnosis procedure so that further addition happens only if the root cause cannot be determined by testing the μSs added so far.

Since replicating a lot of data is expensive, we use an innovative schemes that we call *query transformation* (QT). The idea is to judiciously choose entries from large tables, and alter the queries on the fly to refer to only the limited data range stored in PDT. Joins present a significant challenge in this regard since we ideally want the similar join selectivity across the compressed tables as the original. Our current solution is based on the detailed knowledge of the tables used in application. We use *stratified sampling* of tables, where data is divided into homogeneous subgroups (strata) based on important attributes like time ranges, transaction amounts, or customer demographics. For queries that involve summary statistics, we aggregate the tables to ensure that the statistics will be similar to those using the real tables. The data to be included is determined by analyzing the queries and choosing the tuples that are most frequently invoked. We then implement in-memory caching using a Redis cache server. Our PDT employs polyglot persistence with Redis caching, MongoDB for semi-structured data, and PostgreSQL for ACID-compliant operations, choosing optimal stores based on consistency requirements.

Our diagnosis is triggered by a reported problem in the trouble ticket, followed by any addition to the cached μSs as described above. The test set is determined for each fault-type from a neural net. The neural net is constructed using ZSL [26] to assign a relevance level to each test to the fault category. The tests are run sequentially in the order of decreading relevance. If all such tests are used up without finding the misconfiguration, we add more μSs to the PDT as described above and repeat. Thus in the worst case, it is possible that all μSs need to be replicated but this is extremely unlikely and never observed in our tests.

B. Analysis of SAGA Issues

Once a potential fault category is identified (especially if it involves transaction anomalies), we invoke the **Saga Analyzer** to examine distributed transaction state. The Saga Analyzer gathers all relevant trace spans associated with a particular transaction, which provides details such as the service name, event timestamp, operation performed (e.g., "Reserve Items"), and its outcome (success or failed) as shown in Fig. 3. These individual events are then sorted by their logical time, ensuring that their chronological order within the transaction is maintained. Then, they are grouped together based on the specific instance of the saga they belong to.

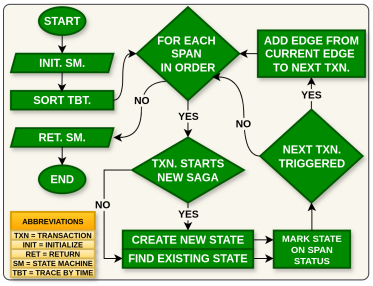

Fig. 3: SM generation flowchart

From this data, the Analyzer constructs a *State Machine*(SM), which records the progress of the overall goal (e.g., order placement). The Saga Analyzer then moves into the Failure Injection Analysis phase, where it actively simulates potential failures to pinpoint the exact step that might have gone wrong. This diagnostic process employs two key strategies. The first is Timeout Simulation. Here, the Analyzer essentially "pauses" or "aborts" a specific transaction step within a cloned version of the state machine. By doing so, it observes whether the system correctly triggers its compensating transactions, i.e., the rollback mechanisms designed to undo any changes made by preceding successful steps. If a timeout at a particular step leads to an unexpected system state or a failure to compensate, it indicates a problem with the timeout configuration or the compensation logic for that specific step. The second strategy is Compensation Replay. In cases where a step has already failed in the original trace, the Analyzer takes the constructed SM and, starting from the failed step, simulates walking backward through the transaction. During this backward walk, it executes the compensation functions associated with each preceding successful step. The goal is to see if the system successfully rolls back to a consistent state. If the rollback process halts prematurely or leaves the system in an inconsistent

state, it signals an issue with the compensation logic at that particular step. This targeted replay capability is invaluable for diagnosing complex distributed transaction bugs and cannot be done without the PDT. The Analyzer effectively narrows down issues to specifics like "Service X never received the compensating call" or "the timeout for service Y was set too short." For concept drift adaptation, the algorithms provide two mechanisms: (1) Continuous cache updates based on fault-score, which prioritizes recently problematic services, and (2) Online learning capability where new fault patterns update the relevance scores.

The system also employs the Persistence Probe Engine (PPE) to scrutinize the data layer for anomalies. This engine conducts a series of read-only tests across various persistence tiers, including caches, DB, and message queues, focusing on specific data keys that might have been affected by a transaction. For each data key under investigation, the PPE first reads its value from the cache, then from the DB, and finally peeks into any relevant message queue entries without consuming them. With this information in hand, it performs a series of tier-wise checks. For cache tiers, such as Redis or Memcached, it verifies the Time-To-Live (TTL) values and version tags. An entry is immediately flagged as "stale cache" if its TTL has expired or, critically, if its version timestamp is older than the corresponding timestamp in the DB, indicating outdated data.

When examining databases like MongoDB, the probe computes a quick checksum or hash over critical fields, ensuring it's version-aware. This allows it to detect subtle inconsistencies like partial writes or missed updates to a record, flagging them as "DB write inconsistency." For message queues like Kafka, the probe scrutinizes offsets and de-duplication markers. It can identify if a message queue has duplicate entries for a key or if there's a violation of message order, reporting these as "Queue anomaly." These probes operate with low priority, often using snapshot reads, to avoid any interference with live data. The anomalies are collected and reported per data key, which is crucial for pinpointing the exact entity whose data has diverged or become inconsistent across the distributed persistence landscape.

Table I presents an extended test suite (we already have 26 tests [25]) designed to validate the behavior and reliability of Saga-based transaction analyzers and persistence probes. The suite includes nine new tests covering log traceability (T1), compensating actions (T2), rollback consistency (T3), cache TTL integrity (T4), replica checksum validation (T5), DB versioning (T6), Kafka consumer lag (T7), replay idempotency (T8), and cache-to-database freshness (T9). Each test specifies a command and a deterministic pass/fail condition to ensure system correctness under distributed failure scenarios.

TABLE I: Extended Test Suite for Saga Analyzer and Persistence Probes

ID	Commands to Run	Pass/Fail Conditions
T1	istioctl proxy-config log <pod> –level debug	1 iff saga transaction logs captured for each step.
T2	curl -X POST <order_api>	1 iff compensating transactions invoked on simulated timeout.
T3	python saga_replay.py –saga-id=<id>	1 iff full rollback of saga confirmed in replay.
T4	redis-cli –scan — while read key; do redis-cli ttl $key; done	1 iff no stale TTLs in cache keys (TTL >0 for active keys).
T5	python db_checksum.py –table=orders	1 iff consistent row checksums across replicas.
T6	db-query "SELECT version, updated_at FROM orders WHERE id=<order_id>;"	1 iff DB record version matches latest (no partial update).
T7	kafka-consumer-groups.sh –bootstrap-server <broker> –describe –group <group>	1 iff consumer offset in sync with latest broker offset (no lag).
T8	kafka-replay-test.sh –topic=<topic> –replay-offset=<offset>	1 iff replay delivers all events exactly once (no duplicates).
T9	python cache_freshness_probe.py –key=<session>	1 iff cache value freshness < 5s w.r.t DB timestamp.

VI. Experimental Evaluation

A. Experimental Setup

We evaluate on a synthetic online shopping system consisting of 26 μSs [6]. These include services for user management, product catalog, shopping cart, order processing (with sub-services for payment, shipping, inventory, billing), as well as ancillary functions (review/blog, recommendations, analytics). Each service is a separate Kubernetes deployment with its own data store. This platform is based on an extended version of the open-source μSs-demoscaled up to cover more realistic e-commerce functions. Services communicate over HTTP/gRPC, and shared Kafka topics and Redis instances implement async workflows and caching. All inter-service traffic is managed by Istio, which provides transparent sidecar proxies. We use JMeter to generate user loads. Our deployment uses 3 microservice replicas per service; we do horizontal autoscaling 1-5 pods based on CPU>70%. JMeter generates 1000 req/s baseline with 2500 req/s bursts. Fault injection follows production incident distributions: 40% network, 35% configuration, 25% data corruption.

Scenarios include browsing products, adding to cart, checking out, and administrative tasks (stock replenishment). We generate a realistic mix of steady background load and burst events (to mimic holiday spikes). While the system runs under these conditions, we inject faults. We implement Istio fault-injection (HTTP aborts or delays) to simulate network latency or errors; custom scripts to corrupt data in the DB; and disabling Kafka brokers to drop messages. Each fault injection is tagged so we know ground truth. The PDT is deployed alongside production: for each production request of interest, Istio's traffic mirroring feature replicates the request into the PDT namespace. The mirrored requests are routed to the replicated service instances in the PDT (which run the same code and have a subset of data). This way, each real user interaction can be played in the twin for diagnosis without affecting the real user. Test scenarios cover a wide fault space. Saga faults include introducing delays in a single service (to trigger compensation paths), failing a compensation step, or causing multi-step timing anomalies. Persistence faults include injecting outdated TTLs in Redis (to cause stale reads), corrupting a MongoDB document (to simulate phantom reads), and duplicating Kafka offsets. We also test independent faults

like a service crash (generic service-unavailable), for which the LLM should classify as "Unreachable Service."

B. Rationale for Partial Digital Twin

The key purpose of PDT mechanism is to enable effective diagnosis using the recent snapshot of the system while minimizing the resource consumption. This is done via Kubernetes-driven dynamic orchestration, which selectively provisions μSs based on fault diagnosis demands rather than replicating all services in the affected transaction statically. Three architectural innovations drive this efficiency: partial data mirroring synchronizes only recent DB partitions ($\leq$72h) and active cache keys (TTL $\leq$ 300 s), reducing storage overhead by 86.3% compared to full-system cloning; tiered service prioritization classifies services into critical-path components (e.g., API gateway, always replicated), service dependencies (e.g., order-service, activated when checkout-service fails too), and low-priority modules (e.g., analytics-service, provisioned in 4% of cases); and overhead-aware scheduling leverages Kubernetes HPA to trigger provisioning only when test queues exceed five pending tasks. This approach reduces median CPU utilization by 72% (32 cores $\rightarrow$ 9 cores) and memory consumption by 85% (128GB $\rightarrow$ 19.2 GB).

TABLE II: Resource Utilization Comparison Between Full Cloning and Incremental PDT Strategies

Metric	Full Cloning	Incremental PDT	Reduction
CPU Cores	32	9 ± 1	72%
Memory (GB)	128	19.2 ± 2.4	85%
Storage (GB)	1040	142.3 ± 15.7	86.3%

The storage savings exhibit nonlinear acceleration at scale. For 78 services (3 instances of the 26 services), our proposed PDT requires merely 218 GB versus 3,600 GB under full cloning as delta snapshots and LRU cache eviction minimize redundant data replication. For instance, in our PDT, launching just five μSs requires provisioning fewer containers, initializing smaller data subsets, and activating minimal service dependencies, resulting in faster deployment and lower baseline resource consumption. By contrast, full-system cloning for an order checkout transaction replicates all 18 services (out of total 26), many of which are memory-heavy or rarely involved in faults (e.g., the analytics-service alone consumes 28 GB of memory but is relevant in $\leq$2% of cases), leading to substantial and unnecessary utilization of CPU cores and memory.

C. Initial Fault-Type Classification Effectiveness

TABLE III: Effectiveness w/o and w/ LLM fine-tuning

Parameter	w/o FT	w/ FT
Overall Accuracy	63%	92%
Ambiguity Score	58%	90%

Table III lists the overall average classification accuracy, which is high enough to be usable in practice. To address the evaluation comprehensiveness, we conducted testing on 12,400 fault reports collected from three different deployment scenarios of our baseline application. These reports were gathered from: (i) a standard deployment running under normal load conditions (n=4200), (2) a high-stress variant with 2.5x traffic load (n=4100), and (3) a multi-tenant configuration with shared databases and 3 instances of the application (n=4100). Note that we used 7B Mistral model; we expect that bigger and more advanced emerging LLM models would yield even higher accuracy. The ambiguity score measures the LLM's success rate in classifying faults from vague or imprecise user descriptions, after engaging in a clarifying dialogue. The goal of the LLM based user dialog is to accurately predict the fault-type so that the diagnosis procedure does not conduct irrelevant tests. Table IV presents the confusion matrix for this classification, where each cell represents the percentage of samples from an actual fault class that were classified into a predicted class. As seen, the diagonal elements are predominantly close to 90%, reflecting strong model performance in most fault classes. However, some off-diagonal elements reveal concentrated misclassifications. For example, the SN and SS classes exhibit mutual misclassification rates. This is likely due to symptom overlap, as both conditions often manifest elevated latency and degraded user experiences. Similarly, the confusion between DE and DC can be attributed to shared symptoms which complicates their separation based solely on observable metrics. Such confusion can actually be exploited to reduce the number of tests; for example, if the DE related tests do not reveal any problem, it would be useful to try DC tests next.

TABLE IV: Normalized Confusion Matrix for LLM Fault Classification (Actual vs. Predicted)

Real Classification \ Predicted Classification	UN	SN	US	SS	DE	DC	UA	BA
UN	93%	2%	0%	0%	0%	0%	3%	2%
SN	4%	91%	2%	2%	0%	0%	1%	0%
US	0%	2%	88%	4%	0%	0%	6%	0%
SS	0%	6%	2%	89%	0%	0%	0%	3%
DE	0%	0%	0%	1%	88%	8%	0%	3%
DC	0%	0%	0%	0%	6%	89%	0%	5%
UA	2%	0%	2%	0%	0%	0%	95%	1%
BA	2%	0%	0%	1%	1%	1%	2%	93%

Fig. 4 illustrates the relationship between the number of words in an input prompt and the latency (in milliseconds) for two different language models: BERT and Mistral. It shows that Mistral consistently has lower inference latency than BERT as prompt length increases from 10 to 50 words. This efficiency in Mistral-7B is due to architectural optimizations like the Grouped-Query Attention (GQA) and Sliding Window Attention (SWA). Its use of parameter-efficient fine-tuning (LoRA) and 4-bit quantization further contributes to its faster processing speeds and lower memory footprint, making it more suitable for real-time interactive dialogues.

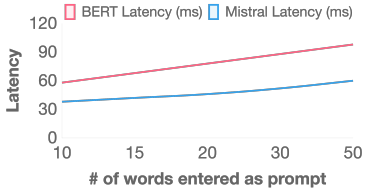

Fig. 4: Input Prompt Length (number of words) vs. Latency

D. Effectiveness of Diagnosis Procedure

The diagnosis procedure utilizes a PDT framework to identify faults without impacting the live production system. When a problem is reported, the system first identifies a subset of relevant microservices for replication within the PDT. This selection is based on a fault-score that considers historical error rates and call frequencies. Once replicated, a series of hierarchical tests are performed on these microservices within the isolated PDT environment. The selection and ordering of these tests are determined using a Zero Shot Learning (ZSL) neural network, which assesses the relevance of various diagnostic tests to the reported fault based on semantic features. The process continues until the root cause of the misconfiguration is found, or further services are added to the PDT if the initial set of tests is inconclusive [25]. Specifically, the Saga Analyzer diagnoses distributed transaction failures by constructing a SM from trace data of reported transactions. It then performs failure injection analysis through timeout simulations and compensation replays. Then it compares the outcomes to expectations to identify the faulty step. This targeted replay helps pinpoint hard-to-reproduce distributed transaction bugs. Fig. 5 reveals a critical insight into our generic diagnostic strategy which is the number of tests required to pinpoint a misconfiguration does not scale proportionally with the increasing number of μSs in the affected chain, but rather quickly plateaus. This non-linear behavior, where the curve flattens around 10-11 affected μSs and remains stable, signifies that our system efficiently localizes misconfigurations without incurring escalating testing overheads.

1) Diagnosis With SAGA Tests: The underlying reason for this plateau lies in the fundamental nature of misconfiguration faults and our intelligent diagnostic approach. Unlike complex transactional faults that might involve a cascade of failures across a SAGA, misconfigurations are often static errors like incorrect environment variables, incorrect API endpoints, or mismatched schema versions, that are highly localized and manifest predictably. Our diagnostic system is designed to leverage this characteristic by employing a highly targeted search strategy, akin to a binary search or dependency-aware probing rather than an exhaustive sweep. For instance, if a misconfigured DB connection in the 'payment-service' is the root cause, a small set of tests targeting its immediate dependencies and configuration will quickly expose the issue, irrespective of whether the overall transaction involves 5 or 15 downstream services. Adding more services to the chain beyond this point does not introduce new types of misconfigurations that necessitate a vastly expanded test suite. The existing, efficient set of tests remains sufficient to identify the limited set of common misconfiguration patterns. This demonstrates that our method effectively bounds the diagnostic effort, ensuring that as the scope of an affected chain grows, the system's ability to find misconfigurations does not degrade, maintaining a consistent and low number of required tests.

Fig. 6 illustrates the relationship between the number of steps in a saga-based transactional workflow and the corresponding number of diagnostic steps required for fault diagnosis for saga specific faults. The x-axis represents the number of μSs involved in the transactional chain, while the y-axis reports the number of test steps executed by the diagnostic framework. The graph compares two curves: the blue line represents the *Theoretical Optimal*, denoting the minimum number of tests achievable under ideal conditions (which implies using the highest test-fault relevance scores and the flowchart hierarchy for test execution demonstrated in our previous work [25]), while the red line shows the performance of the *Proposed PDT*. Notably, the PDT curve closely follows the theoretical optimal across the entire range, indicating that the tool achieves near-optimal diagnostic efficiency. Our previous work lacks saga specific tests and semantic insight into the transactional context, thus requiring broader brute-force coverage of service states and potential compensation paths. In contrast, the optimized PDT, integrated with the saga analyzer, exhibits near-linear scaling. The apparent linear increase in the number of tests required specifically illustrates the diagnostic complexity as the length of a single SAGA transaction chain increases, with the x-axis representing the number of μSs participating in that particular SAGA, not the total number of μSs in the entire system. For an ecosystem comprising hundreds of μSs, a typical fault diagnosis does not necessitate activating and testing every single one. Instead, a fault usually manifests within a specific business transaction, which might involve a SAGA spanning, for example, 5, 10, or up to 15 μSs as shown in the graph. Therefore, if a system possesses 100 μSs but a given fault occurs within a SAGA involving only 7 of them such as an order-service interacting sequentially with payment-service, etc., our Proposed PDT will operate within the complexity bounds corresponding to 7 μSs on the graph, bypassing the remaining 93 services.

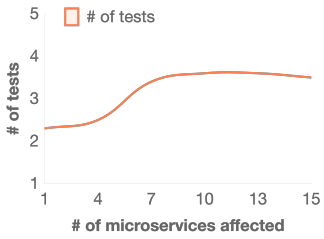

Fig. 5: # of μSs in the diagnosed chain vs # of tests required to find the misconfiguration

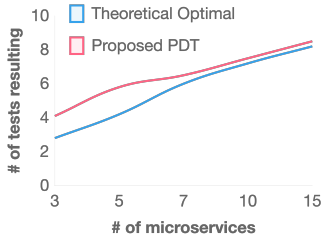

Fig. 6: # of steps for SAGA related faults for increasing number of μSs chain length

2) Performance for Different Persistent Layers: Fig. 7 provides empirical results on the diagnosis success rate of the persistence probes when applied to each memory tier. The x-axis represents the three targeted persistence layers namely cache, DB, and message queue, while the y-axis measures the number of faults correctly identified by the probes. Each bar in the chart corresponds to the count of faults detected for that memory category, derived and aggregated from repeated diagnostic runs across a diverse set of injected faults. The cache probes achieved a detection accuracy of approximately 93%. Despite their lightweight execution, these probes suffer from limitations inherent to eventual consistency and race

conditions—common in high-throughput μSs—where a cache entry might appear valid in isolation but be globally outdated.

The DB probes performed best, with a detection accuracy exceeding 94%. These tests employ version-aware checksumming and snapshot reconciliation techniques. For large record sets or under high write load, computing full-row hashes introduces a latency overhead of up to 6%, which the system mitigates through deferred checksum scheduling or sampling techniques. The message queue probes yielded the lowest accuracy, around 87%. The corresponding tests operate by scanning the consumer offsets, deduplication markers, and message ordering constraints for the target keys. To validate consistency, the system replays message consumption from a specific offset and verifies whether all expected events are delivered once and in order. Failures are recorded if duplicate messages are observed, or if certain events are missing or delayed beyond acceptable bounds. The relative weakness of this layer's detection accuracy is due to the asynchronous and distributed nature of message queues. Offset drift and network jitter contribute to anomalies that evade snapshot-based inspection, making full detection difficult in this tier.

Fig. 7: PDT performance by memory-level categorization

In the message queues, duplicates and reordered messages were produced by manipulating Kafka offset and replication configurations. After fault injection, the PDT issued the probes and logged their ability to correctly detect each anomaly based on pre-known ground truth. The results highlight the diagnostic trade-offs inherent in persistence-layer testing. While DB probes are the most robust, their computational cost is higher. Cache probes are fast but more error-prone due to their distributed nature. Queue probes struggle with detection accuracy due to the timing and ordering complexity of modern event-driven systems. These results underscore the need for multi-tiered diagnostic strategies that adjust not only their probing logic but also their statistical thresholds and frequency depending on the underlying memory model.

VII. Conclusions

In this paper, we introduced a PDT framework for online diagnosis of faults in μSs, incorporating a conversational LLM assisted interface and specialized testing. The LLM component demonstrated 92% accuracy in the initial trouble ticket generation, and the PDT implementation showed 72% reduction in CPU and 85% in memory compared to full cloning, while maintaining operational isolation. The hierarchical testing strategy, especially with the integrated saga and persistence analyzers, achieved near-optimal diagnostic efficiency, requiring only 1-2 more tests than the theoretical ideal. In the future, we will explore continuous learning mechanism and integrate sliding-window correlation analysis for detecting intermittent faults.

References

[1] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310, 2017.
[2] Mojtaba Shahin. Architecting for devops and continuous deployment. In *Proc. of ACM ASWEC*, pages 147–148. ACM, 2015.
[3] Shenglin Zhang, Sibo Xia, Wenzhao Fan, et al. Failure diagnosis in microservice systems: A comprehensive survey and analysis. *ACM Transactions on Software Engineering and Methodology*, 2024.
[4] Shenglin Zhang, Pengxiang Jin, Zihan Lin, et al. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing*, 16(6):3851–3864, 2023.
[5] Nabor Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE Software*, 38:17–22, 09 2021.
[6] "GitHub - GoogleCloudPlatform/microservices-demo" https://github.com/GoogleCloudPlatform/microservices-demo. [Accessed 02-08-2024].
[7] Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Classifying the reliability of the microservice architectures. 2022.
[8] Manish Shetty, Yinfang Chen, et al. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 99–110, 2024.
[9] Arthur Vitui and Tse-Hsun Chen. Empowering aiops: Leveraging large language models for it operations management, 2025.
[10] Yinfang Chen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688, 2024.
[11] Xihuan Lin, Jie Zhang, et al. Ircopilot: Automated incident response with large language models. *arXiv preprint arXiv:2505.20945*, 2025.
[12] Shenglin Zhang, Sibo Xia, Wenzhao Fan, et al. Failure diagnosis in microservice systems: A comprehensive survey and analysis, 2024.
[13] Xinyu Lian, Yinfang Chen, et al. Configuration validation with large language models. *arXiv preprint arXiv:2310.09690*, 2023.
[14] Ziyang Ye, Triet Huynh Minh Le, and M Ali Babar. Llmsecconfig: An llm-based approach for fixing software container misconfigurations. *arXiv preprint arXiv:2502.02009*, 2025.
[15] Michael Assad, et al. Can my microservice tolerate an unreliable database? resilience testing with fault injection and visualization. *Proc. of IEEE/ACM 46th ICSE*, pages 54–58, 2024.
[16] Zhihan Jiang, et al. L4: Diagnosing large-scale llm training failures via automated log analysis. *arXiv preprint arXiv:2503.20263*, 2025.
[17] Wubin Li, Yves Lemieux, Jing Gao, et al. Service mesh: Challenges, state of the art, and future research opportunities. In *Proc. of IEEE SOSE*, pages 122–1225. IEEE, 2019.
[18] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.
[19] George Samaras, Kathryn Britton, Andrew Citron, and C Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995.
[20] Thilo Fromm. Performance benchmark analysis of istio and linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/, May 2019.
[21] Meixia Yang and Ming Huang. An microservices-based openstack monitoring tool. In *Proc. of IEEE ICSESS*, pages 706–709. IEEE, 2019.
[22] Kristina Chodorow. *Scaling MongoDB: Sharding, Cluster Setup, and Administration*. " O'Reilly Media, Inc.", 2011.
[23] Abdul Wahid and Kanupriya Kashyap. Cassandra—a distributed database system: An overview. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2018, Volume 1*, pages 519–526, 2019.
[24] Sean Rooney, Peter Urbanetz, Chris Giblin, et al. Kafka: the database inverted, but not garbled or compromised. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3874–3880. IEEE, 2019.
[25] Sourav Das, Jit Gupta, and Krishna Kant. Online diagnosis of microservices based applications via partial digital twin. *Proc. of IEEE NCA conference*, Oct 2024. Available at https://www.kkant.net/papers/Microservices_diagnosis.pdf.
[26] Farhad Pourpanah, Moloud Abdar, Yuxuan Luo, et al. A review of generalized zero-shot learning methods. *IEEE transactions on pattern analysis and machine intelligence*, 45(4):4051–4070, 2022.