Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis

Xiaofeng Guo*
Fudan University
China

Xin Peng*
Fudan University
China

Hanzhang Wang
eBay Inc.
USA

Wanxue Li
eBay Inc.
China

Huai Jiang
eBay Inc.
China

Dan Ding*
Fudan University
China

Tao Xie*
Peking University
China

Liangfei Su
eBay Inc.
China

ABSTRACT

Microservice systems are highly dynamic and complex. For such systems, operation engineers and developers highly rely on trace analysis to understand architectures and diagnose various problems such as service failures and quality degradation. However, the huge number of traces produced at runtime makes it challenging to capture the required information in real-time. To address the faced challenges, in this paper, we propose a graph-based approach of microservice trace analysis, named GMTA, for understanding architecture and diagnosing various problems. Built on a graph-based representation, GMTA includes efficient processing of traces produced on the fly. It abstracts traces into different paths and further groups them into business flows. To support various analytical applications, GMTA includes an efficient storage and access mechanism by combining a graph database and a real-time analytics database and using a carefully designed storage structure. Based on GMTA, we construct analytical applications for architecture understanding and problem diagnosis; these applications support various needs such as visualizing service dependencies, making architectural decisions, analyzing the changes of service behaviors, detecting performance issues, and locating root causes. GMTA has been implemented and deployed in eBay. An experimental study based on trace data produced by eBay demonstrates GMTA's effectiveness and efficiency for architecture understanding and problem diagnosis. A case study conducted in eBay's monitoring team and Site Reliability Engineering (SRE) team further confirms GMTA's substantial benefits in industrial-scale microservice systems.

*X. Guo, X. Peng, and D. Ding are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China. T. Xie is with the Department of Computer Science and Technology, and the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417066

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Dynamic analysis**.

KEYWORDS

Microservice, tracing, graph, visualization, architecture, fault localization

ACM Reference Format:
Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3417066

1 INTRODUCTION

Microservice architecture has the benefits of faster delivery, improved scalability, and greater autonomy, and thus has been the latest trend in building cloud-native applications. A microservice system is implemented as a suite of small services, each running in its process and communicating via lightweight mechanisms (often an HTTP resource API) [15]. Microservice systems are highly complex and dynamic. In a microservice system, each request may result in a series of distributed service invocations executed synchronously or asynchronously. A service can have several to thousands of instances dynamically created, destroyed, and managed by a microservice discovery service (e.g., the service discovery component of Docker swarm) [21, 22].

For a microservice system, operation engineers and developers highly rely on trace analysis to understand architectures and diagnose various problems. Due to high complexity and dynamism of a microservice system, it is hard for its operation engineers and developers to use static analysis and logs to achieve the purposes. Moreover, a microservice system undergoes frequent deployments, which continuously change the dependencies and behaviors of its services. Therefore, an industrial microservice system has been commonly equipped with distributed tracing, which tracks the execution of a request across service instances. The distributed tracing system records each invocation of the service operation as a *span*

and the execution process of each external request as a *trace*, together with related properties such as latency and error status. By analyzing the trace data (including spans and traces), operation engineers and developers can understand the interactions and dependencies between services and pinpoint where failures occur and what causes poor performance [18].

However, the huge number of traces produced at runtime makes it challenging to capture the required information in real-time, with two particular challenges: (1) the trace data needs to be efficiently processed to produce aggregated trace representations of different levels and high quality; (2) detailed information of specific traces can be made available in an on-demand way. For example, in eBay, the microservice systems produce nearly 150 billion traces per day. For architecture understanding, it is thus necessary to aggregate the traces to exhibit the dependencies and behaviors of a large number of services, while at the same time revealing the changes caused by deployments and updates. For problem diagnosis, abnormal traces need to be quickly identified, and their details (e.g., related metrics such as response time and error rate) can be provided in an on-demand way.

To address these challenges, in this paper, we propose a graph-based approach of trace analysis, named GMTA, for understanding microservice architecture and diagnosing various problems. Built on a graph-based representation, GMTA includes efficient processing of traces produced on the fly. It abstracts traces into different paths and further groups them into business flows. To support various analytical applications, GMTA includes an efficient storage and access mechanism by combining a graph database and a real-time analytics database and using a carefully designed storage structure. Based on GMTA, we construct GMTA Explorer for architecture understanding and problem diagnosis, supporting various needs such as visualizing service dependencies, making architectural decisions, analyzing changes of service behaviors, detecting performance issues, and locating root causes.

GMTA has been implemented and deployed in eBay. To assess GMTA's effectiveness and efficiency, we conduct an experimental study on real trace data of eBay, including 197.89 billion spans and 10.29 billion traces. The study compares GMTA with two traditional trace processing approaches both qualitatively and quantitatively. We derive six trace analysis scenarios that require trace data accesses of different levels (i.e., trace, path, business flow). The results show that GMTA can effectively support all these scenarios and its effectiveness substantially outperforms the two traditional approaches. We present GMTA Explorer based on GMTA to the monitoring team and Site Reliability Engineering (SRE) team of eBay and conduct a case study in real tasks. The results further confirm GMTA's substantial benefits in industrial-scale microservice systems.

2 BACKGROUND AND MOTIVATION

With the microservice trend, one modern distributed system can generally involve hundreds or thousands of microservices. The complex calling relationship of these microservices makes it difficult to conduct development, management, and monitoring for the system. The traditional operation mechanisms of machine-centric monitoring is not effective, due to the lack of a coherent view of the work done by a distributed service's nodes and dependencies. To address this issue, end-to-end tracing based on workflow-centric tracing techniques [5, 8, 18] is proposed in recent years. The basic concept of tracing is straightforward: instrumentation at chosen points in the distributed service's code produces data when executed, and the data from various executed points for a given request can be combined to produce an overall trace. For example, for a request-based distributed service, each trace would show the work done within and among the service's components to process a request. Since end-to-end tracing captures the detailed work of the causally-related activity within and among the components of a distributed system, there are a growing number of industry implementations, including Google's Dapper [18], Cloudera's HTrace [5], Twitter's Zipkin [8], etc. Looking forward, end-to-end tracing has the potential to become the fundamental substrate for providing a global view of intra- and inter-data center activity in cloud environments.

However, leveraging tracing data for monitoring and architecture understanding faces two major challenges. First, it is challenging to process, store and analyze the huge real-time trace data efficiently, given that with the increase of a microservice system's scale, the quantity of trace data increases dramatically. Second, building effective applications using trace data needs to address the quality issues of the trace data such as wrong chain and broken chain, given that the large service ecosystem can consist of various application frameworks and different systems.

To address the preceding two challenges, we propose GMTA and GMTA Explorer to support microservice monitoring and troubleshooting. After the basic distributed tracing implementation is in place, it is valuable to leverage a large number of distributed tracing data for insights. Especially for developers and SREs, observing applications' behaviors and analyzing them enable to gain more knowledge to do troubleshooting, and even business analysis. In 2019, there are a few millions of lines of code and config changes for the whole eBay service ecosystem. It is almost impossible to capture or understand these changes based on only design documents or domain knowledge.

3 GRAPH-BASED MICROSERVICE TRACE ANALYSIS (GMTA) SYSTEM

Figure 1 presents an overview of the Graph-based Microservice Trace Analysis (GMTA) system. It includes three modules: processing, storage, and access. The processing module takes as input raw data, i.e., span logs obtained from the distributed tracing system, and assembles spans into traces, paths, till business flows. These processing results are persisted in the storage for further analysis. To support flexible and efficient trace data access, the storage module combines both the graph database and real-time analytics database and uses a carefully designed storage structure. The access module provides flexible and efficient trace data access interfaces at three levels: the business flow level, path level, and trace level.

Trace data are a kind of streaming data that are generated continuously by a huge number of service instances. The overall design principle of the system is flexibly combining graph and non-graph based trace data processing and storage for efficient access of a vast number of trace data of different levels.

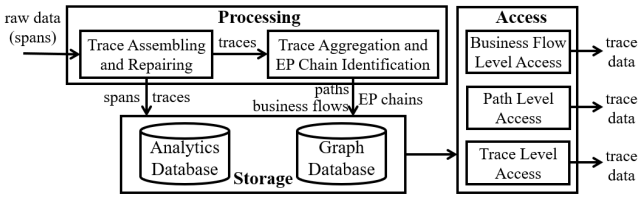

Figure 1: GMTA System Overview

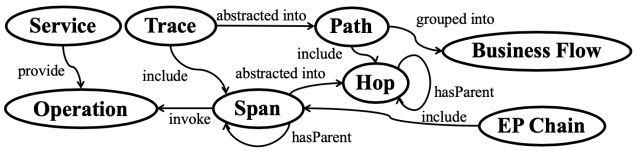

Figure 2: Graph-based Trace Data Representation

In the rest of this section, we first introduce the graph-based representation of trace data of different levels and then detail the three modules (processing, access, and storage).

3.1 Graph-based Representation

Our graph-based system of microservice trace analysis is built on a series of concepts about trace and flow analysis. These concepts and their relationships are described in the conceptual model shown in Figure 2.

A service provides a set of operations and each invocation of an operation is a span. Each span has a unique ID and a set of properties such as invoked service and operation, start time and duration and parent span's ID. A trace represents the execution process of an external request. Each trace has a unique ID and a tree structure consisting of spans; the parent-child relationship in the tree structure represents the service invocation relationship. The traces that have exactly the same tree structures (i.e., the same service operations and invocation orders) can be abstracted into a path. A trace thus can be understood as an instance of a path. A path has a tree structure consisting of hops; each hop is abstracted from the corresponding spans in the traces. The paths that implement the same scenario can be further grouped into a business flow. Therefore, a trace type can be regarded as a variant of a business flow, and the paths of a business flow usually can be selected based on some key service operations that are involved. Usually a business flow can specify the key operations that its paths must go through or further specify the execution order of these operations.

Figure 3 shows an example business flow for placing order, which specifies "createOrder" as its key operation. In the figure, an ellipse represents a service, a rectangle represents an operation, a thick arrow represents a hop, and a thin arrow represents a span. The business flow includes two paths, i.e., guest checkout and user checkout, which consist of a series of hops with the red and green colors, respectively. Each path has a number of traces as its instances. For example, the guest checkout path has two traces consisting of a series of spans with the blue and black colors, respectively. Note that some of their spans are not depicted due to space limit.

The two paths of the business flow are grouped together based on the identified key operation (here "createOrder"). Both of them have a tree structure and differ only in the checkout operation ("guestCheckout" or "userCheckout"). The guest checkout path has

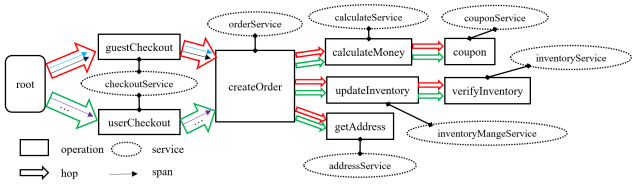

Figure 3: An Example of Business Flow (Place Order)

two traces. They have exactly the same tree structure and record two execution instances of the path. For example, the two spans between "guestCheckout" and "createOrder" have different trace IDs, span IDs, timestamps, and durations.

To support error propagation analysis, we also define the concept of error propagation chain (in short as EP chain), which is a sequence of spans that propagate errors. For example, the EP chain "coupon → calculateMoney → createOrder" indicates that an error is propagated from "coupon" to "createOrder".

3.2 Data Processing

The span logs of a trace are produced in a distributed way by the service instances involved in the trace. The processing module of GMTA assembles these spans into a trace, and further cleans, aggregates, and analyzes the trace. All the preceding processing is performed in a streaming way, which can be implemented based on streaming processing frameworks such as Flink [2].

3.2.1 Trace Assembling and Repairing. To assemble a trace, we need to collect all its spans and connect them according to the parent span ID recorded in each span. Trace assembling is a continuous process during which a huge number of spans from different service instances are produced and received in a streaming way. To efficiently handle the streaming data, we adopt a time window strategy. We collect and group spans that have the same trace ID and can be reached in a given time window (e.g., 5 minutes) based on the assumption that all the spans of a trace can be produced and received in a short time. Then for each group of spans, we try to assemble a trace in memory.

Span logs are generated by a large number of services, which are developed by different developers and collected through a complex distributed network. Therefore, some span logs may include incorrect or incomplete information, making the derived traces invalid. Currently GMTA detects and repairs two kinds of such problems: invalid operation name and broken trace.

Invalid operation name is usually caused by incorrectly passed parameters in logging, e.g., the user ID included in a request may be passed as a part of operation name by mistake. The number of operations of a service is usually stable and not large. Therefore, we can identify invalid operation names in span logs by monitoring the changes of the numbers of different operation names of the same services. We prompt users to check the identified invalid operation names and ask the users to provide matching and replacement rules, e.g., using regular expressions. Then invalid operation names that arrive later can be automatically matched and replaced with correct names. In this way, false paths caused by invalid operation names can be avoided.

Broken trace is usually caused by incomplete data passing in logging because of development specifications and historical legacy

Algorithm 1 SpanHash(span)

```
1: hash = HashCode(span.serviceName) + HashCode(span.operationName)+span.level×PRIME_NUMBER
2: for each child in span.childSpans do
3:     hash+ = SpanHash(child)
4: end for
5: return hash
```

systems, e.g., the parent span ID is missing in the span. Based on the tree structure of an assembled trace, we detect and repair three kinds of broken traces:

(1) a trace has no root;

(2) a trace has more than one root;

(3) a trace has a root but some spans have no parent spans.

For the first case, we simply add a root node and make it the parent of the spans that have no parent spans. For the second and third cases, we try to repair the broken trace by timestamp matching. Given a span that has no parent span or the root of a subtree S, we try to find a parent span for it in the following way: if there is a leaf span P in other subtrees meeting the condition that the start time and end time of S are during the duration of P, P is regarded as the parent span of S, and the two subtrees are thus connected.

During trace assembling, we also recognize EP chains based on the error tags of spans. Given a span with an error tag, we examine whether one of the span's child spans also has an error tag. If not, the current span is regarded as the starting point of the EP chain, and a special tag and the length of the chain are set into the span.

3.2.2 Path Identification. Path identification is continuously performed together with trace assembling. Different from traces, the number of paths is relatively stable. A new path usually appears only when the system is updated or errors/exceptions occur in service invocations. Path identification requires to check whether two traces have the same tree structure. To avoid the expensive tree comparisons between traces, we generate for each trace a path ID that can uniquely identify a path. The path ID is generated by computing a hash code for the root span using Algorithm 1. Given a span, the algorithm computes a hash code for its service name and operation name, respectively, and then adds up the two hash codes and an offset computed based on the level of the span in the trace tree. If the span has child spans, the algorithm recursively computes a hash code for each child span and adds them up together.

Given a trace, we compute a path ID for it and check whether the path with the same ID already exists. If the path exists, we update its properties such as trace count, average latency, and occurrence times of different EP chains. If the path does not exist, we create a new path for the trace. For example, the two traces inside the red arrow in Figure 3 get the same path ID because the operations and sequences passed are exactly the same. When the trace represented by the blue arrow appears, because this path ID appears for the first time, we record the operation and related information passed by the current trace as a new path. But when the trace represented by the purple arrow comes, the path ID already exists, so we update the corresponding path information according to the current trace properties.

3.2.3 Business Flow Identification. Business flows can be defined by developers and operation engineers in an on-demand way. For example, besides the business flow for placing order (see Figure 3), a developer can define a business flow for updating inventory to examine all the paths that involve the invocation of the "updateInventory" operation. By analyzing this business flow, the developer may find clues for a fault about inventory updates.

A business flow can be defined as a logical combination (using AND/OR operations) of any number of basic conditions of the following two types:

(1) a service operation is invoked;

(2) a service operation is invoked before or after another one.

The identification of business flows involves the examination of the existence of specific service invocations and the order of specific service invocations in a path. This examination can be efficiently supported by graph databases such as Neo4j [7]. Therefore, we continuously store and update the identified paths in the graph database of the storage module and use graph queries to dynamically identify paths that meet the definition of a given business flow.

3.3 Data Access

The objective of GMTA is to provide efficient and flexible access supports for various trace analysis applications such as architecture understanding and problem diagnosis. The trace data access requirements can be categorized into the following three levels.

(1) Trace Level Access. Trace level access includes trace searching and trace detail query. Trace searching searches for all the traces that involve the invocations of specific service operations or occur within a given time range. Trace detail query requests various types of information of a given trace (usually by trace ID), including its spans and related metrics such as duration. The query may also request extended information of a trace such as its path ID and involved EP chains.

(2) Path Level Access. Path level access includes path searching and path detail query. Path searching searches for all the paths that involve the invocations of specific service operations. Path detail query requests various types of information of a given path (usually by path ID), including its traces within a given time range, trace count, and other aggregated metrics such as error rate and average latency. The query may also request extended information of a path such as involved EP chains and occurrence times.

(3) Business Flow Level Access. Business flow level access includes business flow searching and business flow detail query. Business flow searching searches for all the business flows that involve specific paths. Business flow detail query requests the paths of a business flow (usually by business flow name) within a given time range.

For each of the preceding requirements, a basic access interface can be defined and implemented to support efficient access of specific trace data. Besides these basic interfaces, other more specific access interfaces can also be defined and implemented based on our graph-based representation of trace data. Although some trace data access requirements can also be satisfied by combining basic access interfaces, it may be more efficient to design specific interfaces for them. For example, a trace analysis requirement may be finding all the paths in which a service directly or indirectly invokes another

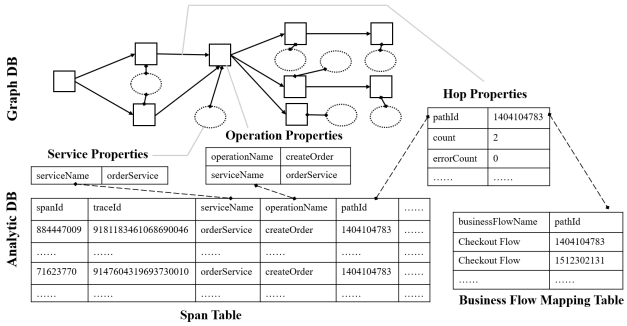

Figure 4: Data Storage Structure

service in n hops. This requirement can be implemented by using the basic access interfaces: we first obtain all the paths that include both of the two services, and then filter out the paths in which the invocation between the two services is beyond n hops. If we design a specific interface for this requirement, the trace data access can be much more efficient, as the paths that meet the condition can be directly obtained by graph query.

3.4 Data Storage

Trace data processing and analysis involve graph-based queries such as searching for paths that involve the invocations of specific services. This kind of operation can be efficiently supported by graph databases such as Neo4j [7]. At the same time, trace data processing and analysis also involve real-time analytics of non-graph-based data such as selecting the spans of a specific service operation in the last one hour and grouping them by path ID. This kind of operations can be efficiently supported by real-time analytics OLAP (On-Line Analytical Processing) databases such as Druid [1] and Clickhouse [4]. Therefore, a key design decision for the storage module is how to distribute trace data between the two kinds of databases.

All the traces of the same path share exactly the same structure, and paths are much more stable than traces. These features imply that trace level graph queries (e.g., finding all the traces that involve a specific service invocation) can be implemented by corresponding path level graph queries. Based on this observation, we design a mixed storage structure as shown in Figure 4. The figure shows the graph structure of the graph database and the main tables of the analytics database. The figure also shows the main properties of the nodes and edges in the graph database together with their references to the fields in the analytics database.

We store the relatively stable part of the graph-based representation (see Figure 2) in the graph database, including services, operations, hops, and paths. There can be multiple edges between two service operations, and each edge represents a hop of a specific path. Therefore, the hops of a path can be obtained by a graph query using the path ID to filter the graph. We store detailed information of trace spans in the analytics database, such as span ID, trace ID, service name, operation name, timestamp, duration, and error tag. The analytics database should support real-time querying of the full scope of data in a certain period of time or automatic aggregation according to specified fields. Each span records its path ID to enable the selection of spans and traces of a given path. As a business flow represents a group of paths, we store the mappings between business flows and paths in the analytics database.

All the preceding data are continuously updated in trace data processing. When a trace is assembled, its spans are added into the span table and its path is analyzed. If a new path is created, its hops and possibly new services and operations are added into the graph database. Periodically business flow identification is conducted by querying the paths in the graph database, and the identified business flows are updated into the business flow mapping table in the analytics database. To support efficient data query, we add some aggregated metrics such as the trace number and error count of a hop as the hop properties in the graph database.

Our current implementation of GMTA uses Neo4j [7] as the graph database and Druid [1] as the analytics database.

Table 1: GMTA Explorer Functionalities

Functions	Sub Functions	Trace	Path	Business Flow
Basic Visualization	Trace View	√	×	×
	Path View	√	√	×
	Business Flow View	√	√	√
Change Visualization	Path Comparison	×	√	×
	Business Flow Comparison	×	√	√
Anomaly Comparison	Path Comparison	×	√	×
	Business Flow Comparison	×	√	√
	EP Chain Comparison	×	√	√
EP Chain Query	EP Chain Query	×	√	√

4 ARCHITECTURE UNDERSTANDING AND PROBLEM DIAGNOSIS

By leveraging GMTA, we can build GMTA Explorer in eBay; GMTA Explorer is designed to help developers and operation engineers within the company quickly understand architecture and improve the efficiency of problem diagnosis. Table 1 shows the functionalities provided by GMTA Explorer and the corresponding levels of trace data access interfaces provided by GMTA. Among these functionalities, basic visualization and change visualization serve for architecture understanding; anomaly comparison and EP chain query serve for problem diagnosis.

4.1 Architecture Understanding

GMTA Explorer supports the following three use cases of architecture understanding:

- U1. as developers, to visualize the dependencies and dependents of a service in a business flow to determine the change impact of the service.
- U2. as architects, to learn the paths and metrics related to critical businesses (e.g., payment) to support architectural decisions.
- U3. as SREs, to confirm the changes or patterns of service behaviors, and evaluate the impact of business changes or other factors on metrics such as traffic, latency, and error rate.

For U1 and U2, by calling GMTA to provide trace level, path level, and business flow level query interfaces, GMTA Explorer can provide interactive visualizations, allowing the users to drill up and down between the three levels. For example, the users can choose to examine the metrics (e.g., traffic, duration, and error rate) of a specific path in a business flow, and then drill down into a specific trace of the path through condition-based filtering. The users can also start from a certain operation, examine the paths through this operation, and then pick a specific path to get the business flows that it belongs to.

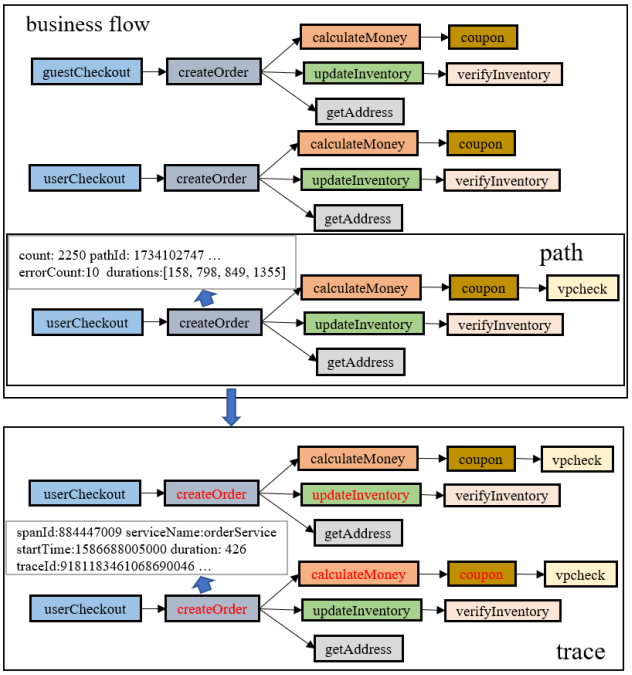

Figure 5: Business Flow Example

Figure 5 shows three views of GMTA Explorer. In the business flow view, each tree represents a path, each rectangle represents a hop in a path, the color of the rectangle represents a service, and the rectangles of the same color belong to the same service. The users can click a hop to see the detailed properties of the corresponding hop in this flow. The users can also enter the trace view below by double-clicking the path. In the trace view, each tree represents a trace, and each rectangle represents a span. The color meaning is the same as above. The users click a rectangle to see the detailed properties of the span. The red operation name indicates that there is an error in the current span. With the help of GMTA Explorer, the users can quickly understand the dependencies between services and operations in a business flow and enter the path view or trace view by filtering to obtain more detailed metrics. This feature can also be used later to discover technical debt and architectural antipattern (e.g., cyclic dependency and god service).

For U3, GMTA Explorer queries the path/business flow information in the two time periods specified by the users, and visualize the changes in the two time periods through graph-based comparison and analysis, so that the users can understand business changes and path changes. Through the comparison of business flow within two time periods, relevant personnel can not only understand the existing business processes but also compare the path changes. For example, the traffic may disappear within a subset of a path and switch to another one. The changes can be visualized by doing a graph comparison between snapshots of the same business flow.

Figure 6 shows an example of business flow changes. The upper part represents the path of this business flow. It can be seen that in this business flow there are three paths (each of which corresponds to a hop), the color of the rectangle represents the latency change of the two periods before and after, and green represents the decrease of latency, the red represents the increase, and the darker the color,

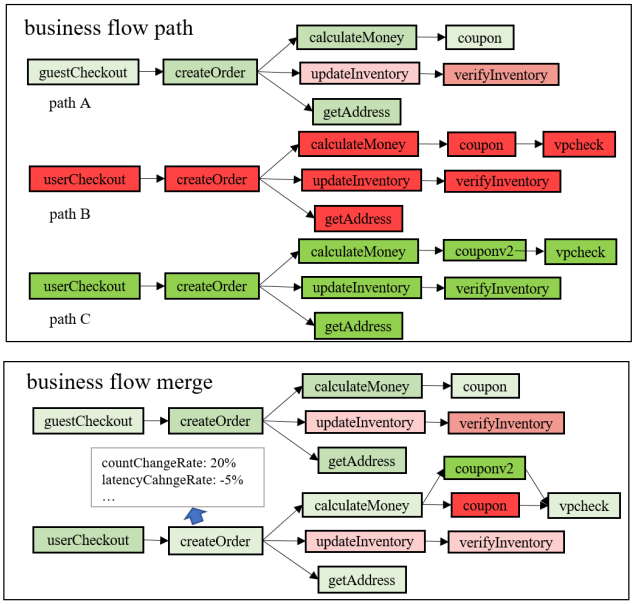

Figure 6: Business Flow Comparison Example

the greater the change. It can be seen from the figure that the operation latency of Path A has partially increased or decreased, path B is all dark red, indicating that the path has disappeared in the new time period, and the dark green path C indicates that the path is new; we can clearly know that path C replaces path B in the new time period. But unlike the preceding simple example, a business flow may contain dozens or hundreds of paths in the actual production environment, so it is sometimes difficult to obtain effective information by directly displaying the paths. So under normal circumstances, we group the paths of the same business flow according to the root, and then merge the same operation in each group, and we also merge the corresponding attributes. The users can also click a box to view the details of the corresponding operation such as the change rate of flow rate and change rate of error rate. Finally, we show the results of the comparison of the two business flows after the merger. The effect is shown in the lower part of Figure 6. This view can more intuitively show the changes that occur in a business flow. From Figure 6, it can be seen that the "coupon" called by "calculateMoney" has been updated to "couponv2" (in the paths) that starts with "userCheckout". This view is more clear and concise, and it is convenient for the users to see the changes in business flow in different time periods more intuitively.

The functionalities of trace, path, and business flow visualization and path/business flow comparison provided by GMTA Explorer can support the preceding three use cases, help developers, SREs, and architects to quickly understand service dependencies, and obtain service and operation runtime information and confirm business changes.

4.2 Problem Diagnosis

GMTA Explorer can help the users locate the scope and confirm the root cause of a problem. In large enterprises, the root cause of the problem needs to be located and repaired as soon as possible after

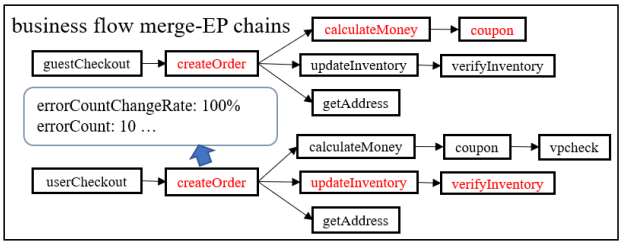

Figure 7: EP Chain Comparison Example

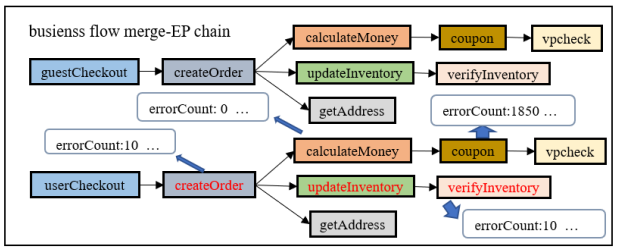

Figure 8: EP Chain Filtering Example

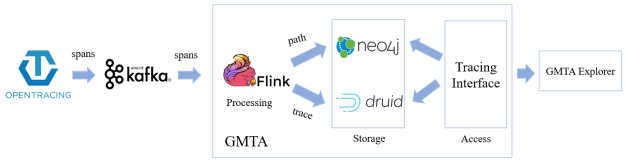

Figure 9: GMTA in eBay

an online issue, but diagnosing a production problem at runtime for the large-scale service ecosystem is troublesome. The diagnosis requires not only domain and site reliability knowledge, but also automated observability support. There are many tools such as metric monitoring and alarming, basic trace visualization, and log visualization tools available for use. However, due to the complexity of the system and the uncertainty of the problem, SREs still cannot quickly diagnose problems in many cases. To help SREs to diagnose various problems, GMTA Explorer supports the following two use cases of problem diagnosis:

U4. as SREs, to reduce the root cause scope of a production problem by comparing the business flows before and after the problem occurs and analyzing the EP chains.

U5. as SREs, to retrieve EP chains and visualize them for given operation(s)/service(s) based on observations such as service/application alerts.

For large-scale microservice systems, many "soft" EP chains exist. A "soft" EP chain passes error messages, but does not impact the availability or performance of the business. When SREs try to find the EP chains that actually cause a production issue, the SREs are often swamped with "soft" error chains. Therefore, "soft" EP chains heavily interfere with problem diagnosis and thus become a challenge for trace analysis.

For U4, GMTA Explorer provides two views for the comparison of paths and business flows. The first view is similar to what is used for U3, as shown in Figure 6. With this comparison view, SREs can carefully choose the time ranges of comparison and examine the differences of the same path or business flow before and after a runtime issue is raised. Based on the comparison, SREs may quickly narrow down the scope of problem diagnosis to a single path and a few operations. The other view provided by GMTA Explorer is the visualization of new EP chains that occur with the reported runtime issue. As shown in Figure 7, GMTA Explorer highlights the EP chains with a significant change: if the count of an EP chain increases by more than 50% and accounts for more than 5% of the traffic of the path, this EP chain is highly likely to be associated with the issue. Then SREs can examine the logs of the highlighted EP chain to further locate the root cause of the issue.

U5 is related to alerts generated for specific services and operations. The root cause of an alert is often not the service where the alter is raised. Therefore, SREs need to identify and examine many dependent services to locate the root cause. For example, a regular service in eBay may have up to 100 dependent services. Therefore, this process of problem diagnosis can be time-consuming and challenging. For U5, GMTA Explorer can retrieve and visualize the EP chains that pass the operation that raises an alert within the specified paths and business flows in near real-time. From the examined EP chains, SREs can identify the root causes more easily.

Figure 8 shows a real case of the EP chain visualization. "Create Order" is first alerted with an error spike alert. The view highlights an EP chain: "verifyInventory → updateInventory → create Order". Although the EP chain of "coupon → calculateMoney" appears 1850 times for retrying, it is filtered out since the operation "caclulateMoney" is not impacted.

GMTA Explorer provides the features of business flow visualization and comparison along with EP chain comparison and EP chain filtering. These features can help users understand service dependencies and business flow changes, narrow down the scope, and improve the efficiency of problem diagnosis.

5 EXPERIMENTAL STUDY

GMTA has been implemented and used in eBay. We conduct an experimental study based on the trace data produced by eBay. The objective of the study is to evaluate the effectiveness and efficiency of GMTA for the requirements of architecture understanding and problem diagnosis.

The microservice system under study in eBay includes around 3,000 services and more than 100,000 operations. These services work together to serve more than 10 business domains and form thousands of paths. In eBay, the microservice system processes around 26 billion spans per day. eBay has established a distributed tracing system with 1% sampling rate. The sampled trace data is pushed to a distributed messaging system implemented with Kafka [3]. Based on the distributed tracing infrastructure, we design GMTA as a pipeline shown in Figure 9. In the current version, the module of trace data processing is implemented based on Flink [2] (a stream processing framework), and the graph database and analytics database in the storage module are implemented with Neo4j [7] and Apache Druid [1], respectively.

In the study, we compare GMTA with two traditional trace processing approaches, i.e., OTD-R and ATD-R, qualitatively and quantitatively. OTD-R stores the original trace data (i.e., spans, traces, and their properties) in a relational analytics database. ATD-R aggregates the traces into paths but still stores the trace and path data in a relational analytics database. OTD-R is often used in small-scale

Table 2: Approaches under Comparison

Approach	Traces	Paths	Business Flow	Graph-based Storage
OTD-R	√	×	×	×
ATD-R	√	√	×	×
GMTA	√	√	√	√

Table 3: Performance Study Results (ms)

Use Case		OTD-R		ATD-R		GMTA	
		cold	hot	cold	hot	cold	hot
S1: Single Trace Query		2130	501	2250	486	2348	545
S2: Single Operation Query		1405	402	482	305	462	283
S3: Single Path Query		timeout	timeout	180	75	196	64
S4: Error Propagation Chain Query		timeout	timeout	551	225	648	43
S5: Business Flow Generation	simple	timeout	timeout	2307	909	1477	117
	complex	timeout	timeout	8829	2385	4036	1144
S6: Service Dependency Analysis		timeout	timeout	timeout	timeout	200	15

microservice systems, while ATD-R was previously used in eBay. The characteristics of the three approaches are shown in Table 2. OTD-R does not support trace aggregation, while ATD-R supports trace aggregation but without graph-based business flow grouping. GMTA supports both of trace aggregation and business flow grouping, and implements graph-based storage. In the study, both OTD-R and ATD-R are implemented based on Apache Druid [1].

Microservice architecture understanding and problem diagnosis involve trace analysis requirements at three levels, i.e., trace, path, business flow. We derive the following six scenarios from the practices in eBay and use these scenarios to evaluate the effectiveness and efficiency of the three approaches.

- **S1: Single Trace Query**. Collect the spans and related metrics (e.g., latency) of a given trace.
- **S2: Single Operation Query**. Collect the metrics (e.g., error rate, traffic, latency) of a given operation of a service.
- **S3: Single Path Query**. Collect the hops and related metrics (e.g., latency) of a given path.
- **S4: Error Propagation Chain Query**. Collect the EP chains that pass more than three services.
- **S5: Business Flow Generation**. Construct and return a business flow that passes the given operation(s), along with related metrics (e.g., error rate, traffic, latency) at the operation level.
- **S6: Service Dependency Analysis**. Return all the services that directly or indirectly invoke a given operation in the same trace.

Based on these scenarios, we conduct a series of experiments in the pre-production environment of eBay.

The first experiment is to evaluate the processing performance of GMTA. We take a part of eBay's trace data as input and use a server with 4 CPU cores and 8GB of memory as the running environment. We gradually increase the data load of GMTA. When GMTA processes 24K spans per second, the server's CPU usage is close to 100%.

In the end, we deploy the GMTA processing module as a Flink [2] job, and run 20 parallel instances to handle eBay's trace data. The configuration of each instance is 4 CPU cores and 8GB of memory. Our initial empirical investigation shows that that 20 instances can process trace data from eBay's production environment in a timely fashion.

The second experiment is to evaluate query performance of GMTA. eBay generates around 300K spans per second. In this experiment, we process and store trace data for 7 days. After sampling, around 2 billion spans and 100 million traces are stored in the database of GMTA. As Apache Druid [1] and Neo4j [7] used by GMTA provide hot data cache, and repeated queries take much less time than the first query, we record the performance of cold query and hot query separately. In the experimental results described below, "cold" indicates the time cost for the first-time query and "hot" indicates the time cost for the subsequent queries. The overall results for the six scenarios are shown in Table 3, where the unit is millisecond and the timeout threshold is 5 minutes.

For S1, we query all spans of a trace by trace ID. The performance of the three approaches is very close, because they all query the trace from Apache Druid [1]. The time cost of GMTA and ATD-R is slightly longer than that of OTD-R, because they involve additional fields for the aggregation of spans.

For S2, we query the metrics of an operation within an hour. The time cost of ATD-R and GMTA is 66% (cold) and 24% (hot) less than that of OTD-R.

For S3 and S4, GMTA and ATD-R have their own advantages and disadvantages in the performance of hot/cold query, but OTD-R fails to return results within 5 minutes. As can be seen, aggregation and analysis can improve query efficiency in certain scenarios, and support more types of queries.

For S5, we study two different types of business flow generated by either a single given operation (denoted as *simple*) or three operations (denoted as *complex*). The query time range of the former is 1 hour, and the latter is 5 minutes. GMTA outperforms ATD-R 36.0% (cold) and 87.1% (hot) in *simple*; and 54.2% (cold) and 52.0% (hot) in *complex*. Thus, the query performance of GMTA is better than that of ATD-R.

For S6, only GMTA can execute this kind of query. The result shows that the graph-based GMTA can handle queries with higher complexity than the non-graph-based ATD-R.

The experimental results show that GMTA provides efficient query support through aggregation, analysis, and graph-based storage. Based on the effective and efficient query interfaces provided by GMTA, we can build applications to solve different practical problems.

6 CASE STUDY

In order to assess the effectiveness of GMTA, we demonstrate and validate GMTA with the monitoring team and the SRE team of eBay. After three training sessions are conducted at the beginning, GMTA Explorer is used and validated by the teams during their day-to-day work. There are 20 team members in total who participate in the validation, including 7 developers, 10 SREs, and 3 architects. A month later, we interview the team members and collect cases presented in this section. In particular, the team members participate in three major tasks:

1. 15 people use GMTA Explorer to understand architecture, including 2 architects, 7 developers, and 6 SREs.
2. 12 people use GMTA Explorer to understand changes, including 6 developers and 6 SREs.

3. 11 people use GMTA Explorer to improve the efficiency of problem diagnosis, including 2 developers and 9 SREs.

During the interview process, we discover and collect feedback on GMTA Explorer with three major findings: (1) 13 out of 15 people who use GMTA Explorer agree that it could provide more information to help them quickly understand the architecture and business processes compared to a general trace visualization system; (2) 8 out of 12 people believe that GMTA Explorer could help them understand business changes; (3) 9 out of 11 people believe that the EP chains in GMTA Explorer could indeed provide a reference for problem diagnosis.

Based on the interview results, we next present some real cases supported by GMTA. The cases are divided into two parts: the first part includes cases in which users use GMTA Explorer to understand architecture, and the second part includes cases in which users use GMTA Explorer to improve the efficiency of problem diagnosis. We remove some sensitive information such as service names and performance metrics in the cases due to eBay's confidentiality requirement.

6.1 Architecture Understanding Cases

In eBay, there are over 3,000 services, each of which is developed by an individual or a team. There are only a few people who can fully understand their business processes and underlying architecture. Even senior domain architects are mostly experienced with their own domain; additionally, they do not know the details of every business change. The following two real cases show how GMTA Explorer helps users understand dependencies and changes.

Case 1: Developer Dev_A receives a change request to modify the operation logic of a service, but Dev_A did not participate in the development of this service before, so she does not understand the logic of the service. The main developer of the service has left the company, and there is no relevant documentation. Therefore, it is difficult for Dev_A to assess the risk and impact of the change. If she wants to implement the change requirement, she has to look at a lot of code to understand the scope of the change. However, after inputting "serviceName" and "operationName" to GMTA Explorer, Dev_A can intuitively see the upstream and downstream services of the target operation, and confirm the scope of change impact to modify the logic of the service. Using the views provided by GMTA Explorer, Dev_A can see the operation-related information such as traffic, latency, and error rate, so she can quickly establish a clear understanding of the operation. By one more click, Dev_A can get the paths and business flows containing the operation. GMTA Explorer shows that a total of 5 services depend on the operation to be modified, and Dev_A finally decides to postpone the change due to potential risks.

Case 2: A service that developer Dev_B is responsible for depends on a service from team C, so every time team C releases a change, Dev_B needs to immediately figure out this specific change to determine whether the change will affect the service that she is responsible for. However, the service from team C is relatively new, so service releases are very frequent, and team C does not provide a change specification document. Dev_B often needs to communicate with members of team C to obtain detailed information, which brings very high communication costs. Now Dev_B can use GMTA Explorer to check the business flow of team C at any time, and understand the details of business changes by Dev_B through the comparison of different time periods. Figure 10 shows an example from two different input time ranges. The presence of dark red and dark green squares in the red box indicates that the business flow of team C has changed: the traffic is switched from red operations to green operations. Dev_B clicks on the corresponding operations to get more detailed indicators about this change, and confirms that this change will not affect the service that Dev_B is responsible for.

In the preceding case, Dev_B quickly notices and understands the change through GMTA Explorer, and avoids the communication overhead with team C.

6.2 Problem Diagnosis Cases

Root cause analysis and incident recovery are critical for business. During an incident, metrics, alerts, traces, logs, and events are collected and used for incident troubleshooting. However, this information is overwhelming for a large-scale system. The following three cases show how users use GMTA Explorer to speed up problem diagnosis.

Case 3: As shown in Figure 11, the incident under diagnosis is observed by a "markdown" alert from "ServiceA" to "ServiceB", indicating that "ServiceA" fails to call "ServiceB" or the invocation latency is too high. SRE_D needs to locate the root cause as soon as possible. However, because there are too many types of different traces, SRE_D checks several traces and still cannot obtain actionable insights related to the incident.

SRE_D runs GMTA Explorer on her laptop, so she switches to GMTA Explorer and inputs the two services' names. GMTA Explorer shows dependencies between different operations with their key performance metrics (e.g., error rate, latency, and traffic). SRE_D could quickly focus on the scope of the root cause. Then she discovers that the metrics of operations c and d of "ServiceC" are abnormal. Finally SRE_D discovers the root cause of the incident: the new code deployment for "ServiceC". Through the query interface and UI provided by GMTA Explorer, SRE_D resolves the incident within a few minutes.

Case 4: SRE_E receives an alert from the monitoring system: the number of errors on the "createLabel" operation of the "label" service increases significantly. SRE_E cannot determine whether the errors are caused by "createLabel" itself or caused by downstream operations. SRE_E queries the EP chains of the "createLabel" operation. The results displayed by the EP chains show that all EP chains start from the operation itself. SRE_E judges that the root cause of the alert is in "createLabel", and then resolves the incident through further investigation on the raw logs.

Case 5: SRE_F receives an alert from the business metrics: the success rate of the payment process drops significantly. In this case, there are many soft error chains, so the overall errors do not spike and the machine learning models for the error spike has not detected any anomaly. SRE_F struggles to find the root cause so she tries the EP chains. SRE_F selects the time period that the business failure occurs as from 2:34 pm to 2:35 pm on March 4, 2020, an hour before the incident. GMTA Explorer returns four

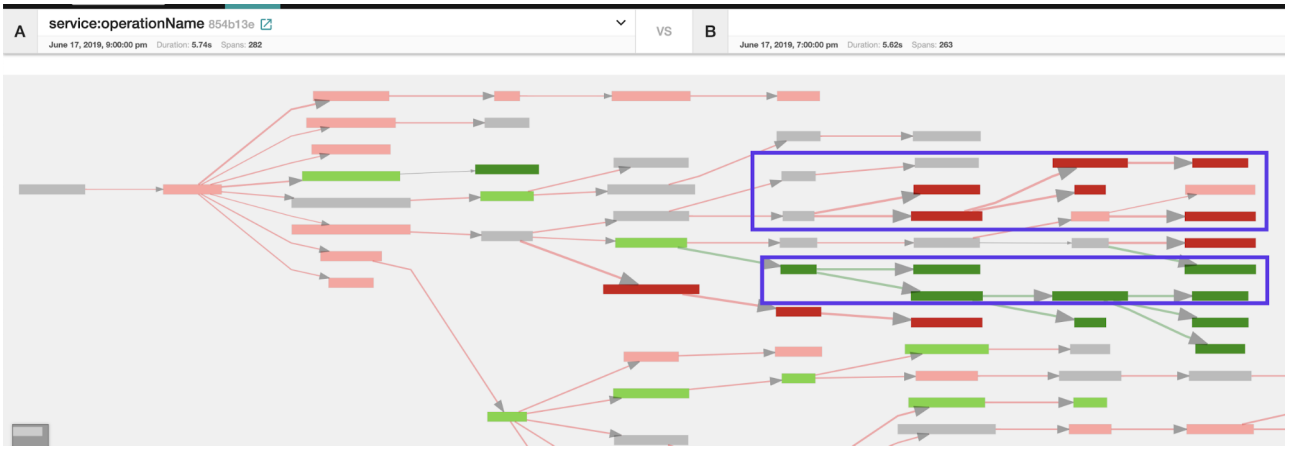

Figure 10: Path Change Example

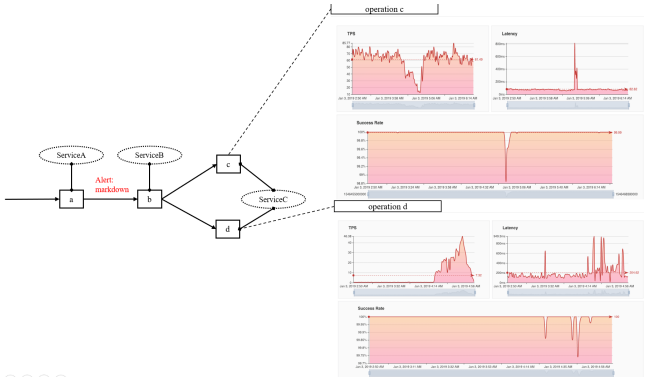

Figure 11: Problem Diagnosis Example

new EP chains that are newly added or increased in number during the business failure. SRE_F directly identifies two of the four EP chains for further investigation, and finally resolves the incident within 4 minutes.

The cases in this section show that GMTA Explorer can help developers, SREs, and architects to understand runtime architecture and diagnose problems of large-scale industrial microservice systems. Users with different roles may focus on different functionalities of GMTA Explorer, but all of these users agree that GMTA Explorer is effective and helpful for their purposes.

7 RELATED WORK

Distributed tracing systems such as Zipkin [8], Jaeger [6], HTrace [5], and Pinpoint [10] have been widely used in industrial microservice systems. These distributed tracing systems provide a reusable framework to generate and collect trace data for distributed systems. Most of these distributed tracing systems are designed to be generic and simple to use for microservice systems of mid to small size. For example, Zipkin [8] and Jaeger [6] directly store the original trace information, and provide visualization for a single trace. As the number of services increases, the number of traces grows accordingly. It is difficult for these systems to provide a clear global view for monitoring a business scenario. Developers have to query a specific trace to make decisions based on their own business knowledge. There is no additional support to extract insights such as extracting critical flow via flow-based monitoring for large-scale services.

Distributed tracing is often used for problem diagnosis [11–13, 16, 17, 21, 22] and architecture understanding [9, 14, 18]. For example, Canopy [14] aggregates trace data and introduces feature extraction as a step in aggregate analysis, but the analysis of the tracing data stays at the visualization level. Chen et al. [11] propose a path-based trace aggregation scheme. Neither work supports the discovery of EP chains or higher-level abstraction to support flow-based analysis or user-case monitoring. Sambasivan et al. [17] compare request flows to diagnose performance changes but do not conduct comparison at the aggregate levels. Hence their work cannot provide users with useful information such as the error rate and traffic. In contrast, GMTA Explorer compares aggregated data to display additional useful information.

Based on our experiences, the preceding previous work faces different difficulties to be applied in eBay's production environments due to performance challenges or our advanced use cases. In eBay, there were a few previous efforts [19, 20] based on graphs, but the graphs are not generated from the distributed tracing data. Therefore, we build from scratch a new system to overcome the challenges and to be adopted for architecture understanding and problem diagnosis.

8 CONCLUSION

In this paper, we have proposed and designed a graph-based approach of microservice trace analysis, named GMTA, for architecture understanding and problem diagnosis. Built on a graph-based representation, GMTA includes efficient processing of traces produced on the fly and an efficient storage and access mechanism by combining a graph database and a real-time analytics database. Extended from GMTA, we have implemented GMTA Explorer and applied it in eBay. The results from our experimental study and case study have confirmed its effectiveness and efficiency in supporting architecture understanding and problem diagnosis in industrial-scale microservice systems.

REFERENCES

[1] [n.d.]. Apache Druid. https://druid.apache.org/. Accessed: 2020-05-19.
[2] [n.d.]. Apache Flink. https://flink.apache.org/. Accessed: 2020-05-19.
[3] [n.d.]. Apache Kafka. https://kafka.apache.org/. Accessed: 2020-05-19.
[4] [n.d.]. ClickHouse. https://clickhouse.tech/. Accessed: 2020-05-19.
[5] [n.d.]. HTrace. http://htrace.org/. Accessed: 2020-05-19.
[6] [n.d.]. Jaeger. https://www.jaegertracing.io/. Accessed: 2020-05-19.
[7] [n.d.]. Neo4j. https://neo4j.com/. Accessed: 2020-05-19.
[8] [n.d.]. Zipkin. https://zipkin.io/. Accessed: 2020-05-19.
[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 259–272. http://www.usenix.org/events/osdi04/tech/barham.html
[10] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS 2003), Lihue (Kauai), Hawaii, USA, May 18-21, 2003*, Michael B. Jones (Ed.). USENIX, 85–90. https://www.usenix.org/conference/hotos-ix/magpie-online-modelling-and-performance-aware-systems
[11] Mike Y. Chen, Anthony J. Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. 2004. Path-Based Failure and Evolution Management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), San Francisco, California, USA, March 29-31, 2004*, Robert Tappan Morris and Stefan Savage (Eds.). USENIX, 309–322. http://www.usenix.org/events/nsdi04/tech/chen.html
[12] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014), Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow
[13] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 2010. Experiences with Tracing Causality in Networked Services. In *Proceedings of 2010 Internet Network Management Workshop / Workshop on Research on Enterprise Networking, San Jose, CA, USA, April, 2010*, Aditya Akella, Nick Feamster, and Sanjay G. Rao (Eds.). USENIX Association. https://www.usenix.org/conference/inmwren-10/experiences-tracing-causality-networked-services
[14] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), Shanghai, China, October 28-31, 2017*. ACM, 34–50. https://doi.org/10.1145/3132747.3132749
[15] James Lewis and Martin Fowler. 2014. Microservices: a Definition of This New Architectural Term. Retrieved May 20, 2020 from http://martinfowler.com/articles/microservices.html
[16] Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), San Jose, California, USA, May 8-10, 2007*, Larry L. Peterson and Timothy Roscoe (Eds.). USENIX. http://www.usenix.org/events/nsdi06/tech/reynolds.html
[17] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011), Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/diagnosing-performance-changes-comparing-request-flows
[18] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf
[19] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Kopru, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. 2019. GRANO: Interactive Graph-based Root Cause Analysis for Cloud-native Distributed Data Platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1942–1945.
[20] Hanzhang Wang, Chirag Shah, Praseeda Sathaye, Amit Nahata, and Sanjeev Katariya. 2019. Service Application Knowledge Graph and Dependency System. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 134–136.
[21] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* (2018), 1–1.
[22] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019), Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 683–694. https://doi.org/10.1145/3338906.3338961