Unveiling Graph Structures in Microservices: Service Dependency Graph, Call Graph, and Causal Graph

Abhishek Tiwari

Citation: *A. Tiwari*, "Unveiling Graph Structures in Microservices: Service Dependency Graph, Call Graph, and Causal Graph", Abhishek Tiwari, 2024.
doi:10.59350/hkjz0-7fb09

Published on: December 21, 2024

The rise of service-oriented architecture (SOA) and microservices architecture has led to a major shift in software development, enabling the creation of complex, distributed systems composed of independent, loosely coupled services. These architectures offer numerous benefits, including scalability, flexibility, and resilience. However, the distributed nature of these systems introduces new challenges related to understanding, managing, and analysing their behaviour. Graph-based modelling has emerged as a powerful technique for addressing these challenges. Graphs provide a natural and intuitive way to represent the relationships between different components of an SOA or microservices system, allowing developers and operators to gain insights into the system's structure, dependencies, behaviour, and performance.

This article examines three graph models: service dependency graphs, call graphs, and causal graphs. For each graph type, the article provides a detailed exploration of its mathematical foundations and use cases applications.

Graphs and Adjacency Metrics

Graphs are fundamental mathematical structures used to model relationships and interactions in various systems, including social networks, communication systems, and biological interactions. A graph provides a flexible and intuitive framework to represent entities (nodes) and their relationships (edges), enabling efficient analysis of both structural and functional properties.

Graph Basics

A graph G is formally defined as:

$$G = (V, E),$$

Where,

- V: A set of vertices (or nodes) representing entities.
- E: A set of edges, where each edge $e \in E$ is a pair (u, v) with $u, v \in V$. An edge represents a relationship or interaction between the vertices u and v.

Graphs can be classified based on their structure and properties:

1. **Directed Graphs** ($G = (V, E)$): Edges have a direction, meaning $(u, v) \neq (v, u)$. Directed graphs are suitable for modeling asymmetric relationships, such as service dependencies or communication flows.

2. **Undirected Graphs**: Edges are bidirectional, where $(u, v) = (v, u)$. These are used for symmetric relationships, like mutual friendships.
3. **Weighted Graphs**: Each edge (u, v) has an associated weight $w(u, v)$, representing the strength, cost, or frequency of the relationship.

Adjacency Matrix

The adjacency matrix is a mathematical representation of a graph, providing a compact way to encode the relationships between nodes. For a graph $G = (V, E)$ with n nodes, the adjacency matrix A is an $n \times n$ matrix defined as:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from node } i \text{ to node } j, \\ \\ 0 & \text{otherwise.} \end{cases}$$

For weighted graphs, the adjacency matrix is generalized to include weights:

$$A[i][j] = \begin{cases} w(i, j) & \text{if there is an edge from node } i \text{ to node } j, \\ \\ 0 & \text{otherwise.} \end{cases}$$

For example: Consider the directed graph G with $V = \{A, B, C\}$ and $E = \{(A, B), (B, C), (A, C)\}$:

$$A = \begin{bmatrix} 0 & 1 & 1 \\ \\ 0 & 0 & 1 \\ \\ 0 & 0 & 0 \end{bmatrix}$$

- $A[1][2] = 1$: Indicates an edge from A to B.
- $A[2][3] = 1$: Indicates an edge from B to C.

The adjacency matrix is a foundational tool in graph theory, offering a structured way to represent and analyze the relationships in a graph. Its compact representation and mathematical properties make it an indispensable instrument for studying graph-based models for SOA.

Service Dependency Graphs

Dependency graphs serve as a visual representation of the intricate web of interdependencies between services within an SOA or microservices architecture. These graphs, often referred to as service dependency graphs, are directed graphs in which nodes symbolise services or microservices, and edges represent dependencies between them, typically in the form of service invocations.

Definition of Service Dependency Graph

An Service Dependency Graph (SDG) is a directed graph $G = (V, E)$,

Where,

- V: The set of vertices, where each vertex $v_i \in V$ represents a service.
- E: The set of directed edges, where each edge $(v_i, v_j) \in E$ indicates that service v_i depends on service v_j.

For example: A directed edge $(S1, S2)$ signifies that Service $S1$ requires Service $S2$ to perform its operations. Following illustrated different archetypes in a service dependency graph.

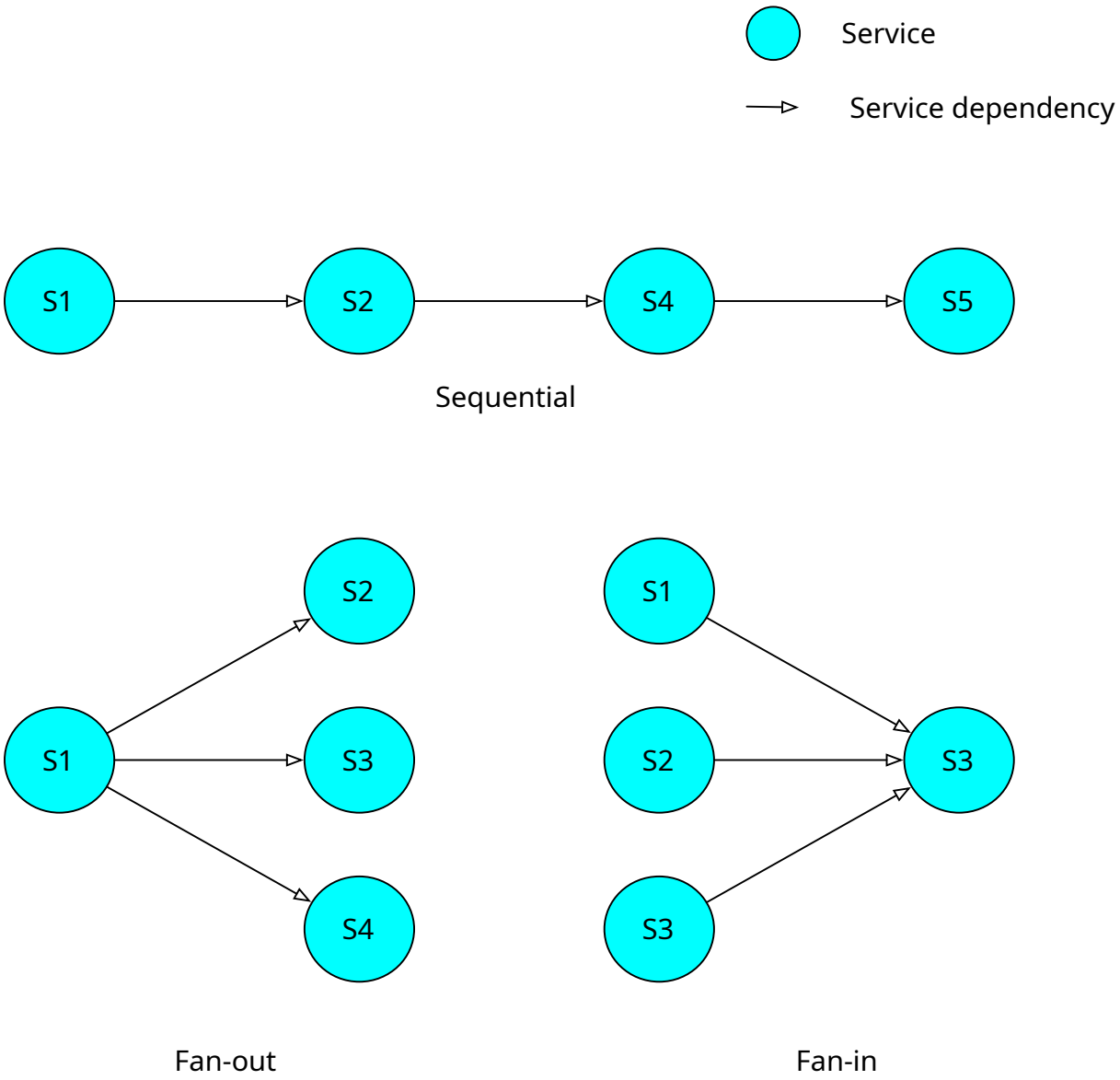

Figure 1: Service Dependency Graph and example archetypes. For a sequential archetype, Service S1 depends on Service S2 to perform its operation. Likewise Service S2 depends on Service S3 and S4 to perform its job hence S1 has a transitive dependency on S3 and S4. Similar service dependencies can be seen in fan-out and fan-in archetypes.

Adjacency Matrix

An example of adjacency matrix A for the SDG with services $S1, S2, S3, S4$ in sequential pattern can be defined as:

$$A = \begin{bmatrix} \emptyset & 1 & \emptyset & \emptyset \\ \emptyset & \emptyset & 1 & \emptyset \\ \emptyset & \emptyset & \emptyset & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

Explanation:

- Row i and column j correspond to services S_i and S_j, respectively.
- $A[i][j]$ represents dependency between S_i and S_j.
- $\emptyset$ indicates no dependency exist between the corresponding services.

Use Cases & Applications

SDGs offer a range of applications in various stages of the software development lifecycle, including:

Visualisation of Service Dependencies SDGs provide a clear and intuitive way to visualise the complex relationships between services in a system. This visualisation aids in understanding the overall system architecture, identifying critical services, and pinpointing potential bottlenecks.

System Complexity Analysis By analysing the structure of an SDG, developers can assess the complexity of the system, identify areas of high coupling, and evaluate the impact of changes on different services. Metrics such as node degree, graph diameter, and clustering coefficient can be used to quantify system complexity and guide design decisions.

Microservice Testing SDGs play a crucial role in testing microservice systems by enabling developers to identify the dependencies of a particular service under test and design comprehensive test cases that cover all potential interaction scenarios (see [1]). Tools can automatically generate and execute tests based on the SDG, improving test coverage and efficiency.

Monolithic Application Decomposition During the process of migrating a monolithic application to a microservice architecture, SDGs assist in identifying logical boundaries and dependencies between different components, facilitating the decomposition into smaller, independent services. This decomposition process can be automated using clustering algorithms and graph partitioning techniques applied to the SDG.

Design Error and Anti-pattern Identification SDGs can be used to detect design flaws and anti-patterns, such as cyclic dependencies, long chains of dependencies, and hub-and-spoke structures, that can negatively impact system maintainability, scalability, and resilience. Tools can automatically analyse SDGs and flag potential issues, providing developers with valuable insights for improving the system design.

Attack modelling and response strategies SDGs are used in attack modelling and analysis to visualise and quantify the impact of attacks, and to inform response strategies. They capture functional dependencies, showing how the availability, confidentiality, and integrity of one service can impact others. SDGs are often combined with attack graphs (AGs) to provide a more comprehensive view of the attack landscape. AGs model the steps an attacker might take to exploit vulnerabilities and reach a specific goal, while SDGs capture the service dependencies that can be leveraged for attack propagation. Combining these two graphs allows for a more accurate assessment of the attack impact and informs the selection of appropriate responses

Service Call Graphs

A service call graph (SCG) or call graph is extension of service dependency graph where edges are labelled with specific operation or endpoint name. Call graphs capture the dynamic flow of execution within a microservices system by meticulously tracing the sequence of service invocations. SCG covers all possible invocation or call paths in a given microservices topology. SCG is often used as permission graph. Call graphs provide a valuable runtime perspective of the system's behaviour, enabling developers and operators to understand how services interact during actual execution.

Definition of Call Graphs

A **Call Graph** is a directed, labeled, and possibly weighted graph $G = (V, E, L)$.

Where,

- V: The set of vertices, each representing a service or service instance.
- $E \subseteq V \times V$: The set of directed edges, where an edge (u, v) indicates that service u makes a call to service v.
- L: A labeling function $L : E \to \Sigma$, where Σ is the set of endpoint names or operations. For example, $L((u, v)) =$ operation specifies the name of the operation invoked by u on v.
- **Weights (optional)**: A weight $w(u, v)$ can be associated with each edge to represent runtime metrics such as latency, throughput, or call frequency.

Adjacency Matrix

For a Call Graph $G = (V, E, L)$, the adjacency matrix A is a matrix of dimensions $|V| \times |V|$, where each entry $A[i][j]$ contains a set of labels or metrics associated with the call from node i to node j.

$$A[i][j] = \begin{cases} \{l_1, l_2, \ldots, l_k\} & \text{if there are operations } \{l_1, l_2, \ldots, l_k\} \text{ invoked from node } i \text{ to node } j, \\ \emptyset & \text{if no call exists from node } i \text{ to node } j. \end{cases}$$

For instance:

- $A[i][j] = \{\text{GET /users}, \text{POST /orders}\}$ indicates that service i makes two calls to service j using these operations.
- $A[i][j] = \emptyset$ means there are no calls from i to j.

Example: The adjacency matrix A for the SCG with services $S1, S2, S3, S4$ is defined as:

$$A = \begin{bmatrix} \emptyset & \{\text{putX}, \text{getX}, \text{deleteX}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\text{putY}, \text{getY}, \text{deleteY}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\text{putZ}, \text{getZ}, \text{deleteZ}\} \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

Explanation:

- Row i and column j correspond to services S_i and S_j, respectively.
- $A[i][j]$ represents the set of operations that S_i invokes on S_j.
- $\emptyset$ indicates no direct calls exist between the corresponding services.

Service-to-Service Mappings:

1. $S1 \rightarrow S2$: $\{\text{putX}, \text{getX}, \text{deleteX}\}$
2. $S2 \rightarrow S3$: $\{\text{putY}, \text{getY}, \text{deleteY}\}$
3. $S3 \rightarrow S4$: $\{\text{putZ}, \text{getZ}, \text{deleteZ}\}$

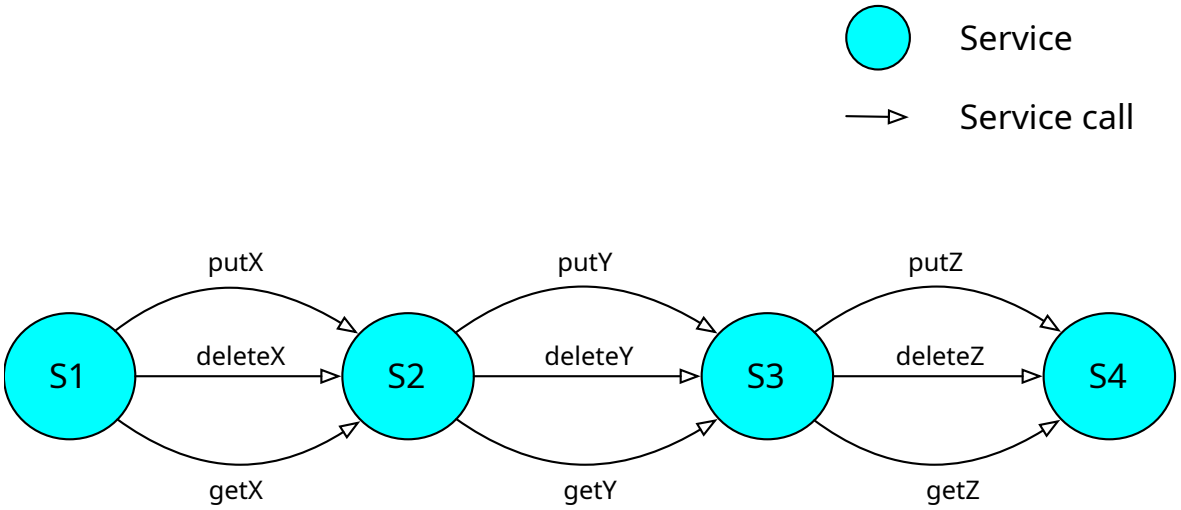

Figure 2: Service Call Graph

Weighted Adjacency Matrix

If the SCG includes runtime metrics, the adjacency matrix can store weighted values for each operation. The matrix W is defined as:

$$W[i][j](l) = \text{metric}(i, j, l),$$

Where l is a specific label (operation or endpoint), and metric(i, j, l) represents the runtime value associated with the call l from i to j.

In addition metric(i, j, l) could be:

- Call frequency: metric$(i, j, l) =$ Number of calls per unit time,
- Latency: metric$(i, j, l) =$ Average time taken for the call,
- Error rate: metric$(i, j, l) =$ Percentage of failed calls.

For example: A weighted adjacency matrix SCG with latency (in ms) of calls,

$$A = \begin{bmatrix} \emptyset & \{\text{(putX,30)}, \text{(getX, 10)}, \text{(deleteX, 30)}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\text{(putY, 15)}, \text{(getY, 10)}, \text{(deleteY, 30)}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\text{(putZ, 35)}, \text{(getZ, 15)}, \text{(deleteZ} \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

Service invocation chain

A service call chain (SCC) or service invocation chain (SIC) indicates a sequence of service invocations between multiple microservices to fulfil a request or complete an operation. Invocations can be synchronous (blocking) or asynchronous (non-blocking), depending on the system design and requirements. For a given call graph $G = (V, E, L)$, there may exist multiple call chains between a source node and a destination node. These chains represent alternate paths through the graph. In addition call chains may vary depending on runtime conditions, such as failovers, or feature toggles.

A SCC is a sequence $P = [(v_1, l_1), (v_2, l_2), \ldots, (v_k, l_k)]$ such that:

$$(v_i, v_{i+1}) \in E \quad \text{and} \quad L(v_i, v_{i+1}) = l_i \quad \forall i \in \{1, 2, \ldots, k-1\}.$$

Depth or length of call chain is described by k or number of elements in sequence.

A service call chain P can be constructed by traversing the adjacency matrix A, ensuring that both the vertices v_i and the labels l_i meet the specified criteria.

As illustrated below, three call chains in a topology of 4 services. One of the call from $S1$ to $S4$ could be represented by P:

$$P = [(S1, \text{putX}), (S2, \text{putY}), (S3, \text{putZ})].$$

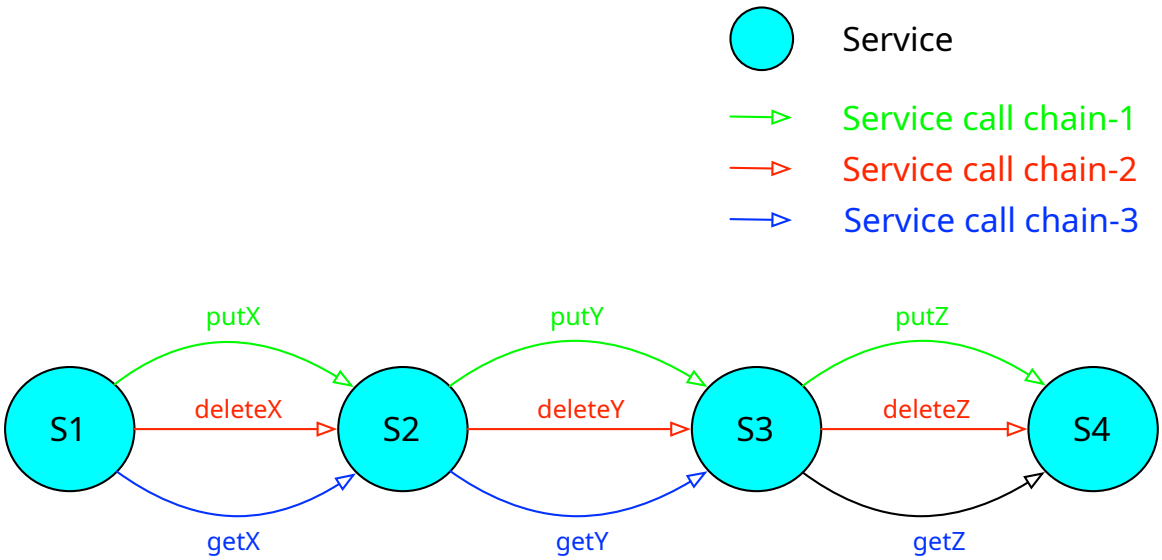

Figure 3: Service Call Chain. 3 call chains in a topology of 4 services.

Error and latency If an error occurs in one of the service calls in the chain, it can propagate back up the chain and affect the overall request processing.

Similarly, latency of a call chain is defined as the total time required to traverse the chain. This is calculated as the sum of the latencies of each invocation in the chain.

$$\text{Latency}(P) = \sum_{i=1}^{k-1} \text{Latency}(v_i, l_i).$$

Consider the following Service Call Chain:

$$P = [(S1, \text{putX}), (S2, \text{putY}), (S3, \text{putZ})],$$

with the following latencies:

- $\text{Latency}(S1, \text{putX}) = 30$ ms
- $\text{Latency}(S2, \text{putY}) = 15$ ms
- $\text{Latency}(S3, \text{putZ}) = 35$ ms

The total latency of the chain P is:

$$\text{Latency}(P) = 30 + 15 + 35 = 80 \text{ ms}.$$

Use Cases & Applications

Call graphs facilitate a variety of applications in understanding and managing microservice systems:

Understanding Runtime Behaviour Call graphs offer a visual representation of the execution flow within a microservice system, allowing developers and operators to understand the sequence of service interactions during runtime. This understanding is crucial for debugging and troubleshooting issues, especially in distributed systems where the flow of execution can be complex and difficult to follow.

Performance Profiling and Bottleneck Analysis By analysing call graphs, developers can identify performance bottlenecks by pinpointing services that experience high invocation frequencies or long execution times. This information is crucial for optimizing resource allocation and improving the overall system performance (see [2]).

Debugging and Troubleshooting Distributed Systems In distributed systems, tracing the flow of execution across multiple services can be challenging. Call graphs help developers to visualise the interactions between services, pinpoint the origin of errors, and identify the root cause of failures.

Designing caching strategy for microservices Call graphs are important to design and implement the caching strategy for microservices in particular for providing automatic and coherent caching for microservices call graphs (see [3]).

Causal Graphs

Causal graphs aim to capture the cause-and-effect relationships between different metrics or events in a microservice system. These graphs are used to understand the root causes of performance issues, failures, and other anomalies in complex distributed systems. In a causal graph, nodes represent metrics or events, while directed edges denote causal relationships. An edge from node A to node B indicates that A causes B. The acyclic nature of the graph ensures that there are no circular dependencies.

Definition of Causal Graphs

Let $G(V, E)$ be the causal graph where:

1. Nodes (V) represent metrics from all services. For n services ($S_1, S_2, \dots, S_n$), the nodes are:

$$V = \{W_{S1}, U_{c,S1}, U_{m,S1}, L_{S1}, E_{S1}, \dots, W_{Sn}, U_{c,Sn}, U_{m,Sn}, L_{Sn}, E_{Sn}\}$$

$W_{Si}, U_{c,Si}, U_{m,Si}, L_{Si}, E_{Si}$ are metrics described in following section.

2. Edges (E) represent causal relationships among metrics. There can be two types of edges in causal graphs,

- Intra-service edges: Represent causal relationships among metrics within a single service (e.g., $W_{S1} \rightarrow U_{c,S1}$).
- Inter-service edges: Represent causal relationships between metrics of different services (e.g., $L_{S2} \rightarrow L_{S1}$).

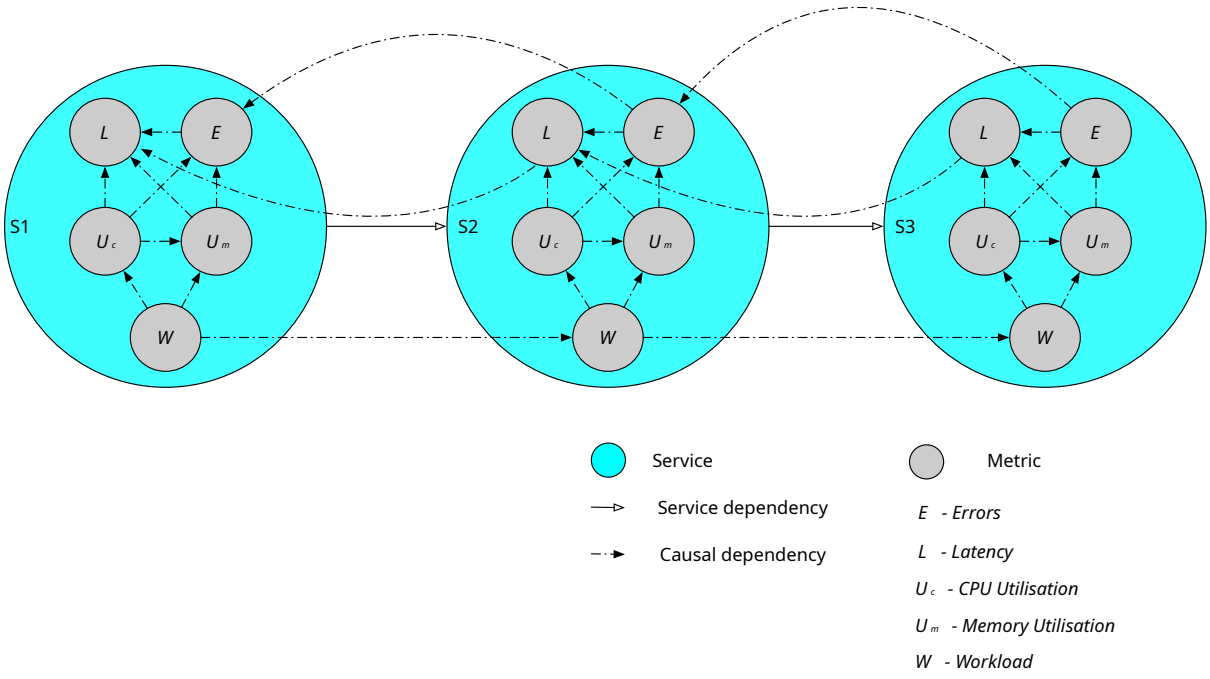

Figure 4: Causal graph of 3 services S1, S2, and S3 using golden metrics workload, CPU utilisation, memory utilisation, latency and errors.

Golden Signals

In microservices systems, the following key metrics (golden signals) are typically used for building causal graphs (see [4] and [5]),

1. **Workload (W)**: Represents the incoming requests or load on the system.

2. **CPU Utilization (U_c)**: Percentage of CPU resources utilized by the service.
3. **Memory Utilization (U_m)**: Percentage of memory resources consumed.
4. **Latency (L)**: The time taken to process a request.
5. **Errors (E)**: The rate of failed requests.

Example causal dependency between above metrics,

1. $W \rightarrow U_c$: Increased workload directly increases CPU utilization.
2. $W \rightarrow U_m$: Higher workload can lead to higher memory consumption.
3. $U_c \rightarrow L$: High CPU utilization can cause increased latency.
4. $U_m \rightarrow L$: High memory utilization might also contribute to latency.
5. $L \rightarrow E$: Increased latency can lead to higher error rates.

Use Cases & Applications

Causal graphs provide valuable insights into the behaviour of microservice systems, enabling a range of applications:

Root Cause Analysis Causal graphs are particularly valuable for root cause analysis of failures and performance issues in microservice systems. By tracing the causal relationships between different metrics and events, operators can identify the underlying factors contributing to an observed problem.

Performance Diagnosis and Optimization Causal graphs aid in identifying performance bottlenecks by highlighting metrics and events that have a significant impact on system performance. This information can be used to optimise resource allocation, fine-tune system configurations, and implement performance improvement measures.

Anomaly Detection and Alerting Causal graphs can be used to develop anomaly detection systems that can identify unusual patterns of behaviour and alert operators to potential issues before they impact users. By monitoring the causal relationships between metrics, the system can detect deviations from normal behaviour and trigger alerts.

Construction of Graphs

Construction of these graphs is a multi-step process that leverages runtime data, system architecture, and dependency tracking methods to accurately model the relationships among microservices in a system.

Static Analysis

Static analysis involves analysing software artefacts before deployment, typically by inspecting source code, configuration files, and documentation. Static analysis tools can identify service endpoints, dependencies between microservices based on direct calls, and data entity sharing. For instance, by parsing source code, one can determine which services make HTTP requests to other services and visualise these interactions as graph.

Static analysis has the advantage of being performed early in the development lifecycle, potentially identifying issues before deployment. It also provides a complete view of all possible execution paths;

However, static analysis can be limited by:

- Language dependency: Static analysis tools are often language-specific, making it challenging to analyse polyglot microservice systems. There are ongoing attempts to create Universal Language-Agnostic AST (LAAST) which may eventually overcome this gap.
- Difficulty in handling dynamic behaviour: Static analysis cannot fully capture dynamic aspects like load balancing or service discovery, which can affect runtime dependencies.
- Overhead: Performing comprehensive static analysis on large codebases can be computationally expensive and time-consuming.

Dynamic Analysis

Dynamic analysis relies on gathering information from a running system, typically through monitoring tools that collect telemetry data such as logs, traces, and metrics. This data can reveal the actual interactions between microservices in a production environment. By analysing runtime traces, one can determine the sequence of service calls, identify service dependencies, and quantify their frequency. Dynamic analysis can also capture asynchronous communication patterns like publish-subscribe, which static analysis might miss.

Dynamic analysis offers several benefits:

- Language agnosticism: As it relies on runtime behaviour, dynamic analysis is generally independent of programming languages.
- Realistic view of the system: Dynamic analysis provides insights into the actual interactions and dependencies in a live system, considering factors like load balancing.

While dynamic analysis provides a runtime perspective, it is limited by:

- Incomplete coverage: The accuracy of dynamic analysis depends on the test coverage or user interactions that trigger the service calls (see [6]).

- Potential performance impact: Instrumenting microservices to collect telemetry data can introduce performance overhead.

Hybrid Approaches

Hybrid approaches combine the strengths of both static and dynamic analysis. They can utilise static analysis to identify potential dependencies and then use dynamic analysis to confirm or refine those dependencies based on actual runtime behaviour. This can result in a more comprehensive and accurate SDG that reflects the complexities of a microservices system. Hybrid approaches can also leverage dynamic analysis to uncover aspects like service popularity and usage patterns, which can be overlaid onto the statically derived graphs.

Advanced Graph-based Techniques

When microservice architectures are modelled as graphs, various graph-based techniques can be applied to analyse and understand their structure and behaviour. Here are some techniques:

Path

A path in a graph is a sequence of nodes connected by edges. In a microservice architecture, paths represent the flow of requests between services. By analysing paths, you can understand the different ways requests are processed and identify potential performance issues.

- Critical Path: Identifies the longest path of dependencies, representing the maximum latency that could occur due to cascading interactions (see [7]).
- Shortest Path: Represents the minimal number of intermediate services required for communication between two services.
- Longest Path: Maximum number of intermediate services for communication between two services of a given call graph.

Centrality

Centrality measures identify the most important nodes in a graph. Different centrality measures focus on different aspects of importance. For example, degree centrality measures the number of connections a node has, while betweenness centrality measures how often a node lies on the shortest path between other nodes. Applying centrality measures to a microservices graph can help identify critical services that require special attention in terms of monitoring and maintenance.

Community Detection

This technique identifies clusters of densely connected nodes within a graph. Applying community detection to a microservice graph can help identify groups of services that perform related functions, which can be useful for decomposing the system into smaller, more manageable units.

Graph Similarity

This technique compares the structure and attributes of two graphs to determine their similarity. Graph similarity can be used for tasks like anomaly detection by comparing the current state of the system to a library of known anomalous patterns.

Graph Kernels

These are functions that measure the similarity between two graphs, often by comparing their substructures. Graph kernels can be used for tasks like classification or regression, for example, predicting the performance of a microservice system based on its graph structure.

Multimodal Graph Analysis

Multimodal Graph Analysis focuses on combining different graph types, such as dependency graphs, causal graphs, and call graphs, to create a more holistic and comprehensive view of the system's behaviour. By integrating multiple perspectives, developers and operators can gain deeper insights into system dynamics, dependencies, performance bottlenecks, and security vulnerabilities.

Graph Neural Networks (GNNs)

GNNs, a powerful class of machine learning models designed to operate on graph-structured data, are being increasingly applied to analyse and learn complex patterns and relationships from SOA and microservices graphs. GNNs can be used for tasks like anomaly detection, root cause analysis, performance prediction, and security vulnerability assessment.

Dynamic Graph Analysis

Microservice systems are inherently dynamic, with services being scaled up or down, updated, and redeployed frequently. Dynamic graph analysis aims to develop techniques for analysing and visualising the evolution of graphs over time, enabling developers and operators to understand how the

system's structure and behaviour change in response to various events. They can be also used to identify the "diamond" patterns.

Conclusion

Graph-based modelling is a powerful tool for navigating the complexities of SOA and microservices architecture. By understanding these graph-based models, software architects and engineers can effectively address the complexities of modern software architectures, leading to improved system design, development, and operation.

References

[1] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, 2018, pp. 81–86. doi: 10.1109/COMPSAC.2018.10207.

[2] S. Luo *et al.*, "An In-Depth Study of Microservice Call Graph and Runtime Performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3901–3914, 2022, doi: 10.1109/TPDS.2022.3174631.

[3] A. Tiwari, "Cache Me If You Can: Taming the Caching Complexity of Microservice Call Graphs," 2024, *Abhishek Tiwari*. doi: 10.59350/cq9aw-a9821.

[4] S. Chakraborty, S. Garg, S. Agarwal, A. Chauhan, and S. K. Saini, "CausIL: Causal Graph for Instance Level Microservice Data," in *Proceedings of the ACM Web Conference 2023*, pp. 2905–2915. doi: 10.1145/3543507.3583274.

[5] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Causal Inference Techniques for Microservice Performance Diagnosis: Evaluation and Guiding Recommendations," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, IEEE, 2021, pp. 21–30. doi: 10.1109/ACSOS52086.2021.00029.

[6] A. Tiwari, "Microservice architecture of Meta," 2024, *Abhishek Tiwari*. doi: 10.59350/7x9hc-t2q45.

[7] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, USENIX Association, 2022, pp. 655–672. Available: https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou