STIGS: Spatio-Temporal Interference Graph Simulator for Self-Configurable Multi-Tenant Cloud Systems

Iqra Zafar*
Hasso Plattner Institute
University of Potsdam
iqra.zafar@hpi.de

Christian Medeiros Adriano
Hasso Plattner Institute
University of Potsdam
christian.adriano@hpi.de

Holger Giese†
Hasso Plattner Institute
University of Potsdam
holger.giese@hpi.de

ABSTRACT

The finer-granularity of microservices facilitate their evolution and deployment on shared resources. However, resource concurrency creates elusive interdependencies, which can cause complex interference patterns to propagate in the form of performance anomalies across distinct applications. Meanwhile, the existing methods for Anomaly Detection (*AD*) and Root-Cause Analysis (*RCA*) are confounded by this phenomenon of interference because they operate within single call-graphs. To bridge this gap, we develop a graph formalism (Spatio-Temporal Interference Graph - *STIG*) to express interference patterns and an artifact to simulate their dynamics. Our simulator contributes to the study and mitigation of interference patterns as a performance phenomenon that emerges from regular resource consumption anomalies.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**;

KEYWORDS

Microservices, Anomaly Propagation, Interference, Multi-tenant Cloud Systems, Self-Configuration

ACM Reference Format:
Iqra Zafar, Christian Medeiros Adriano, and Holger Giese. 2024. STIGS: Spatio-Temporal Interference Graph Simulator for Self-Configurable Multi-Tenant Cloud Systems. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3629527.3653664

1 INTRODUCTION

In the ever-evolving landscape of cloud computing, microservices have emerged as a dominant architectural style, enabling more flexible and scalable applications. This style relies on a finer-granularity of functions and more radical resource sharing among different applications. However, this strategy increases overall system complexity by adding elusive interdependencies among microservices [6] from distinct applications.

Definition 1.1. **Interference** happens when two services that have no logical dependency (caller-callee relation) compete for the same resource (compute, memory, I/O) to the extent that they affect each other's performance (e.g., throughput, latency) [9].

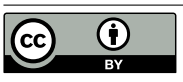
Contrary to the caller-callee relations [5], in application call-graphs and abstract syntax trees, these new interference-enabling interdependencies are more elusive because their presence and flow of direction are not deterministic. Instead, interdependencies might appear and disappear according to the non-stationary patterns of the applications' usage and the work of load balancing or self-configurable service placement mechanisms. Therefore, cross-application services interference confounds the outcome of traditional microservice diagnostic methods like Anomaly Detection (*AD*) and Root-Cause Analysis (*RCA*) [4, 11], as these methods rely on stable and predictable call-graph dependencies [5].

While self-configuration solutions can dynamically adapt to changes in the application usage [3], multi-tenant systems require more involved approaches [10]. For that, various interference mitigation (*IM*) methods have been developed - originally, for virtualized cloud environments [9] and, lately, for microservices [1, 7]. Nonetheless, there are still at least two obstacles that prevent existing *IM* methods from reducing confounding in *AD* and *RCA* approaches: (1) limited number of covered services (four as in [1, 12]), and (2) reliance on metrics that are agnostic to the interdependencies across applications. These methods measure interference w.r.t. *sensitivity* (the susceptibility of a service to be influenced by other services) and *contention* (the service consumption demand on a resource, e.g., CPU) between service pairs, but they are oblivious of the many-to-many relationship nature of interference.

Conversely, our approach overcomes these limitations by formulating the interference phenomenon as a spatio-temporal graph. Our corresponding simulation helps mitigate the probability and impact of the interference phenomenon by de-confounding the diagnostics from the *AD*, *RCA*, and *IM* methods, hence, rendering these methods more effective for complex multi-tenant cloud systems [1, 12]. We contribute with (1) a **formalism** to capture interference patterns as spatio-temporal graphs (*STIG*), (2) a **simulator** called STIGS (Figure 2) for generating interference patterns, and (3) a practical **evaluation** with three popular microservice benchmarks (Bookinfo[1], TeaStore[2] and SockShop[3]).

Definition 1.2. **Spatio-Temporal Interference Graph (*STIG*)** is denoted as $\mathcal{G} = (V, E, X_{v(t)}, X_{e(t)})$, where V are nodes representing services, E are directed edges representing interference between

*ACM Member
†IEEE and ACM Member



ICPE Companion '24 , May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0445-1/24/05
https://doi.org/10.1145/3629527.3653664

[1]Bookinfo:https://github.com/nocalhost/bookinfo
[2]Tea-Store: https://github.com/DaGeRe/TeaStore
[3]SockShop: https://microservices-demo.github.io/

services across applications, $X_{v(t)}$ are the time-varying node features (e.g., resource per service), and $X_{e(t)}$ the edge features (e.g., interference probability).

2 INTERFERENCE ANOMALY SCENARIO

As an example, assume three e-commerce applications having 14 microservices (shown in Figure 1) deployed on the same server (either *host1*, *host2*) each with a CPU of 4 cores and 10 GB of memory. The occurrence of a sudden surge of 100% in users during a flash sales event could subsequently cause an increase in the demand for these applications, e.g., from 60% to 90% CPU and memory usage from 5GB to 10GB. As the services compete for shared resources, the increased load could induce a low response time, e.g., 1000ms from the original 100ms among the resource-sharing services. This, in turn, could evolve to more severe problems like intermittent or permanent failures. Because anomalies jump across the applications' borders, one cannot rely on the individual call-graphs and performance metrics. To address this situation, the *STIG* model captures the dependencies originating both from the call-graph and the deployment graph (e.g., service placement configuration).

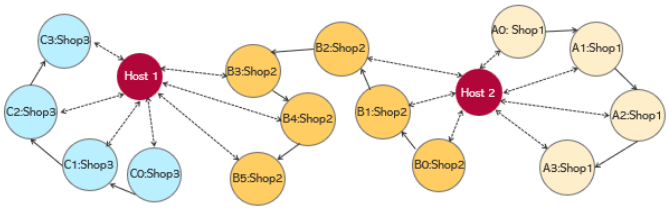


Figure 1: Knowledge Deployment Graph. Nodes colors for distinct applications (shops) and maroon/red color for host nodes. The dashed arrows for hosting service relationships and the solid arrows for caller-callee relationships.

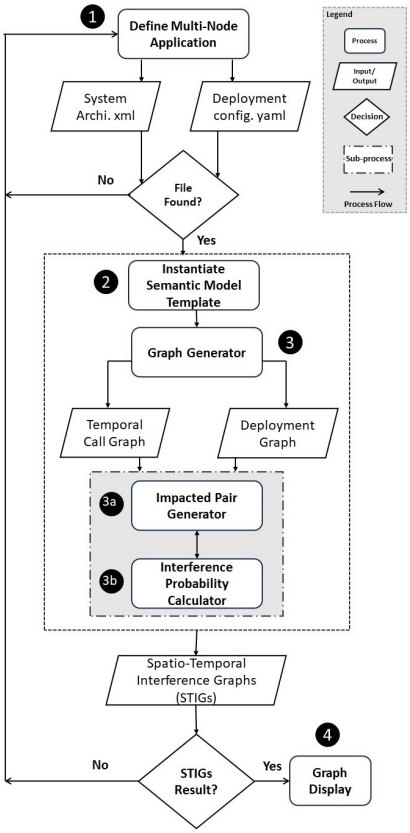


Figure 2: *STIG* Simulator Workflow

3 STIG SIMULATOR

3.1 Design and Architecture

The workflow of the STIGS depicted in Figure 2 represents a structured approach to modeling and analyzing interference in multi-node applications, which we detail next. The task **Define Multi-Node Application** generates the dependency graphs from the system architecture (*System Archi.xml*) and the deployment configuration (*Deployment config.yaml*). Based on that, we **Instantiate the Semantic Model Template** to extract distinct interference-enabling paths. The **Graph Generator** combines the set of distinct paths and the multi-tenant setup (*Deployment config.yaml*) to generate (1) a knowledge deployment graph (e.g. Figure 1) and (2) the time-annotated call-graphs, which serve as ground truth for the *STIG* generation process. The **Impacted Pair Generator** task identifies the candidate pairs of service nodes with the potential for mutual interference. The **Interference Probability Calculator** estimates the likelihood of interference by taking into account both the execution timings and their history of service anomalies. Finally, one or multiple instances of the *STIG* (e.g. Figure 3) are generated to represent distinct likelihood scenarios of anomalies induced by interference between services across applications. If at least one *STIG* was generated, the workflow ends and the simulator proceeds to **Graph Display**, where the *STIG* set is made available for analysis. We provided detailed instructions on how to install the STIG Simulator which is available for download on Zenodo[4] and Github[5].

3.2 Algorithms

To investigate the interference phenomenon, we identify the source and corresponding impact of the interference through the proposed algorithms. In Algorithm 1, we computed query predicate stack that acts as sources and targets of interference, respectively, from the Knowledge Deployment Graph (*kgraph*) and particular host node (*Host1* in Figure 1). These stack computations depend on the execution order of calls at the specific host (**line 5 and 10**). The source of interference on one or more targets services is capture as a probability measure proportionate to the magnitude of shared resources within a time window. Consequently, longer time intervals and higher resource utilization entail higher probability of interference (computed by the Algorithm 2). This involves generating a list of the impacted node pairs (*sourceStack* and the *targetStack*) based on their execution overlapping times. The algorithm first sorts these stacks by their execution start time (**line 2**) and matches the current

[4]Zenodo repository: https://zenodo.org/records/10610874
[5]https://github.com/christianadriano/STIGS-Artifact

source node and the target nodes list given their execution time conditions (**lines 3-10**). The probability of interference is derived for each source node (*curSource*) and their respective overlapping target nodes (*curTargetList*), also factoring-in their levels of shared resource usage. With that, we can estimate the interference probabilities for the *STIG* (**line 12** by calling Algorithm 3). This involves computing for each source node (*curSource*) the list of target nodes (*curTargetList*) and their corresponding execution time overlap, as well as the magnitude of the resource usage shared with each source and target nodes (**lines 3-6**). The resulting list of impacted pairs is then returned by Algorithm 2 (**line 15**).

Algorithm 1 Compute Query Predicate Stack

```
1:  procedure GENERATEQPSTACK(kgraph, host, filepath)
2:      arch = get.architecture.callgraph(kgraph)
3:      deploy = get.deploy.callgraph(kgraph)
4:      if file at filepath exists then
5:          exeOrders = Load data from filepath
6:      else
7:          exeOrders = createTempgraph(arch, filepath)
8:      end if
9:      serPaths = getDistPaths(archi)
10:     nodes.at.host = List all nodes deployed on host
11:     exe.orders= exe orders in nodes at host
12:     Initialize Query.Predicate.stack as an empty list
13:     for each exe.order in exe.orders.at.host do
14:         Get index of first service path in serPaths
15:     end for
16:     for each ser in exe.orders do
17:         Create stack.entry with service details
18:         if ser on same path of service in exe.orders then
19:             Add stack.entry to query
20:         else
21:             Add stack.entry to predicate
22:         end if
23:     end for
24:     Update start time for each entry in query and predicate lists
25:     Add a dictionary with query and predicate to
    query.predicate.stacks
26:     return query.predicate.stacks
27: end procedure
```

Using this information, we can construct Spatio-Temporal Interference Graphs (STIGs) as described in Algorithms 1,2 and 3. The STIG, as seen in Figure 3, consists of nodes as services, solid edges as service calls within the same application, and the dotted edges standing for interference paths. The weights on the interference edges can be initialized with prior probabilities based on temporal execution overlap across application services sharing the same resource (worker-node).

4 EVALUATION CASE STUDY

We deploy three popular benchmarks (**BookShop**, **TeaShop**, and **SockShop**) on a Kubernetes cluster and generate traces by injecting requests (10 to 1000) to their front webpages. Traces are collected based on the following Table 1 configurations. The "Number of

Algorithm 2 Compute List of Impacted Pairs

```
1:  procedure IMPACTEDPAIRLIST(sourceStk, targetStk)
2:      Sort both input sets by start of execution
3:      while sourceStk is not empty do
4:          Pop curSource from sourceStk
5:          while endTime of curSource > starting.time.target at
    head of targetStk do
6:              Pop curTarget from targetStk
7:              Put curTarget into curTargetList
8:              if endingTime of curSource is < ending time of
    curTarget then
9:                  Set starting time of curTarget to endingTime of
    curSource
10:                 Push it back to stack
11:             end if
12:             Append ComputeSTIGProb (curSource,
    curTargetList) to resultList
13:         end while
14:     end while
15:     return resultList
16: end procedure
```

Algorithm 3 Compute STIG Interference Probability Edges

```
1:  procedure COMPUTESTIGPROB(curSource,curTargetList)
2:      totalSourceT=curSource.endTime-curSource.startTime
3:      for each node in curTargetList do
4:          totalTargetT=min(curTarget.endTime,
    curSource.endTime)-curTarget.startTime
5:          curMag=curSource.resUsage+curTarget.resUsage
6:          return {source, target, prob, mag}=curSource,
    curTarget, totalTargetT / totalSourceT, curMag
7:      end for
8:  end procedure
```

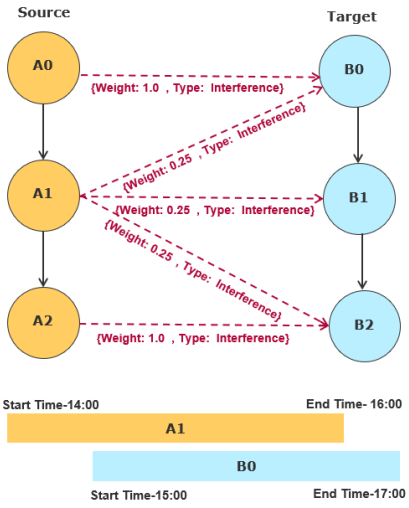


Figure 3: Spatio-Temporal-Interference-Graph. (*STIG*). Nodes are services, solid edges are calls within one application, and the dotted edges are interference paths

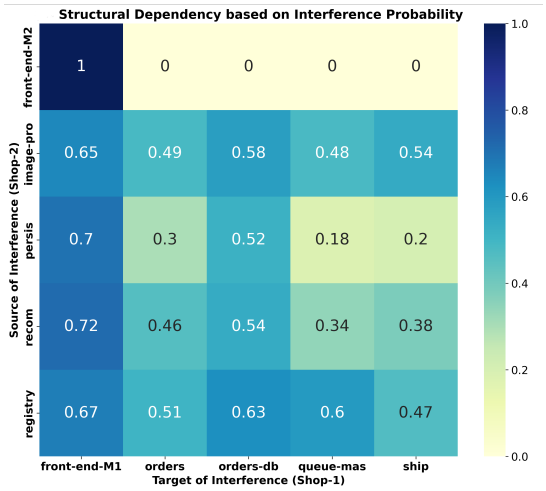


Figure 4: Structural Dependency Matrix: consolidates the averages of interference across a *STIG* set.

Requests" column shows how many requests are made in each configuration. This starts at 10 requests in *config1* and increases progressively, reaching up to 1000 requests in *config11*. The "Rate" of request is every 1 min. These generated traces will help in our analysis in combination with STIGs. Traces dataset is available at simulator's Github repository.

Table 1: Configuration of Traces Generation

Config	of Requests	Rate
config1	10	every 1 min
config2	20	every 1 min
config3	30	every 1 min
...	...	...
config10	800	every 1 min
config11	1000	every 1 min

4.1 STIG Analysis

To visualize the cause-effect phenomenon on generated STIGs, we extracted only the source and target pairs of the front-end service based on the maximum interference effect and obtained all associated source and target pairs. As a reference, Figure 4 shows a *structural dependency matrix* (*SDM* [2]) representing the interference probabilities (*STIG* edges) between source and target services (*STIG* nodes) of **SockShop** and **TeaShop**, where the darker colors represent higher probability. In the *SDM*, the *front-end-M1:shop1* shows the highest probability (1.0) of being interfered with by *front-end-M2:shop2*, which stems from the assumed determinism of these services starting simultaneously. Conversely, as the effect of interference propagates, there is a lower interference probability, which reflects smaller execution overlap between downstream services.

4.2 Reconfiguration Plan

The reconfiguration plan involves ranking the services with respect to the highest probability of necessity and sufficiency of being the culprit of the anomaly induced by interference. Because interference happens both ways, the plan can attribute source and target to anomalous services in either side of an interference association. For this, we monitored and collected traces from shops (Table 1) and performed probabilistic analysis on them. Probability of Necessity (*PN*) consists of the chance that an effect (anomaly) on a target node ($Y = 1$, i.e., Y) is caused (interfered) by an anomaly on a source node ($X = 1$, i.e., X), given that there is a history of absence of anomaly on the target node ($Y = 0$, i.e., Y') and there is an absence of anomaly on the source node ($X = 0$ or X'). Formally, from Pearl [8], *PN(Y,X) = P(Y,X|Y',X')*. The Probability of Sufficiency (*PS*) is the reverse case *PS(Y,X) = P(Y',X'|Y,X)*, while the probability of both Necessity and Sufficiency (*PNS*) is the weighted average *PNS(Y,X) = P(X,Y)PN(Y,X) + P(X',Y')PS(Y,X)*. Among the various approaches to compute these probabilities, we adopted the formulations in [8] (section 19.3.3) that assume causal *exogeneity*[6] and *monotonicity*[7]. The formulations are the following *PNS = P(Y|X) - P(Y|X')*, *PN = PNS / P(Y|X)*, and *PS = PNS / [1 - P(Y|X')]*. The results in Table 2 show that *PN* is more than two orders of magnitude higher than *PS* and *PNS*. This means that one can focus primarily on tackling the *necessary* sources of the induced anomaly, i.e., *product-page* and *reviews*. To mitigate the interference-induced anomalies on *teastore-webui*, one could reconfigure the deployment graph in a way that this microservice is placed on a *worker-node* where there are no instances of the *product-page* and *reviews* microservices. Meanwhile, the *STIG* simulated data also informs us that the other anomalies (e.g., on (*teastore-auth* and *teastore-image* services) are not induced by an interference. For these cases, the solution is to add more resources (compute, memory) to their corresponding worker-nodes. For more details, an analysis is available under the artifact Github repository data/traces folder[8].

Table 2: Results for two interfering service pairs[a]

Item	Pair 1	Pair 2
X	product-page	reviews
Y	teastore-webui	teastore-webui
PN	8.40%	24.97%
PS	0.05%	0.05%
PNS	0.05%	0.05%

[a] The only services with anomalies in **BookShop** are *product-page* and reviews, whereas in **TeaShop** only the *teastore-webui*, *teastore-auth*, and *teastore-image* have anomalies. However, there were only two pairs of services with joint probabilities $P(Y, X) > 0$ (shown in the table).

5 CONCLUSION AND FUTURE WORK

We presented a novel approach to the problem of service interference in multi-tenant microservice architectures, where concurrency over shared resources induces the propagation of elusive anomaly patterns. Our formalism and simulator are a contribution to the study of interference anomalies and the mitigation of this complex emergent phenomenon. The artifact components and the interference simulation can be easily extended to new performance anomaly scenarios. In future work, we plan to study the scalability and latency of the simulator within larger and more heterogeneous deployments.

[6] *Exogeneity* = no hidden confounders beyond the detected anomalies
[7] Monotonicity of the causal effects, i.e., anomalies cannot cancel each other.
[8] Analysis Details: https://github.com/christianadriano/STIGS-Artifact/blob/main/data/traces/excel-filtered-combined-services-anomalies.xlsx

REFERENCES

[1] Madhura Adeppady, Paolo Giaccone, Holger Karl, and Carla Fabiana Chiasserini. 2023. Reducing Microservices Interference and Deployment Time in Resource-constrained Cloud Systems. *IEEE Transactions on Network and Service Management* (2023). https://doi.org/10.1109/TNSM.2023.3235710

[2] Tyson R Browning. 2015. Design structure matrix extensions and innovations: a survey and new opportunities. *IEEE Transactions on engineering management* 63, 1 (2015), 27–52.

[3] Vincent Bushong, Amr S. Abdelfattah, Abdullah A. Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, and Miroslav Bures. 2021. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. *Applied Sciences* 11, 17 (2021). https://doi.org/10.3390/app11177856

[4] Shenghui Gu, Guoping Rong, Tian Ren, He Zhang, Haifeng Shen, Yongda Yu, Xian Li, Jian Ouyang, and Chunan Chen. 2023. TrinityRCL: Multi-Granular and Code-Level Root Cause Localization Using Multiple Types of Telemetry Data in Microservice Systems. *IEEE Transactions on Software Engineering* (2023).

[5] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. 2018. A holistic evaluation of docker containers for interfering microservices. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 33–40.

[6] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)* 18, 2 (2018), 1–23.

[7] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, deeper, easier: crowdsourcing diagnosis of microservice kernel failure from user space. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 646–657.

[8] Judea Pearl. 2022. Probabilities of causation: three counterfactual interpretations and their identification. In *Probabilistic and Causal Inference: The Works of Judea Pearl*. 317–372.

[9] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding performance interference of I/O workload in virtualized cloud environments. In *2010 IEEE 3rd international conference on cloud computing*. 51–58. https://doi.org/10.1109/CLOUD.2010.65

[10] Miguel G Xavier, Kassiano J Matteussi, Fabian Lorenzo, and Cesar AF De Rose. 2016. Understanding performance interference in multi-tenant cloud databases and web applications. In *2016 IEEE international conference on big data (big data)*. IEEE, 2847–2852.

[11] Ruyue Xin, Peng Chen, and Zhiming Zhao. 2023. Causalrca: Causal inference based precise fine-grained root cause localization for microservice applications. *J. of Systems and Software* (2023).

[12] Chaobing Zeng, Fangming Liu, Shutong Chen, Weixiang Jiang, and Miao Li. 2018. Demystifying the Performance Interference of Co-Located Virtual Network Functions. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 765–773. https://doi.org/10.1109/INFOCOM.2018.8486246