Murphy: Performance Diagnosis of Distributed Cloud Applications

Vipul Harsh[1,2], Wenxuan Zhou[2], Sachin Ashok[1], Radhika Niranjan Mysore[3]
P. Brighten Godfrey[1,2], Sujata Banerjee[3]
[1]University of Illinois Urbana-Champaign [2]VMware [3]VMware Research

ABSTRACT

Modern cloud-based applications have complex inter-dependencies on both distributed application components as well as network infrastructure, making it difficult to reason about their performance. As a result, a rich body of work seeks to automate performance diagnosis of enterprise networks and such cloud applications. However, existing methods either ignore inter-dependencies which results in poor accuracy, or require causal acyclic dependencies which cannot model common enterprise environments.

We describe the design and implementation of Murphy, an automated performance diagnosis system, that can work with commonly available telemetry in practical enterprise environments, while achieving high accuracy. Murphy utilizes loosely-defined associations between entities obtained from commonly available monitoring data. Its learning algorithm is based on a Markov Random Field (MRF) that can take advantage of such loose associations to reason about how entities affect each other in the context of a specific incident. We evaluate Murphy in an emulated microservice environment and in real incidents from a large enterprise. Compared to past work, Murphy is able to reduce diagnosis error by $\approx 1.35\times$ in restrictive environments supported by past work, and by $\geq 4.7\times$ in more general environments.

CCS CONCEPTS

• **Networks** → **Network management**; *Network monitoring*;

KEYWORDS

performance diagnosis, cyclic dependencies, enterprise networks, microservices

ACM Reference Format:
Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, P. Brighten Godfrey, Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications . In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604877

1 INTRODUCTION

Troubleshooting IT incidents, such as slow responsiveness of a service or loss of connectivity, is getting harder due to the large number of application and infrastructure components and complexity of dependencies between them. Modern microservices-based architectures contribute to this complexity as do the increasingly disaggregated, virtualized, and distributed deployments, even spanning multiple clouds. As a result, when incidents happen, multiple teams, responsible for different infrastructure components, scramble to pinpoint the source of the outage or performance degradation.

This paper focuses on an important step of resolving incidents – *performance diagnosis*. Operators are commonly presented with an observed entity E_o (say, a backend database server) which has a problematic metric M_o (e.g., high memory usage). The goal is to find a "root cause": an entity E_r and an associated metric or property M_r which led to the observed problem (or a short list of likely root causes). Unlike fault localization works [4, 10, 11, 19, 23, 35] that infer some hidden state of components like links silently dropping packets, the challenge here is not necessarily to infer hidden metrics; indeed, performance diagnosis systems typically leverage extensive telemetry so that relevant entities (applications, containers, VMs, routers, data flows, etc.) and their associated metrics (API latency, CPU utilization, flow throughput, etc.) may be well known. Instead, the core challenge is to infer the causality relationship $(E_r, M_r) \rightsquigarrow (E_o, M_o)$.

There has been an active push in industry and in academia towards performance diagnosis – spanning from early work focused on enterprise networks [11, 27], to applications in data centers [25], to recently-renewed interest in performance diagnosis for microservices [16]. We found to our surprise, however, that these designs are limited in either accuracy or applicability of their model to our target environment. A key design choice is the kind of input they utilize to model dependencies (i.e., potential functional influence) between entities in the system, which we can classify into three types.

(1) *No dependency knowledge:* ExplainIt [25] requires no knowledge of the dependency structure, but we found this results in low accuracy.
(2) *Directed acyclic graph (DAG) of dependencies:* Sage [16] requires a known dependency DAG, specifically the microservice call graph. Unfortunately, real-world enterprise environments contain many cyclic dependencies. Even in the restricted setting of microservices, cycles exist in practice among services within the execution of a single request [33]. Furthermore, independent requests affect each other indirectly due to resource utilization, and infrastructure components like CPUs, NICs, virtual routers, etc. introduce further bidirectional (and thus cyclic) influence.

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09...$15.00
https://doi.org/10.1145/3603269.3604877

Furthermore, the specific direction of a dependency between entities can sometimes be hard to determine.

(3) *Relationship graph:* Another option is to model known potential dependencies, but such that there may be cycles and the relationships themselves are loosely defined, in contrast to causal dependencies. We refer to this as a *relationship graph.* NetMedic [27] takes this approach, and is perhaps closest to meeting our needs, but we found it too resulted in poor accuracy. This may be because its inference algorithm uses fixed heuristics (as opposed to a learning-based approach) that are unable to capture complex and variable patterns in our environments.

In this paper we seek to design a performance diagnosis scheme which (a) is applicable to common enterprise environments, including cloud applications and enterprise infrastructure, while (b) achieving high diagnosis accuracy. Intuitively, this involves a choice of input information, and algorithms to effectively reason about that information.

First, Murphy is built to use telemetry from common enterprise monitoring software. Monitoring platforms, including the one we will use to test Murphy, can see entities in the system like VMs, containers, hosts, routers, TCP flows, etc., as well as relationships between them – for example, VM v_1 is located on host h_5 and it has a TCP connection to v_2. Such relationships imply a likely influence between entities, and we want to make use of this commonly-available information to improve accuracy. But the directionality of that influence might be either or both, in a way that might be dependent on the application, API call, scenario, and moment in time, and it might even be a weak, non-consequential influence. Therefore, Murphy models entity dependencies with a graphical model that can accommodate cycles, including bidirectional edges that avoid assumptions on the specific nature of the relationship between two entities; cycles thus may be the common case in the input. This input is of type (3) (a relationship graph), not unlike NetMedic.

Second, to achieve high accuracy, we need a powerful reasoning algorithm to predict causality in the relationship graph. We therefore design a new learning-based method: Murphy uses a *Markov random field (MRF)* [29][1], a type of probabilistic graphical model that can represent nodes that might simultaneously affect and be affected by their neighbors. We found it was important to train the model on demand, and developed an adaptation of Gibbs sampling for inference. Using this design, Murphy learns a joint distribution of all entity metrics from historical values, which it uses to predict the impact of a potential root cause. Finally, after generating the root causes, Murphy goes one step further and generates simple human-interpretable explanations about the root causes using a threshold-based labeling scheme.

In addition to the design of Murphy, this paper describes the following evaluation results:

- **Metric prediction model selection:** Accurately predicting the effect of changing a metric is a key sub-component for performance diagnosis (and may also be of independent use). Using telemetry data from a large enterprise with over 300 production applications comprising ≈ 17,000 entities, we evaluate four candidate designs for this sub-task and further refine our scheme. We also show that a prediction technique which better captures cyclic influence improves accuracy, suggesting that these complex interactions are indeed present in deployed applications.
- **Diagnosis accuracy:** We evaluate Murphy and several recent schemes in two common environments, which may include cyclic inputs. **(a)** In the DeathStarBench [17] microservice benchmarking suite, we test an environment with a microservice application and its associated infrastructure which may induce cyclic relationships. Murphy achieved 86% accuracy in diagnosing injected faults, whereas NetMedic and ExplainIT achieved very low accuracy (§ 6.1). When we ignore cycles so that it is possible to run Sage on the input, it did not produce the root cause as it wasn't captured by its modeling. **(b)** We use an evaluation on real IT incidents from a large enterprise. While the incident set is relatively small, the result is promising: Murphy produces ≥ 4.7× fewer false positives than ExplainIt and NetMedic, while producing a similar number of false negatives, when taking the operators' manual judgements as the ground truth. (Sage is not included because it cannot model this environment.)
- **Diagnosis accuracy in DAG environments:** Although handling more general non-DAG environments is our goal, Murphy should also perform well when a DAG is known and is an appropriate model of the environment. We compared accuracy using the DeathStarBench environments for which Sage was designed (§ 6.3). Here, Sage performs well, averaging 77% accuracy, but Murphy performs even better with 83% accuracy, illustrating the power of MRF-based reasoning. Both NetMedic and ExplainIT perform poorly in this environment.
- **Robustness**: Although using dependency information from telemetry is useful, there is a risk that the telemetry is incomplete or has errors, as is often the case with monitoring data from a large infrastructure. We evaluate robustness by introducing a series of types of input data degradation. Overall, Murphy achieves 1.5× less diagnosis error than Sage, while NetMedic and ExplainIT perform much worse.

Traces we generated from the DeathStarBench [17] benchmarking suite are available at [3]. Some of the ideas in this paper are being adopted in VMware Aria Operations for Networks [6], a commercial multi-cloud network observability[2] platform, to provide insights about infrastructure dependencies and problems. This work does not raise any ethical issues.

2 BACKGROUND

2.1 Enterprise network monitoring

Our target domain is cloud infrastructure for medium to large enterprises. A typical enterprise uses private clouds (i.e., on-premises data centers), as well as virtual infrastructure in public clouds. Workloads include both monolithic and microservice-based applications, on virtual machines (VMs) and containers. Environments continuously change as services and infrastructure components are brought up, scaled up or down, moved, and decommissioned. The scale of

[1] MRFs have been previously used for medical diagnosis [39], pattern recognition [12], image analysis [29] among several other applications.

[2] We use the terms observability and monitoring interchangeably.

infrastructure varies, with hundreds up to several thousands of applications in a very large enterprise.

Most enterprise IT teams use multiple monitoring tools to gain visibility into the environment, including hosts, applications, logs, and network infrastructure. Microservice-based applications often utilize additional specialized monitoring with tracing tools like Jaeger [2] and Zipkin [9] tracking application-level metrics such as RPC latency and errors.

We describe VMware Aria Operations for Networks [6], an application-aware multi-cloud network observability platform, which is our source of experimental data in this paper and is reasonably representative of common enterprise monitoring software. This platform obtains passive telemetry from multiple data sources, including: VM management platforms and SDDC controllers (which in turn monitor individual hosts and virtual networks), physical network devices (routers, switches, firewalls, etc.), APIs from public cloud providers, and Netflow/IPFIX sources for information about data flows. This telemetry includes metadata about a variety of entity types, with generally multiple performance metrics for each entity. Each metric is a time series, collected in intervals within minutes. Data older than a day is aggregated for most metrics into longer time intervals, and is stored for a rolling summarized window. Example entity types and associated metric values relevant to the present paper include:

Entity type	Example metrics
VM	CPU utilization, memory utilization, network transmit rate, receive rate, packet drops, disk read/write rate
Host	*Metrics similar to VM metrics*
Container	*Metrics similar to VM metrics*
Virtual NIC	Transmit rate, receive rate, dropped packets
Physical NIC	Transmit rate, receive rate, dropped packets, latency, interface peak buffer utilization
Flow	Session count, throughput, RTT, packet loss, retransmission ratio
Switch interface	Network rate, dropped packets, latency, interface peak buffer utilization
Datastore	Space utilization

The monitoring platform also provides entity metadata, encoding entity associations. For instance, a VM is related to its physical host, NIC, and flows that originate or terminate at it. Flows identified by 4-tuple (source IP, destination IP, destination port, and protocol) are related to their source and destination. Metadata can also encode application definitions: Operators can tag or classify VMs comprising an application and also define "tiers" within an application (e.g., web tier, database tier, etc.). Application definition can be manual, or automatic based on tags, naming conventions, and analysis of flow communication patterns [7]. The data set used in this paper has over 300 defined applications (§ 5).

Network monitoring platforms can have visibility into a single data center or an entire enterprise (depending on deployment). As such, the scale can be large.[3]

Enterprise operators use monitoring tools like the above to help gain visibility into the state of the network and applications. Some platforms also detect anomalies. However, automated diagnosis is still lacking and operators rely on manual intervention, trial and error and domain knowledge of engineers across several teams to diagnose performance issues. Often the root cause is unknown even after days of manual diagnosis and disappears after operators restart some components in an attempt to remove the offending triggers.

2.2 The need for handling cyclic dependencies

A key consequence of relying on commonly-available monitoring data in typical enterprise environments is that we must work with *highly complex, often cyclic, and often uncertain, dependencies.* In contrast, prior works [11, 16] expect, as input, clean causal dependencies between entities, like dependencies of the form (A–>B), where A depends on B but not vice versa. They also expect the totality of dependencies to form a causal directed acyclic graph (DAG).

Cyclic dependencies, in a system representation, can arise in two ways. The first is actual cyclic influences in the system. Even in the restricted setting of microservices, cycles exist in practice among services within the execution of a single request [33].[4] Different user-facing services can also affect each other indirectly as they can share resources and common downstream microservices. Even more cycles appear when we include infrastructure entities. Consider an example: a VM v_1 resides on host h (a similar relation exists in the incident in Figure 1). The CPU usage of v_1 influences the overall CPU usage of h, but because the physical CPU is a shared limited resource, this will also influence the CPU usage of another VM v_2 on h – and symmetrically, CPU usage of v_2 will affect h and v_1's CPU usage, thus forming a cyclic influence. A similar effect would occur with Kubernetes pods in burstable or best-effort mode on a shared node, as well as other shared resources like memory, disk, NIC drop rates and throughput, and indirect influence through TCP flows. All of these dependencies are present in our enterprise environment.

The second source of cyclic dependencies is when the direction or existence of dependency between two entities is unclear, and thus *the relationship graph over-approximates the actual influence*. For example, suppose there is a TCP flow from a web front end VM v_1 to another VM v_2. This flow will be visible in the monitoring system, but its exact nature is not. It might be that the performance of the application at v_1 depends on the RTT or throughput of the flow, which in turn might depend on CPU resources in a back-end database running at v_2, so that v_1 indirectly depends on v_2. Resources at v_2 may also depend on v_1, as handling the requests in the flow involves added work. Both of these patterns exist in the incident shown in Figure 1. But either of these possible influences might turn out to be negligible, if, for instance, the flow was logging information in the background and not involved in the web front end's primary traffic. These and other possibilities cannot be discerned only through metadata. Past works such as Orion [14] try to automatically infer the direction of influence, but require fine grained ($\approx$ 10 ms) timing information of every single request. This is out of scope for our enterprise environment.[5] Hence we add an association in both directions as an over-approximation of actual influence, creating many additional cycles and leaving it to the

[3] Large monitoring deployments can have hundreds of thousands of VMs monitored by a single instance of an observability platform, and larger enterprises can deploy multiple instances with federation visibility.

[4] In fact, [33] identified lack of cycles as an unrealistic aspect of current microservices testbeds.

[5] If more detailed, accurate telemetry were available – for example, via eBPF – Murphy could use it. However, true cyclic dependencies will still be present.

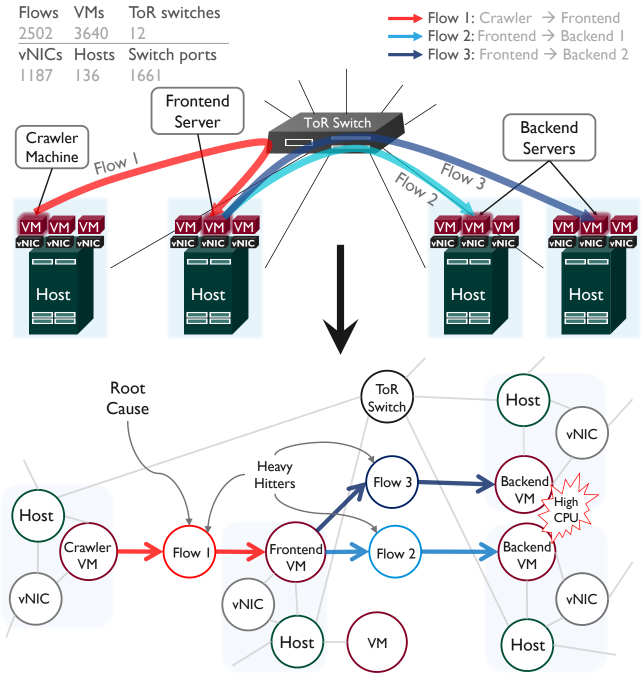


Figure 1: Production incident. Shown at the top is a physical topology associated with an incident where a crawler VM was sending too many queries to the frontend at a high rate. This resulted in the front-end initiating a large number of requests to the backend, causing high CPU load and application unresponsiveness. The bottom figure shows the relationship graph corresponding to the physical topology that Murphy built to analyze this incident. Only a subset of the entities are shown but the top left table enumerates the total no. of entities in our relationship graph. On analysis, Murphy correctly ascertained flow 1 as the root cause for the high CPU load at the backend and flagged it for the operator to handle.

diagnosis algorithm to reason about actual influence in a particular incident.

The above effects combine so that cycles are the norm. Across our data set of 13 incidents in our enterprise environment, on average, the relationship graph had over 2000 cycles of length 2 and over 4000 cycles of length 3, and *all* VMs of affected applications in every incident were involved in at least one cycle.

2.3 Related work

There is a broad set of work that attempts to automate various aspects of troubleshooting, such as routing incoming tickets to the right team [18], orchestrating active probes to check availability or latency [21], localizing faults [4, 10, 11, 19, 23, 35] like links silently dropping packets and also performance diagnosis [16, 25, 27]. It is helpful to understand work in two main subareas.

Fault localization. A rich body of work [10, 11, 14, 26, 28, 31, 35, 38] performs fault localization, distinct from ours, in which components have well known quantifiable failure signatures. For example, Sherlock [11] models entities as being in one of three states (up, troubled, or down), and other works focus on even more specialized faults than Sherlock, e.g., physical network packet drops or link loss or delay [10, 28, 35, 38]. In contrast, our work is intended for performance diagnosis, where there is not a common definition of failure signature and diagnosis seeks to answer richer metric-related questions of the form "which component's performance metrics influence the observation the most?"

We considered trying to repurpose the underlying models of the above work for our goals, but their models do not meet the needs of our environment. [11, 14] rely on detailed models of how an entity depends on another, obtained through fine-grained (e.g., 10 ms granularity) packet timing which is not available in general enterprise environments. Also, as noted in [27], Bayesian techniques [11, 26, 31, 38] can only model acyclic dependencies; the cyclic case is fundamentally more challenging due to the resulting complex interactions among entities. We have to work with entity topologies that contain many cycles, due to lack of detailed dependency knowledge as well as true cyclic influence.

Performance diagnosis. Performance diagnosis tools solve a problem that is closest to ours. We therefore discuss them in more detail.

ExplainIt [25] performs pairwise correlations between metrics of the observed problem and of each candidate root cause.[6] However, as we will show, simple metric correlation-based analysis can yield inaccurate results as it does not take into account the topological structure between entities.

NetMedic [27] uses a dependency graph to capture the known entity dependency structure. It labels edges with weights based on pairwise correlation between neighbors using historical metric values, augmented with heuristics to reduce weights when metric values are roughly normal, and remove or coalesce redundant or aggregate metrics. Finally, it ranks root causes based on a geometric-mean of path weights, and a score of the global downstream impact of the candidate root cause. This approach can have a similar limitation as ExplainIt due to use of pairwise correlations. In general, NetMedic's inference is based on fixed heuristic rules (ignoring "normal" influence, geometric-mean weighting, etc.) which can be brittle in real environments. In our tests, NetMedic's accuracy was low, except with lenient definitions of the root cause (§ 6). This suggests to us that more powerful and flexible learning-based approaches are needed.

Sage [16] uses a probabilistic distribution over metrics to identify the resources in a microservice that cause QoS violations. While its model can do the sort of flexible reasoning we target, it employs a large neural network superimposed on a directed acyclic graph representing causal dependencies between microservices. It is unclear how to adapt this model to handle cycles. Cycles are the common case in our target environments, even microservice-based environments (§ 2.2). Using data like the enterprise monitoring telemetry we work with – where there are numerous cycles amid hundreds of applications of essentially arbitrary functionality – there is no clear way to produce an acylic graph. As we show later, not handling cycles results in not being able to model all relationships which

[6] ExplainIt also was designed to answer multi-node conditional correlation queries interactively posed by the user, which is an assisted rather than fully-automated diagnosis that falls outside our use case.

ultimately makes Sage incapable of producing the right root cause (§ 6.1), so it is effectively inapplicable in such environments.

Provenance-based troubleshooting systems [13, 22, 34, 36, 37] track all events and the causality between them to find the root cause of a performance error using the event DAG. However, obtaining such detailed monitoring data requires request- or packet-level tracing which is not feasible in most enterprise networks.

2.4 Summary of goals and existing methods

In order to meet our goal of performance diagnosis for common enterprise cloud environments, we need a scheme that (1) utilizes common monitoring information, which implies handling complex dependency topologies that lack detailed dependencies and contain many cyclic relationships; and (2) achieves high accuracy. To the best of our knowledge, past approaches do not achieve these goals. Yet we can draw two important ideas from past work. First, NetMedic's idea of using an arbitrary directed graph to model dependencies makes no assumptions about cycles, allowing us to model directed dependencies where we have them, and use bidirectional dependencies otherwise. Second, Sage's approach involves an idea of using counterfactual reasoning that offers a more principled way to avoid the assumptions of heuristic-based approaches, by phrasing diagnosis as a *what-if* question: *If I changed metric M_r to a certain value, what effect would that have on the problematic metric M_o?* However, for the reasons noted above, we will need entirely new algorithms to make use of these high-level ideas in our environment.

3 USING MURPHY FOR PERFORMANCE DIAGNOSIS

We describe a typical troubleshooting experience with Murphy (see Figure 2 for example inputs/outputs of the tool).

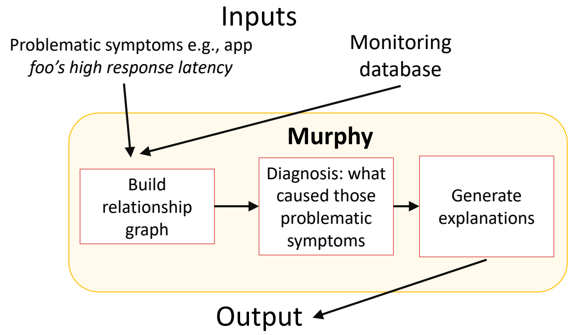
Let's suppose an incident is reported in the IT infrastructure about a client facing service *foo* experiencing performance degradation. IT admins can run Murphy providing, as input, a *problematic symptom* – a problematic metric M_o of an entity E_o they want to know the reason for. This could be, for example, high memory usage of a SQL server used by *foo*. The problematic symptoms can also be obtiained via other methods such as by finding all entity metrics related to an affected application that are above thresholds preset by operators or via automated tools such as [15]. As the identification of problematic symptoms is not central to our design, we refer to Appendix A.1 for a more detailed discussion.

For each problematic symptom (M_o, E_o), Murphy outputs a ranked list of root cause entities for that symptom. Additionally, for each root cause entity, Murphy produces a causal explanation chain tying it back to the symptom.

4 DESIGN

There are three parts to the Murphy system (Figure 2):

(1) The first part constructs the *relationship graph* using data obtained from the monitoring system (§ 4.1).
(2) For each problematic symptom in the input, Murphy's inference algorithm generates candidate root causes (§ 4.2) using a Markov random field (MRF). The MRF models a joint probability distribution of all entity metrics and is learned via historical values.
(3) Murphy generates explanations for the root causes tying them back to the problematic symptoms (§ 4.3).


Inputs
Problematic symptoms e.g., app *foo's high response latency*
Monitoring database
Murphy
Build relationship graph
Diagnosis: what caused those problematic symptoms
Generate explanations
Output

Root cause with explanation

e.g., Root cause: **Crawler machine**
Entity A (Crawler machine) sent high requests to Entity B (front-end).
Entity B (front-end) sent high requests to Entity C (back-end).
Entity C (back-end) faced high load and CPU usage.

Figure 2: Murphy workflow

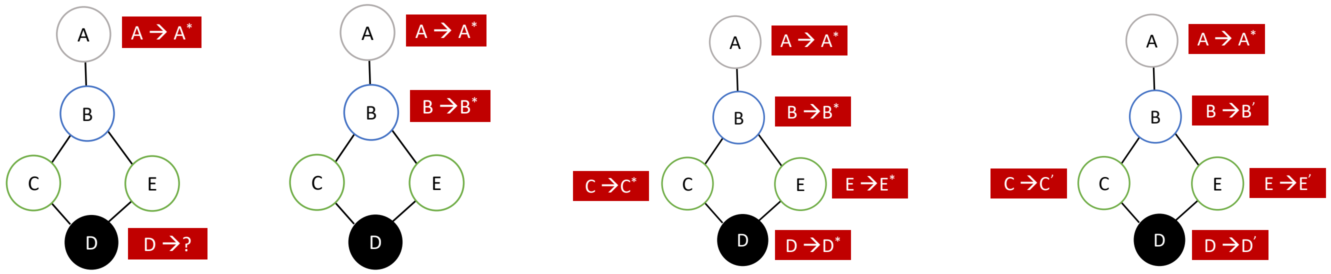
Our core contribution is in the design of the MRF framework which lets us utilize commonly available telemetry while producing accurate results. We describe each part next.

4.1 Constructing the relationship graph

Murphy employs a *relationship graph*, where the edges between entities are based on simple "neighborhood" relationships that are predefined by the monitoring software and can be easily extracted from the input. For example, a flow has edges to its source/destination VM, a VM has edges to its host and NIC, a microservice has edges to the container it resides on, and so on. This neighborhood definition is deliberately loose in order to work with common monitoring telemetry that doesn't have information about causal DAGs. Note this means that the relationship graph may have cycles.

To construct the relationship graph, Murphy makes an initial query to the monitoring database to obtain descriptions of a set of entities S relevant to the problem. If the input to Murphy is an affected application A, then S is the set of all entities that the system considers to be *members of* A. If the input specifies a problematic entity e, then S is the singleton set: $\{e\}$. Starting from S, Murphy constructs the relationship graph recursively by exploring the neighborhood of S and updating S as $S = \text{neighbors}(S)$. If the relationship graph becomes intractably large, then optionally this exploration is stopped after a few iterations. Figure 1 shows the relationship graph for a real incident. For each entity in the relationship graph, we also have timeseries data for various metrics (e.g. CPU/memory/network usage for a VM, session count and bytes sent/received for a flow, etc.).

By default, to be conservative, we add directed edges in both directions between two neighbor entities A and B in the relationship graph. The directed edges represent potential dependencies in both directions A⟶B and B⟶A. If a directional dependency is known between two entities, such as in the case of caller and callee



(a) Step 1 **(b) Step 2** **(c) Step 3** **(d) Step 4**

Figure 3: Inference algorithm- evaluating whether A is the root cause for D's high metric value for a toy relationship graph with only one metric per entity. For clarity, we draw an undirected edge to represent a directed edge in both directions. We (a) change A to a counter-factual value A^*, (b) resample B assuming the new value A^*, (c) resample C, D and E in a similar way, and (d) resample C, D and E, again g times for Gibbs sampling. We run the same resampling algorithm, this time starting with the current true value of A, instead of a counterfactual, and obtained the resampled value D''. After this sampling process, if $D' \ll D''$, then we classify A as a root cause for the high value of D.

microservices [16], then Murphy can incorporate that via a single directed edge. This design ensures that entity dependencies can be captured in the most general way and allows Murphy to work with commonly available telemetry information that may not include information about causal dependencies.

4.2 Performance diagnosis

For each problematic symptom (M_o, E_o) provided as an input (§ 3), Murphy separately runs the performance diagnosis algorithm.

The relationship graph lends itself naturally for performance diagnosis via an appropriately designed graphical model. Markov random fields (MRFs) are one such family of probabilistic models. A MRF is a probability model superimposed on a graph that may have cycles, making it suitable to reason about entities in a relationship graph (§ 4.1). While acyclic causal edges make Bayesian networks easy to understand, a MRF is harder to interpret making it more challenging to apply. For the same reason, inference algorithms for Bayesian networks [11] don't apply to MRFs.

We've designed a MRF framework, according to the needs of our environment (§ 2.1), with available telemetry and its scale as the focus. We describe the MRF framework in two sub-parts: (a) model and (b) inference algorithm.

Model: To reason about a problematic (entity, metric) symptom, provided as input (§ 3), Murphy models the distribution of metrics for all entities as a MRF; call this distribution P_G. P_G denotes the joint probability of all entity metrics taking certain values and is calculated using a general directed relationship graph, that can potentially have directed cycles. P_G is defined as a product of individual entity factors P_v for every entity v:

$$P_G = \frac{1}{Z} \prod_{v:V(G)} P_v(v|\text{in\_nbrs}(v)).$$

In the above,

- $V(G)$ denotes the entities in the relationship graph G.
- in_nbrs(v) denotes the set of neighbor entities w such that there is an edge from w to v in the relationship graph.
- P_v is a function that takes as input, the values of metrics of v and v's incoming neighbors (in_nbrs(v)), and outputs a probability score between 0 and 1.
- Z is a (unknown) normalizing constant that ensures the probabilities add up to 1. The inference algorithm does not require the value of Z.

The relationship between an entity's metrics and its neighboring entities can be complex and variable across entities. Hence, the function P_v is determined by relating metrics of entity v in a time slice to the metrics of the neighbors of v in the same time slice. Specifically, Murphy learns a multivariate distribution for P_v, for all v, using a standard model such as linear regression with normal error, Gaussian mixture model (GMM), neural networks or SVMs using historical metric values. Different models could be suitable in different environments. The right choice of model can be determined by analyzing training errors in learning P_v across several entities v. In our production environment, we found ridge regression (a form of robust linear regression) to work best, evaluated using a large real world dataset (§ 6.6.1). Hence, we employ Ridge regression in Murphy for all our experiments.

Model training: Murphy doesn't keep any pre-trained models; every time Murphy is called, online training is triggered. Training online on fresh data has three advantages over training offline: (a) Applications get updated frequently, e.g. the application topology or software version might change from time to time. Using a model trained on outdated application topology or outdated software may not be ideal for diagnosis. (b) Online training eliminates the inconvenience of storing and maintaining a large number of entity models P_v. (c) Most importantly, an operator will run Murphy in the middle of an incident, so, with online training, the last few data points in the training data are from during the time of the incident. This turns out to be crucial (§ 6.5.1) as often an incident involves a pattern of metrics which hasn't occurred in the past (§ 6.2).

Every time Murphy is called, we train the linear regression model for each P_v using data from one week prior to the incident, which in our environment constitutes of a few hundreds time points for training. We didn't exhaustively explore all possibilities for the training period length, but found one week to be reasonable given using older data for training might include stale patterns from older app deployments as apps constantly get updated (also see § 6.5.2). A valuable area of future work would to be fully explore the tradeoff between using more data vs. fresher data, which could even depend on the specific incident.

Using a large number of features may cause overfitting when the training data is small. Hence, guided by the "one in ten" thumb rule [5] for regression problems (use at least 10 observations/parameter), we pick the top $B = 10$ neighbor metrics, based on their correlation with v's metrics (number of parameters is also B in a linear regression model). We also tried $B = 5$ and $B = 20$ and found training error to be within 3% of $B = 10$.

Inference algorithm: The distribution P_G provides a powerful way to reason about entities and their states. We describe the algorithm using a toy example, shown in Figure 3. Let's say we want to determine what caused a high CPU utilization at server D. To evaluate if another entity A in the relationship graph (not necessarily D's neighbor) might be responsible, we change the value of A's metric to a "counterfactual" value A', keeping the value of all other entities unchanged. Using a Gibbs-sampling like algorithm, we then resample D via the MRF to get a new value D', starting from the "counterfactual" value A' and $\mathcal{E}$, where $\mathcal{E}$ represents the original value of all entities besides A and D. We note that for resampling D, the sampling algorithm will have to first resample other entities via which A would affect D. We refer to this sampling algorithm as $P_G(D|A', \mathcal{E})$ which is described in the next paragraph. Intuitively, if this new CPU utilization D' is less than D, we can conclude that A is a contributor to the high CPU utilization at D. This probabilistic "what-if" analysis is referred to as the *counterfactuals* [20] technique. We use this idea to find out which entities can alleviate a problematic symptom (a problematic metric of an entity). Note that Murphy runs this inference algorithm on the current metric values whereas the training happens on the prior one week's metrics.

The last piece in the algorithm is the sampling method to sample D from $P_G(D|A, \mathcal{E})$ for entities A and D, which are not necessarily neighbors. Gibbs sampling is commonly used to sample from a MRF [29], but it entails executing several iterations of "pick a random entity and resample it given its neighbor entities". There are two problems with running exact Gibbs sampling. (a) It would be computationally too expensive in our environment since the relationship graph could have several thousands of entities. (b) We want to preserve the values of entities that are likely unrelated to A but might affect D. Running the sampling algorithm on those entities might destroy their values. Instead, we resample only a subset of entities that are on paths from A to D. We thus use a variant of the Gibbs sampling algorithm (illustrated in Figure 3 and detailed below).

Putting all the pieces together, the algorithm is as follows:

(1) We set the value of A's metric to a lower/higher counterfactual value: A', that is 2 standard deviations away from its current value.
(2) We consider the entities in the shortest path subgraph from A to D, defined as $\mathcal{T}$. For each entity v in $\mathcal{T}$, ordered in increasing distance from A, we resample the metrics of v from $P_v(v|\text{in\_nbrs}(v))$. The last step of this process, therefore, gives us a new sample value for D.
(3) We repeat step (2) for $W = 4$ iterations (see § 6.6.2 on how we picked W). Sampling nodes more than once, as in Gibbs sampling, helps to propagate effects across cycles in the graph (§ 6.6.2).
(4) Let d_1 be the sampled value for D thus obtained after step (3), having started with the counterfactual value A' in step (1). We also run the procedure (2)-(3), but this time start with the current true value of A, instead of the counterfactual value A', obtaining sampled value d_2. Then, we generate many such samples (5,000 in our implementation) for each of d_1 and d_2 via steps (1)-(3). If the d_1's are significantly less than d_2's (decided via a T-test), we conclude that A is a root cause for D. In our implementation, we generate 5000 samples for d_1 and d_2 each for the T-test.

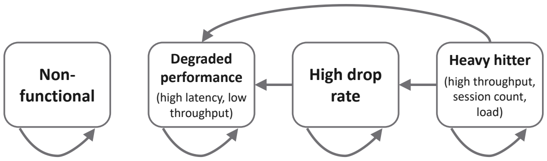

Non-functional
Degraded performance
(high latency, low throughput)
High drop rate
Heavy hitter
(high throughput, session count, load)

Figure 4: State machine encoding causal rules between entities' states for generating labeled explanation chains

We note that Gibbs sampling helps in propagating newer values across cyclic dependencies in the model. Let's say we need to resample a set of entities $\mathcal{T}$. Consider a cyclic dependency $A \rightarrow B \rightarrow A$ where A, B are entities in the set $\mathcal{T}$. If we only resample entities in S once and say we sample A before B, the newer value of B does not propagate across the dependency from B to A. Resampling A and B multiple times, as done in Gibbs sampling, solves this problem partially and improves accuracy (§ 6.6.2).

For each entity in the relationship graph, Murphy evaluates if its a potential root cause using the above algorithm. Murphy limits this search space of potential root cause entities via the following method: it runs a breadth first search starting from the problematic entity, exploring neighboring entities that have metrics above very conservative thresholds, while pruning out the rest. This reduces running time and improves precision. For fairness, we provide this pruned search space to all reference schemes that we compare with in our evaluation (this improved their accuracy).

Ranking the root causes: Once Murphy's inference algorithm produces the root cause entities, we rank them based on how anomalous their current metrics are. To do so, we consider how many standard deviations away a metric is from its historical mean value, which translates to a score for a single metric of an entity. We set the entity's score to be the score of its most anomalous metric. The ranking between the root cause entities is inversely proportional to this score.

Correlation vs causation: Note that as in [25, 27], the resulting candidate root causes have been determined to be *correlated* with the problem, but *causality* has not been determined. In the absence of precise information about causal dependencies between entities, Murphy does not guarantee causality between the root cause and the observed problems, but the candidate "shortlist" of potential root causes it outputs is still useful (as we will show later via experiments), since correlation is a necessary condition for the failure types within scope of Murphy.

Edge cases: Historical metric values may be missing for a newly introduced entity. To construct the training set for our algorithm in such a case, we use a default metric value (such as 0% for CPU usage) as a placeholder for missing values. Murphy also presents all recent configuration changes to the operator to catch problems caused by recently spawned VMs. Although the MRF framework captures a wide range of practical cases such as high CPU usage of a VM, high drop rate of a NIC, high latency of a service etc., other failure types are outside Murphy's scope (see § 7).

4.3 Explaining the diagnosis

Once Murphy finishes diagnosis and the candidate root causes have been found, we also generate human-readable explanations for them. We first assign one of the following labels to each entity based on their current metrics and conservative thresholds[7]: Faulty/Non-functional; Degraded performance; High drop rate; Heavy hitter; Okay.

We encode prior domain knowledge about causal truths using a state machine (Figure 4). Each node is a possible label state and the arrows indicate causal truths e.g. "Heavy hitter flow can cause high drop rate on a virtual NIC" or "Heavy hitter flow can cause high load on a VM". Although simple, such a labeling scheme produces semantically meaningful explanations for each root cause. Once we've decided the entity labels for every entity, we trace paths from root cause to affected application entities in such a way that each edge in the traced path respects the label causality rules described above. Note that this step does not affect the selection of root causes, and hence does not affect accuracy. We found it was convenient to provide plausible intuition for those root causes.

5 IMPLEMENTATION AND SETUP

We implemented Murphy in Python with ~7K LOC. For reference schemes, we used the author-provided implementation for Sage and our own implementation of NetMedic and ExplainIT as their code wasn't available publicly. We test the schemes in two environments: (a) a cloud environment of a large enterprise running many production applications and (b) microservice-based applications (from the DeathStarBench suite [17]) running on private servers and a public cloud environment (AWS).

5.1 Setup and datasets

For our evaluation, we utilize datasets from two environments.

5.1.1 Datasets from apps in production environment: We utilize two real world datasets that we collected from a commercial network observability platform (§ 2.1) monitoring the production infrastructure of a large enterprise.

Incident dataset: We describe our experience with a dataset of 13 real incidents (Table 1), with varying complexity and resolution times ranging from a few minutes to a few hours.

For each incident, we collected data for entities up to four hops away in the relationship graph from the affected entities (e.g. "all VMs of application foo").[8] We extracted the entities involved in the resolution from the trouble ticket, and treat that as the ground truth. We note however that this human-operator-decided ground truth may not always be the true root cause (e.g. the root cause was heavy load caused by a flow session that originated elsewhere but went away after rebooting all application VMs, and the reboots were stated as the resolution).

Metrics dataset: We collected metrics of ~17K entities associated with over 300 production applications, for a period of a week. This data on its own is not sufficient to evaluate diagnosis accuracy as it does not come with information about failures. But, we can use it to run micro-benchmarks to evaluate model training accuracy, test various subroutines of the algorithm and fine-tune Murphy's algorithms on large scale production data.

5.1.2 Datasets from microservices in public clouds: We ran two microservice apps from DeathStarBench [17]:

- Hotel-reservation on a dedicated 7-node Kubernetes cluster hosted on AWS (across multiple availability zones in *us-east-2*) with each node provisioned with 4-core Intel(R) Xeon(R) series CPU, 16 GB of RAM, 32 GB of SSD, and up to 5 Gbps bandwidth capacity.
- Social-network hosted on a single-node environment on a private cloud. The node was provisioned with 8 Intel(R) Xeon(R) series CPUs and 32 GB of RAM. The microservice applications are orchestrated using Docker with all inter-container traffic traversing through localhost.

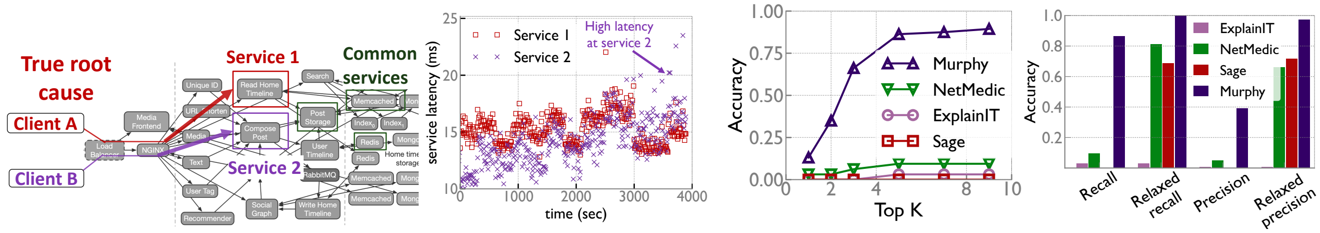
The hotel-reservation app comprises 8 services and 16 total relationship graph entities including containers and services. The social-network app comprises 24 services and 57 total entities. We obtain metrics of the application entities from two sources, viz., (1) container metrics like average CPU/memory/disk/network usage from Cadvisor [1], aggregated over 10 second intervals and (2) microservice service latencies by aggregating the individual response latencies, also over 10 second intervals. We use wrk2 [8], an open-loop workload generator to send requests to the application. We get the request traces via Jaeger [2].

We create multiple types of failure scenarios in these microservice environments:

- **Performance interference**: We set up two clients, A and B, who send requests to two different API endpoints, service 1 and 2 respectively. The API call trees of service 1 and 2 share some common backend services as shown in Figure 5a. Client A generates a high request load, overwhelming a subset of these shared downstream microservices. As a result, response latency of service 2 increases (see Figure 5b) impacting the latency observed by client B. The problematic symptom that we provide to the tool to diagnose is client B's observed latency and the true root cause is high RPS load of client A. We generate 32 variants of such scenarios by changing the RPS load sent to the services. This failure scenario was motivated by the production incident mentioned in § 4 (Figure 1).
- **Resource contention**: using `stress-ng`, we inject CPU, memory and disk faults to randomly chosen application containers, as in [16]. We generate more than 200 such fault scenarios across both the setups, varying intensity, duration (5-10 mins) and location of the faults while client workload (30-90 minutes long) is in progress.

[7] 25% CPU/memory/disk/port utilization, 0.1% drop rate, 50 TCP sessions or 1GB byte count for a flow in a single time interval.

[8] As this environment is a private cloud, both virtual and physical entities are visible to the enterprise's monitoring platform and are included in the relationship graph in this data set.



(a) Cause: Performance interference (b) Symptom: Latency spike (c) Top-K Accuracy (d) Precision and recall

Figure 5: Performance interference experiment in DeathStarBench (§ 6.1). (a) Fault scenario: client A sends a lot of requests to service 1 which overwhelms the downstream common services shared between service 1 and 2, causing high latency for service 2 for Client B. “Common containers” are containers that “common services” reside on. (b) The fault gets injected just after 3000 seconds when Client A begins to send a lot of requests. (c) Accuracy (recall) in top-K. (d) Precision and recall (see § 6.1 for definition) of various schemes in producing the true root cause. Also shown is a relaxed notion of accuracy which measures how often a scheme gets at least one entity in top 5 that’s either the true root cause or a common container or a common service.

6 EVALUATION

Evaluation goals: The goal of our evaluation is to investigate Murphy’s diagnosis accuracy compared to reference schemes Sage, NetMedic, and ExplainIT. We feed the same input relationship graph to all schemes when possible (i.e., if the algorithm can take it as input). We first evaluate scenarios in our enterprise environment and an emulated microservices environment which commonly have cyclic dependencies. Second, to enable comparison with Sage, we also consider a restricted set of scenarios where there are no cyclic dependencies between entities as done in [16]. We evaluate how robust each scheme is in handling degraded data with omissions/errors, which can be present in common telemetry. We then show several microbenchmarks to quantify the effect of the design choices and algorithmic subroutines of Murphy. Finally, we discuss the runtime performance and the sensitivity analysis.

Measuring accuracy: We measure *Top-K* accuracy, equivalently called *recall*, defined as the fraction of times the true root cause entity is among the first k entities in the ordered list of candidate root causes generated by the scheme. We use K=5 unless stated otherwise.

We also show *precision*, defined as either $1/r$ if the scheme outputs the true root cause as the rth candidate, or 0 if the output does not include the true root cause at all. The intuition for this is the operator will start at the top of the list and will have to check r suggestions before finding the right answer, and any false positives ranked beyond r won’t matter.

6.1 Performance interference in microservices

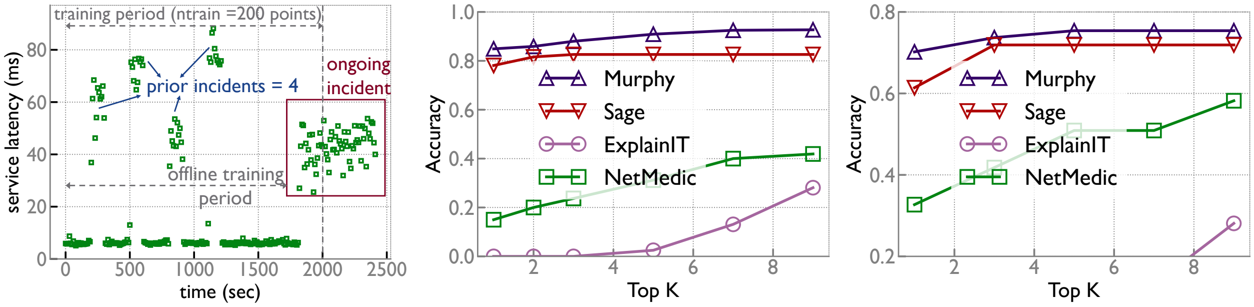
We consider the performance interference failure scenarios described in § 5.1.2 where high load at service 1 overwhelms the common downstream microservices it shares with service 2, causing high latency for service 2. To model the effect of the two services on each other, the relationship graph should have a path from service 1 to service 2 and vice-versa. This however induces a cycle in the relationship graph. Sage’s model can’t handle such cycles, and hence only models a single user-facing service and its downstream services. As a result, the true root cause (service 1) falls outside its model, preventing Sage from catching it.

Incident (observed problems)	Murphy FPs	NetMedic FPs	ExplainIT FPs
1. Two apps nodes crashed due to a plugin	6	69	93
2. App returning a 502 error	0	1	0
3. App unavailable	4	40	60
4. App slow, experiencing timeouts	10	4	4
5. App unavailable	1	1	1
6. App redirecting to a maintenance page	4	1	1
7. Heap memory issue with a node	1	1	1
8. App performance degradation	6	67	189
9. App failing with 503 error	1	1	1
10. Health check failing on 2 nodes	2	2	23
11. App redirecting to a maintenance page	6	10	22
12. Slowness in loading data	20	101	21
13. Performance alert about a node exceeding thresholds	0	0	0
Average false positives	**4.9**	**23.2**	**32.3**

Table 1: Number of false positives (FP) produced by each scheme for each incident, according to operator decided resolution (see § 6.2), for incidents in the dataset. Only FPs are shown, rather than false negatives, because the schemes were calibrated to have similar false negatives (§ 6.2).

Figure 5c shows Top-K accuracy for varying K and Figure 5d shows the precision and recall for K=5. Murphy produced the true root cause in the top-5 86% of the times, while Sage, on account of the true root cause being outside its model, did not produce the true root cause (service 1 in our example) in any case (i.e. 0 recall). Other schemes also did not produce the true root cause most of the times (accuracy < 15%).

Can Sage get *close* to the root cause, while working within the scope of its model? Specifically, identifying the common containers that are overwhelmed may be a helpful step towards the true root cause. To test this, Figure 5d also shows a very relaxed notion of recall: a scheme achieves 100% recall if its top-5 contains at least one entity that is either the true root cause, a common container or a common service. Relaxed precision is defined similarly: it is inversely proportional to the number of false positives seen by the



(a) A sample latency trace **(b) Top-K Acc (social-network)** **(c) Top-K Acc (hotel-reserve)**

Figure 6: Resource contention experiment on DeathStarBench applications: (a) service latency in a sample trace across time. The training period contains 4 prior incidents while the main incident is triggered at t = 1800s. (b), (c) Top-K Accuracy (Acc) for varying K.

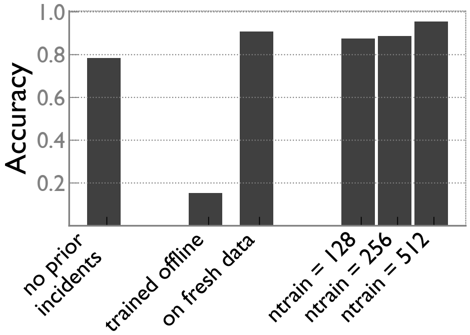


Figure 7: Various microbenchmarks with Murphy. Fresh data implies that Murphy was trained with data that included several minutes during which the incident was in progress.

operator before one of the "relaxed"-root causes is produced by a scheme. Murphy achieves perfect relaxed-recall while NetMedic also has good relaxed-recall of 0.81. Murphy has significantly better relaxed-precision and relaxed-recall than Sage, NetMedic and ExplainIT.

6.2 Incidents in production environment

Table 1 shows the number of false positives (FPs) produced by NetMedic, ExplainIT and Murphy for each of the 13 incidents in the incident dataset (§ 5.1.1) from the production environment. The table shows only FPs, because the schemes were calibrated to have similar false negatives.[9] Sage is incapable of working in this environment as it requires a causal DAG of dependencies which we don't have. This major limitation prevents us from using Sage in our production environment.

As can be seen from Table 1, Murphy overall produced 4.7x fewer false positives than NetMedic and 6.6x fewer than ExplainIT. We observed that for some incidents, both NetMedic and ExplainIT produced many false positive root cause entities that were highly correlated with the problem while Murphy was able to prune them out (incidents 1, 3, 8 and 12).

We remark on some incidents below, including discussions with network operators about the utility of Murphy's analysis:

- For incident 2 in Table 1 (illustrated in Figure 1), Murphy correctly identified the root cause entity. We validated both the root cause and the explanation produced by Murphy by talking to network operators. The top explanation chain produced by Murphy for this incident was:
  - Heavy-hitter flow from crawler VM (true root cause)
    → Front-end VM
    → Heavy hitter flow
    → High CPU on backend VM
- In one incident, 2 nodes failed health checks. Murphy flagged flows that were sending high traffic to the nodes. However, the operators rebooted the nodes to resolve the incident, so the operator-decided ground truth did not include the flows.
- In another incident, operators were unable to pinpoint the root cause of an incident that lasted only for six minutes. Interestingly, a similar incident occurred after a few weeks. Murphy flagged two flows, which were likely due to network upgrades, as culprits.

6.3 Resource contention in microservices

We consider the resource contention failure scenarios (§ 5.1.2) which don't have any cycles. Sage was designed for such scenarios [16]. Figure 6a shows the response latency in a sample scenario. For realism, as in [16], we induce up to 14 "prior incidents" where short-lived faults are injected on randomly chosen containers before the actual incident. Refer to § 6.5.3 for accuracy when there are no prior incidents.

Figures 6b, 6c show the accuracy in producing the right root cause in the top K for varying K on the x-axis. Murphy produced the true root cause with high accuracy (overall, 77% as the top candidate and 83% of the times in top-5) and had a somewhat higher accuracy than Sage (69% in top-1 and 77% in top-5). ExplainIT looks at the

[9] We calibrated each scheme's parameters to minimize false positives under the constraint that they produce recall = 1 (equivalently, zero false negatives) on a certain set of "calibration incidents". On the full set of 13 incidents, recall was not quite identical across all schemes, but was very close– all schemes had a recall in the range $[0.53, 0.56]$.

The calibration incidents were the 2 incidents for which we had full certainty in the ground truth via discussions with operators. Recall that in general, we took ground truth to be the entities involved in the operator's resolution of the incident, which in some cases may not be the true root cause.

entity whose metrics are most correlated with the high latency of the client and ends up producing entities that are closest (in the microservice communication graph) to the problematic entity such as the front-end container. NetMedic did not perform adequately, likely because of assumptions in its ranking heuristics: we found that its geometric mean based path weights can produce entirely unrelated subtrees of entities in the microservice graph.

6.4 Accuracy with incomplete data

Table 2 shows accuracy when the data is "corrupted" with some omissions or errors. Such errors can exist in the monitoring data for large infrastructure and hence a diagnosis scheme should be robust to them. To ensure a comparison with Sage, we use the same setup as § 6.3 with no cycles. We evaluate four such cases where we introduce errors in the monitoring data:

- Missing edge: we remove the association between a randomly chosen RPC and its parent (caller) RPC
- Missing entity: we remove a randomly chosen entity, including all its metrics and associations
- Missing metric: we remove a single metric (e.g. memory usage) for the root cause entity
- Missing values: for randomly selected 25% of the entities, we remove their historical values (except for the values during the incident itself, which is still present)

Such errors could arise from bugs in the tracing framework (missing edge), or missing coverage in monitoring (missing entity/metric) or a newly spawned entity (missing values). As can be seen from Table 2, both Murphy and Sage are fairly robust, incurring 6% and 10% loss respectively. Missing values have a minimal effect on Murphy since the most recent data related to the incident is still present (see § 6.5.3 and § 6.5.1); it affects Sage significantly likely because its neural networks require more data points to learn the right pattern.

6.5 Microbenchmark experiments

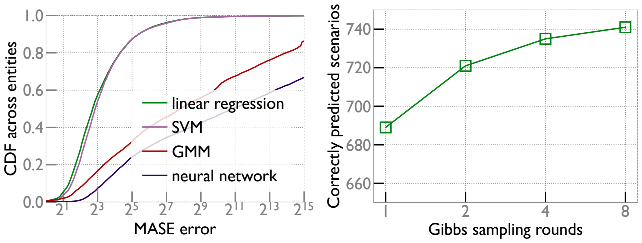
We describe "microbenchmark" experiments where we evaluate various aspects of Murphy's design.

6.5.1 Online vs offline training: Murphy learns the distribution online from past metrics of one week so that the training data includes some recent data points when the incident has happened. Another alternative could be to train the system offline, so that training time is not a concern and potentially more training data can be used (as in [16]). However, as the bar labelled *trained offline* in Figure 7 shows, not including the incident data points results in a drastic drop in accuracy from 90% to 15% on the resource contention scenarios (§ 5.1.2). To aid offline training, we used scenarios with maximum prior incidents (=14). This poor accuracy on debugging incidents not seen before drives our design choice to train Murphy online, every time it's called, on the latest data. Although Sage originally was designed to be trained offline, for fairness, we also train Sage online which yielded higher accuracy for Sage.

6.5.2 Effect of length of training data: The last 3 bars of Figure 7 show Murphy's accuracy with 3 different lengths of training durations on the setup in § 6.3. Murphy's accuracy improves significantly from 87% with 128 points to 95% with 4x more training data. There's a tradeoff between using longer training data and the running time since the training happens online in Murphy, when the tool is run by an operator. We found using the prior one week of historical data to be a good tradeoff point in our production environment (§ 6.2). The application characteristics like topology, configs, application version, workload, etc., also change from time to time, which is another reason to train only on recent metrics.

Scheme	Missing values	Missing edge	Missing entity	Missing metric	Aggregate (avg(1-4))	Unchanged input
Murphy	0.84	0.75	0.78	0.81	0.80	0.86
Sage	0.64	0.67	0.66	0.82	0.70	0.80
NetMedic	0.16	0.20	0.16	0.22	0.18	0.22
ExplainIT	0.05	0	0	0	0.01	0

Table 2: Robustness: Accuracy with degraded/incomplete data. Numbers show recall in top-5. ExplainIT was designed for a different use case – as an interactive tool for queries posed by the user, hence its accuracy was low for automated diagnosis. Both Sage and Murphy were fairly robust.



(a) Error in metric prediction for 17K entities

(b) Multiple iterations of Gibbs sampling improves accuracy

Figure 8: (a) Errors in predicting the metrics of an entity, given the metrics of all its neighbors across 17K entities spanning 300 production apps. (b) Gibbs sampling improved accuracy of prediction across multiple hops, consistent with the existence of cyclic effects in the production environment.

6.5.3 Accuracy with no prior incidents: Often, the root cause for a problematic symptom involves a pattern of metrics that hasn't been seen before. Being able to reason about such scenarios is crucial for Murphy's usability. To test such cases, we generated 64 traces using a similar setup as § 6.3, each with no prior injected incidents. Since Murphy learns the distribution online, the training data still includes data from the current incident which needs to be diagnosed (see § 4.2). Figure 7, in the bar labelled *no prior incidents*, shows that Murphy correctly produces the true root cause 78% of the times in the top-5 and 62% of the times as the top root cause candidate.

6.6 Testing Murphy's internal components

6.6.1 Comparing metric prediction models: Recall that Murphy internally reasons about root causes using a metric prediction model, that predicts how a change in one metric will affect others in the relationship graph. Here we test accuracy of the prediction model, using various methods of learning the model. Testing just this model doesn't require incidents, so we can leverage our much larger metrics dataset (§ 5.1.1).

Figure 8a shows the CDF of absolute error in predicted values across 17K entities from the metrics dataset, when using Ridge, Gaussian Mixture Model (GMM), SVM, and neural networks[10] as the metric prediction model. We found that Ridge linear regression, a variant of robust linear regression, works the best in our production environment. We believe neural networks pose a challenge because the number of available training data points is small (a few hundred).

6.6.2 Verifying existence of cyclic effects: To evaluate whether accounting for cyclic effects in the model improves the training accuracy in our production environment, we design an experiment to measure the effect of changing flow entities' metrics on a backend SQL server that's multiple hops away from the flows. Figure 8b shows that running more than one round of Gibbs sampling in the resampling algorithm (§ 4.2) increases accuracy by 5-10%. Here, we define accuracy as the number of cases where the metrics of the backend SQL server was correctly predicted (see appendix A.2 for more details). Since running more rounds of Gibbs sampling propagates cyclic effects in the relationship graph, this demonstrates that handling cycles correctly boosts the metric prediction accuracy and also confirms the existence of cyclic effects in our enterprise environment.

6.7 Performance and handling scale

Murphy's inference algorithm has a runtime complexity of $O((N+M)T+(N+M)W)$ where N, M are the number of entities and edges respectively in the relationship graph, T is the number of time slices in the monitoring data that the model gets trained on and W is the number of Gibbs sampling iterations. For our incidents dataset, N was typically a few thousands, M was roughly 10-20 times N, T was around 300 and W was 4. Two components contribute to Murphy's running time in our production environment. First, for the incident in Figure 1, it took less than a minute to fetch the metadata and metric values from the database for over 10 thousand entities – the typical size of the relationship graph. Second, Murphy took ~2 minutes on average to produce the root cause entities for a problematic symptom, including the online training time. We can optimize this further with parallelism and by leveraging a C++/Java implementation (as opposed to Python).

6.8 Sensitivity Analysis

Murphy has some parameters, all of which can be tuned offline. We discuss Murphy's sensitivity to these parameters

- Gibbs sampling iterations (W): Figure 8b shows that increasing W led to improvement in accuracy and in general higher W would lead to higher accuracy by giving Gibbs sampling to converge more quickly. A higher W also means a higher running time, hence there's a tradeoff. As Figure 8b shows, the marginal benefit decreases with increasing W and so we settled on $W = 4$.
- Length of training data: We found a slight increase in accuracy with an increase in the length of training data. Refer to the last 3 bars of Figure 7 and § 6.5.2.

7 LIMITATIONS AND FUTURE WORK

While more advanced monitoring can be useful in some cases, e.g. to attribute the high request rate of a VM to a software bug, it's important to have effective diagnosis tools based on commonly available monitoring tools. Murphy's entity-level localization is already useful to operators; we leave diagnosis with advanced monitoring for future work.

Failures outside Murphy's scope: Although Murphy captures many practical cases such as high CPU/drop rate of a VM, high latency of a service etc., many scenarios are not covered, and could be the subject of future work. One such case is where cause and effect are separated in time, or metastable failures that persist even after the trigger is removed [24]. Murphy might not handle nonlinearity in metrics (e.g. if load shedding kicks in after a threshold) since its implementation uses linear regression. Using a different learning model, such as neural networks, for Murphy's MRF framework might resolve this problem. Other examples include: failures that are local to an entity (process or thread), which may require finer-grained telemetry; software errors that don't manifest in any metrics and may be more suitable for program analysis [30]; and scenarios where the root cause is because of an aggregated metric of multiple metrics such as the combined session count of multiple flows.

Using Murphy for performance reasoning: Murphy's counterfactual reasoning framework was useful for performance diagnosis, and may also be applicable to other use cases. For example, it provides a way to evaluate the effect of a config change on a metric: e.g., how would the response latency change if allocated CPUs of the VM is increased by 2x? However, the required level of accuracy for this use case may be different than for performance diagnosis.

Leveraging offline training: While online training works well, a combination of offline + online training could still be leveraged; see § 6.5.1. This is common in the NLP domain [32] that has time-series data similar to ours.

8 CONCLUSION

Performance diagnosis is a persistent challenge for enterprise infrastructure teams. In our work we found that making assumptions about the performance relationships between components or using fixed heuristics for inference harms diagnosis. Instead, we propose using MRFs and learning-based methods that do not make these assumptions for diagnosis in enterprise environments where relationships are rich and complex and evolve across time. We presented a possible design for such an algorithm in Murphy. Murphy not only meets the criteria for diagnosis in our production setup (by virtue of handling complex, cyclic dependencies) but also outperforms current diagnostic tools in their intended environments.

ACKNOWLEDGEMENTS

We thank Ray Belleville, Amarjit Gupta, Abhijit Sharma, Madan Singhal, and other members of the VMware Aria Operations for Networks team for helpful discussions and for their enthusiasm for this project. We thank members of VMware Research Group and the anonymous reviewers of SIGCOMM 2023 for feedback on earlier versions of this paper. We also thank the authors of [16] for providing us their implementation of Sage.

[10] We tried small neural networks up to 3 layers, with 5 neurons each.

REFERENCES

[1] Cadvisor. https://github.com/google/cadvisor.

[2] Jaeger. https://www.jaegertracing.io/.

[3] Murphy: Deathstarbench traces. https://github.com/netarch/Murphy-traces.

[4] Netnorad: Troubleshooting networks via end-to-end probing. https://code.fb.com/networking-traffic/netnorad-troubleshooting-networks\-via-end-to-end-probing/.

[5] One in ten rule of thumb. https://en.wikipedia.org/wiki/One_in_ten_rule.

[6] VMware Aria Operations for Networks. https://www.vmware.com/in/products/aria-operations-for-networks.html.

[7] VMware Aria Operations for Networks - Working with Application Discovery. https://docs.vmware.com/en/VMware-Aria-Operations-for-Networks/SaaS/Using-Operations-for-Networks/GUID-71D10285-1CA6-4F85-A5D8-01B0BEEF3BC2.html.

[8] wrk2. https://github.com/giltene/wrk2.

[9] Zipkin. https://zipkin.io/.

[10] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.

[11] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

[12] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[13] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Xu Chen, Ming Zhang, Z. Morley Mao, and Victor Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, January 2008.

[15] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Saini, George Varghese, and Ravi Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 601–622, 2022.

[16] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, Minlan Yu, and Behnaz Arzani. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 253–269, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[20] Matthew L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–79, 1986.

[21] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[22] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, Seattle, WA, April 2014. USENIX Association.

[23] Herodotos Herodotou, Bolin Ding, Shobana Balakrishnan, Geoff Outhred, and Percy Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1689–1698, New York, NY, USA, 2014. ACM.

[24] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[25] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. Explainit! – a declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 333–348, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in ip networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data*, MineNet '05, pages 173–178, New York, NY, USA, 2005. ACM.

[27] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. *SIGCOMM Comput. Commun. Rev.*, 39(4):243–254, August 2009.

[28] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *Proceedings of the IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 2180–2188, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Stan Z Li. *Markov random field modeling in image analysis*. Springer Science & Business Media, 2009.

[30] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[31] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 255–267, Philadelphia, PA, 2014. USENIX Association.

[32] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 7–14, Florence, Italy, August 2019. Association for Computational Linguistics.

[33] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R. Sambasivan. Identifying Mismatches Between Microservice Testbeds and Industrial Perceptions of Microservices. *Journal of Systems Research*, 2(1), June 2022.

[34] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[35] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, Boston, MA, 2019. USENIX Association.

[36] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 395–420, Boston, MA, February 2019. USENIX Association.

[37] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 383–394, New York, NY, USA, 2014. Association for Computing Machinery.

[38] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. Technical Report MSR-TR-2015-55, June 2015.

[39] Y. Zhang, M. Brady, and S. Smith. Segmentation of brain mr images through a hidden markov random field model and the expectation-maximization algorithm. *IEEE Transactions on Medical Imaging*, 20(1):45–57, 2001.

A APPENDICES

Appendices are supporting material that has not been peer-reviewed.

A.1 Identifying problematic symptoms

A trouble ticket may not directly specify a problematic symptom in the form of an entity metric pair (M_o, E_o). How do we get from a ticket to a problematic symptom? In many cases, operators are able to identify troublesome symptoms in an application e.g. high user response times for a client-facing microservice, high resource utilization of a backend machine, high drop rate at a VM etc.. Operators can then specify problematic (entity, metric) symptoms to Murphy that they want to find reasons for.

Optionally, this step can be skipped and Murphy will find problematic symptoms on its own by scanning the affected application's entities, looking for anomalous metrics in the current time slice using preset thresholds [11]. Alternatively, one could also use other automated tools such as Revelio [15] to identify problematic symptoms that Murphy can diagnose, we leave this for future work.

For each problematic symptom, Murphy separately runs the inference algorithm for performance diagnosis.

A.2 Verifying existence of cyclic effects: supplementary

We wanted to evaluate if accounting for cyclic effects in the model improves the training accuracy in our production environment. Extensive testing in the real world is hard in the absence of the ability to run controlled experiments.

To do so, we design an experiment to measure the effect of flow entities on an entity that's multiple hops away from the flows- a backend SQL server. We picked from the metrics dataset of production apps(§ 5.1.1)- 24 applications that had at least one SQL server. For each application, we chose a randomly chosen backend SQL server Q, and use correlation scores with Q to pick top 5 flows, say flows' F, that sent requests to the front-end of the application. We then test if Murphy can predict the effect of changing metrics of flows in F, on the SQL server Q which is multiple hops away in the relationship graph (see figure 9). More precisely, we take two points in time t_1 and t_2 when Q had significantly different metrics. Keeping the metrics value of all entities other than F to be same as t_1, we update the metrics of the flows F to their values from t_2. We then run Murphy's resampling algorithm (§ 4.2) to obtain predicted metric values of the backend-SQL server- M^Q_{pred}. We measure if the predicted metrics M^Q_{pred} is "close"[12] to the real value M^Q at t_2. Figure 8b shows that running more than one round of Gibbs sampling in the resampling algorithm (§ 4.2) increases the number of scenarios correctly predicted by 5-10%. Since running more round of Gibbs sampling propagates cyclic effects in the relationship graph, this demonstrates that handling cycles correctly boosts the metric prediction accuracy and also confirms the existence of cyclic effects in our production environment.

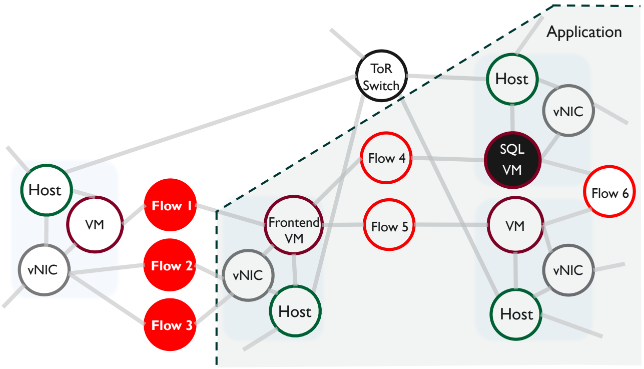


Figure 9: Experimental setup for verifying existence of cyclic effects. We change the values of the three flows in red and measure how well our Gibbs sampling algorithm can predict the corresponding change in SQL VM in black.

[11] conservative thresholds: 25% CPU/memory/disk/port utilization, 0.1% drop rate, 50 TCP sessions or 1GB byte count for a flow in a single time interval. In enterprise environments, operators often configure such thresholds to receive alerts. Those thresholds could be used too.

[12] We define closeness using a criteria that allows for a small additive error and a constant multiplicative error characterized by constants ϵ and Δ, defined as: (Δ, ϵ)-criteria: if the predicted change is δ and the actual change is δ^*, we say that the algorithm is right if either $\delta^*/\Delta < \delta < \Delta.\delta^*$ or $|\delta - \delta^*| < \epsilon.V$, where V is the maximum value of the metric seen so far.