Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows

Darby Huye, *Tufts University, Meta;* Yuri Shkuro, *Meta;*
Raja R. Sambasivan, *Tufts University*

https://www.usenix.org/conference/atc23/presentation/huye

This paper is included in the Proceedings of the
2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the
2023 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows

Darby Huye
Tufts University, Meta

Yuri Shkuro
Meta

Raja R. Sambasivan
Tufts University

Abstract

The microservice architecture is a novel paradigm for building and operating distributed applications in many organizations. This paradigm changes many aspects of how distributed applications are built, managed, and operated in contrast to monolithic applications. It introduces new challenges to solve and requires changing assumptions about previously well-known ones. But, today, the characteristics of large-scale microservice architectures are invisible outside their organizations, depressing opportunities for research. Recent studies provide only partial glimpses and represent only single design points. This paper enriches our understanding of large-scale microservices by characterizing Meta's microservice architecture. It focuses on previously unreported (or underreported) aspects important to developing and researching tools that use the microservice topology or traces of request workflows. We find that the topology is extremely heterogeneous, is in constant flux, and includes software entities that do not cleanly fit in the microservice architecture. Request workflows are highly dynamic, but local properties can be predicted using service and endpoint names. We quantify the impact of obfuscating factors in microservice measurement and conclude with implications for tools and future-work opportunities.

1 Introduction

Microservice architectures are the de-facto method for building distributed systems in large-scale organizations [8, 13]. The basic tenants of this architectural style are well-known—monolithic applications are decomposed into smaller software services that communicate with one another over well-defined APIs, facilitating independence of different development teams, increased deployment velocity, and fine-grained scaling [11, 22]. But, outside of this basic understanding, there is a lack of clarity about industrial microservice architectures' design choices and their resulting characteristics. This ambiguity curtails the impact of microservices research. It is impossible to identify the microservice designs to which improvements suggested in the literature are best suited or which assumptions about microservices' characteristics are valid.

There has been a plethora of research seeking to improve the community's understanding of microservices. Many are qualitative, focusing on reasons for deploying microservice architectures [18, 24, 39], methods for decomposing monolithic applications to microservice architectures with many smaller services [9, 17, 36], and difficulties introduced by microservice architectures [39]. Though useful, they do not provide quantitative data about different organizations' microservice architectures, such as (but not limited to) their scale, topologies, or communication methods, all of which are critical to inform future research.

The community has also created many open-source testbeds built with the microservices design philosophy [1, 13, 46]. But, their scale and complexity do not match that of large-scale organizations' microservice architectures. Past research has shown that these testbeds exhibit much simpler behaviors than industrial architectures [31]. As a result, quantitative data about microservices obtained from these testbeds are not representative of industrial microservice architectures where the microservice architectural style is perhaps most valuable. This is concerning due to the number of research papers that rely on these testbeds [12, 14, 23, 25, 37, 38, 40, 43, 44]. For example, Sage [12] assumes synchronous RPCs. Tprof's [14] layer 4 grouping assumes non-combinatorial explosion when grouping requests by visited services' execution order. Both assumptions are invalid at Meta.

Recent publications from other large cloud companies provide quantitative data about their microservice architectures [20, 41]. But, they represent only partial views of single design points. Additional quantitative studies—both confirming existing findings and focusing on unexplored dimensions— are needed to enrich the community's understanding of large-scale microservices. We envision that these studies will collectively inform robust assumptions for use in microservice research and development.

We present a top-down analysis of Meta's microservice architecture, starting from its service-level topology and descending into individual request workflows. (Request workflows describe the order and timing of services visited by requests when executing.) Our focus is on underreported characteristics of microservice architectures important for developing microservice tools and artificially modeling microservice topologies. Specifically, we describe growth and churn of the microservice topology (to inform tools that learn models of the topology [12, 25, 44]), whether elements of the topology fit power-law distributions common to large graphs (to inform potential artificial topology generators), and the predictability of individual request workflows (to inform the vast number of tools that work by aggregating trace data [14, 29, 45]). We report on characteristics described in previous studies, such as workflows' sizes and shape, to enable qualitative comparisons.

We perform our study using production datasets[1] describing Meta's microservice topology and request workflows within it.

[1] https://github.com/facebookresearch/distributed_traces

Our datasets for topological analyses span a 22-month period (the entire amount of time historical data about the topology has been maintained). We focus on 1-day of distributed traces [16] (totaling 6.5 million) for our analyses of request workflows, which allows us to focus on predictability of specific request behaviors.

We present our main findings below and conclude this paper with their implications along with a discussion of future research opportunities.

(1) Topological characteristics: The topology is very diverse containing many types of software entities that are deployed as services. The topology is constantly growing, sees daily churn in deployed and deprecated services, and (mostly) does not exhibit power-law relationships.

(2) Workflow characteristics: Traces of request workflows vary in size depending on the high-level functionalities they represent. Similar to previous studies [20, 41], we find that traces are small in size and wide in number of communication calls. Service and endpoint names do not predict number of communication calls or their concurrency. But, they reduce uncertainty in the set of services they will call (callers and callees are specified as services + ingress endpoint). Adding knowledge of the children service set better predicts concurrency.

(3) Obfuscating factors preventing quantitative comparisons between architectures: Scale and complexity analyses are hindered because the term "service" is ill-defined for microservices and previous studies do not report their definitions. Different organizations use different tracing platforms with unspecified assumptions about how workflows are sampled and what sampling policies are used. We find that these factors have non-negligible effects on our results.

2 Toward characterizing Meta's microservices

Figure 1 illustrates Meta's microservice architecture. It is similar to other large-scale microservice architectures [11, 20, 22, 31, 41], consisting of ① (in Figure 1) a topology of interconnected, replicated software services running in dozens of datacenters; ② load balancers for distributing requests amongst service replicas; ③ an observability framework for monitoring the topology and creating traces (graphs) of a sampled set of request workflows; and ④ a globally-federated scheduler for running services on host machines within containers. A basic assumption of Meta's architecture (which may or may not be true for other organizations' architectures) is that *business use case* is a sufficient partitioning by which to define services, scale functionality, and observe behaviors.

The rest of this section motivates the value of studying the topology and request workflows, discusses limitations of previous studies, and fills in important details about Meta's architecture relevant to our analyses. We conclude by discussing the observability framework and the datasets generated from it that we use in our analyses. Given the sparsity of information

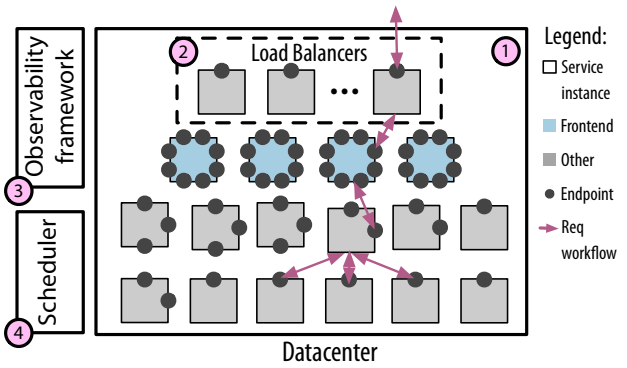


Figure 1: **Meta's microservice architecture and an example request workflow.** The architecture consists of many service instances distributed across many datacenters.

available about Meta's microservice architecture, we err on the side of providing more information than strictly needed.

How do applications use Meta's microservice architecture? Customer applications, such as Instagram or Facebook mobile, issue requests that are load balanced by DNS to specific datacenters and processed by a subset of the architecture's software services. Example requests include those to save photos or record reactions to posts. Applications internal to Meta, such as dashboard or internal tools, use the architecture similarly. But, their requests are load balanced via internal mechanisms, not DNS.

2.1 Topology: services & communication

Why study microservice topologies? We need to understand their complexity, factors that influence their complexity, heterogeneity of constituent services, and the speed at which the topology changes. These characteristics are important to inform tools that visualize the topology, learn models based on the topology, or make assumptions about services' homogeneity [12, 14, 25, 44].

Limitations of existing studies: Only Wen et al. [41] focuses on the microservice topology. The scale they report for number of services is based on a sampled dataset of request workflows, which may not reflect the true scale of their architecture. No existing study defines what constitutes a service or how their definition impacts analyses of the topology (e.g., number of services and communication edges). Existing studies do not report on how the topology evolves or the velocity of change [20, 45].

Meta's microservice topology: The topology is formed by many replicated software services (☐ in Figure 1) deployed across dozens of geographically-distributed datacenters along with their communication to process application requests. (Replicas are typically called *instances*). We note that within the topology, the notion of an application is ill-defined. Individual service instances may process work on behalf of multiple applications. They may also issue requests with batched data from multiple applications to other service instances. The topology evolves *organically* with no central coordination

because development teams responsible for services have complete control over how they are built and maintained.

Services: Services are defined as units of software with well-defined API interfaces, called endpoints (● in Figure 1). Each service satisfies a specific *business use case* (e.g., caching a photo feed). There is significant room for interpretation in defining the scope of a business use case. Additionally, software that pre-dates the microservice architecture may serve multiple business use cases, but be deployed as a single service. Both services and endpoints are named by respective services' developers. (We use *Service ID* and service name interchangeably in latter sections to refer to services' names.)

Services can be stateful or stateless [11]. Stateful services, such as databases, persist state for callers whereas stateless ones, such as search frontends, call other services and integrate their results. A variety of programming languages are used to write services depending on fit for the business use case and societal pressures within the organization.

Load-balancing & Communication: Requests to services are load balanced across their instances. Initially, a datacenter load balancer, itself a service, load balances incoming application requests. Afterward, requests between service instances are load balanced by a service-routing library [30] either linked to applications or to outbound sidecar proxies. (Only some services use sidecars, e.g., when their runtimes do not support linking the routing library directly). The routing library periodically communicates with a global service registry to discover services and routes for their instances. Requests can be load balanced to instances within the same datacenter or to instances in different datacenters. Only the datacenter load balancer is depicted in Figure 1.

Most services at Meta use two-way Thrift RPCs [35] for communication, with payloads serialized in Thrift binary format. Many frontend and some backend services also expose numerous HTTP (REST and GraphQL) endpoints; however, they do not have canonical names that we can use for our analyses. For this reason, we limit the endpoint analysis only to Thrift RPCs reported in the dataset from the routing library.

2.2 Individual request workflows

Why study request workflows: We need to understand the dynamic nature of request workflows. Given a single request execution, what will vary in subsequent executions versus what will remain stable? How much of a statement can we make about other request executions after seeing one or a limited number of samples? Such information is important to inform tools that predict performance, extract critical paths, and present aggregate analyses of request workflows.

Limitations of existing studies: Luo et al. [20] present a way to predict the total number of services that will be called at any hierarchical level of a request workflow. But, they do not discuss whether the number, set, or concurrency level of services called by a specific parent can be predicted. Wen et al. [41] present the amount of time children execute concurrently. Zhang, et al. [45] present distributions of the maximum number of concurrent services observed in workflows. But, neither discuss if information in request workflows can predict concurrency or other workflow characteristics.

Request workflows at Meta: Requests from external applications originate at a datacenter load balancer. This load balancer sends requests to instances of *frontend services*, which are entry points for executing request business logic. There are several frontends at Meta serving different subsets of applications and each has many instances. Frontends may call many services, which in turn may call other services. The resulting hierarchy can be described as forming parent/child relationships. Request workflows for requests originating from internal applications are similar, but originate at the first service that executes business logic on behalf of them.

The set of services involved in a request workflow depends on a number of factors including (but not limited to) the business logic that must be executed on behalf of application requests and whether any requested data is cached. On the other hand, the specific set of instances involved in a request workflow depends on the current load and the load-balancing policy in use.

Concurrency & latent work: Within request workflows, parent services may call all or a subset of children services sequentially or concurrently. The former will be the case if parents are blocking (e.g., single threaded so cannot do work while there is an outstanding call). It may also be the case if there are data dependencies between subsequent calls to children, such as an authentication token that must be returned from one service and passed as input to others. The critical path of concurrently-called children services includes only the slowest one, whereas that of sequentially-called ones includes all of them. Children may perform additional, *latent* work after replying to the parent (e.g., for garbage collection or data replication).

Sample request workflow: The arrows (↔) in Figure 1 show a request traversing a single datacenter. The request first arrives to an instance of the datacenter load balancer, which routes it to an instance of a frontend service, such as `www`. The request then traverses deeper into the topology to backend services.

2.3 Observability framework & datasets

Meta's observability framework includes monitoring mechanisms for recording metrics, logging mechanisms for recording various events, and a distributed-tracing infrastructure, Canopy [16], for recording graphs (called traces) of request workflows. Data generated by the framework is retained for a limited time period to reduce storage volume and due to policy. We describe Canopy in more detail below due to its criticality to observability of microservices. We conclude with a description of the log-based and trace-based datasets we use for our analyses.

Canopy for recording request workflows: Canopy works similarly to most existing distributed-tracing infrastructures [27]. It provides APIs that developers use to define

a request workflow and capture important information about the workflow that should be recorded in traces. The former involves modifying services' code to propagate per-request context—e.g., request IDs and happens-before relationships—within and among the services involved in request execution. The latter involves adding *trace points*, similar to log messages, within source code. During runtime, records of trace points executed by requests are annotated with request context and timestamps. Off of the critical path of request execution, records with identical trace IDs are ordered by happens-before relationships to create traces.

Under the hood, Canopy's implementation is similar to *event-based* tracing infrastructures [10, 26, 29]. However, the way developers instrument services and use the resulting traces is similar to *span-based* tracing infrastructures [4, 32, 34]. *Implementation*: *(1)* Trace points are single events. Higher-level blocks demarcating various intervals (e.g., service executions, queuing time, or function executions) are constructed via annotations added to them. *(2)* context is propagated on both request (forward) and response (reverse) paths, allowing points to be ordered globally within and across services. *Usage*: *(1)* developers (typically) only add blocks denoting service executions; *(2)* happens-before relationships are only established in the forward direction of context propagation, meaning they identify parent/child relationships between blocks and not ordering between siblings; (3) causality between sibling blocks is not explicitly captured via alternate mechanisms. It is impossible to tell whether siblings that execute sequentially as per timestamps in one trace must execute sequentially in other traces.

We describe aspects of Canopy relevant to our workflow analyses. A key observation is that traces created with Canopy may—*by design*—not capture all of a request's workflow.

Effective trace model: Traces are graphs. Nodes are blocks (spans) indicating service execution and hierarchical levels indicate parent/child relationships. Blocks include trace points indicating message send and receives. They may contain additional points indicating other events of interest. Edges between points represent network communication. Latent work started on behalf of a request after the response is returned to the client, such as data replication or asynchronous notifications, may be recorded as additional points on the service block, or as a separate trace with a link back to the originating request's trace (similar to OpenTelemetry's *span links* [5]). Service blocks automatically record *Service ID*s and endpoint names for communication using Thrift RPCs [35]. Developers must manually provide names for services that use custom communication methods. Figure 2 shows an example Canopy trace originating at the *www* service. It has two children services. One of the children (Service B) also has a child (DB).

Streaming model for trace creation with timeout: A stream-processing framework [21] is used to construct traces from trace-point records for subsequent post-processing, such as computing critical path or generating end-to-end latency metrics. The framework accumulates trace events using a

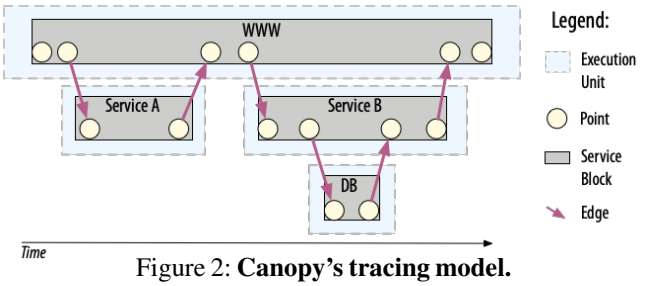


Figure 2: **Canopy's tracing model.**

session window with a fixed gap of inactivity [6]. Traces whose events have a gap in the arrival time larger than the session window would be accumulated in more than one session, but only the first one would be used to trigger post-processing, which may result in processing of partial traces.

Per-service sampling profiles (policies) with rate limiting: Sampling profiles are unique to Canopy. They can be attached to any service and indicate sampling policies to apply based on specific attributes of incoming requests. Traces reflect the union of all sampling profiles that their corresponding requests encounter while executing. This means that a request's trace may start at a service deep in the topology, not recording prior services executed by the request. Trace branches may end prematurely at services whose profiles chose to stop recording the rest of the request's workflow.

A policy specifies: *(1)* a set of conditions for when it is applicable, such as group of endpoints, *(2)* a sampling method, *(3)* a maximum rate of trace data, measured over a sliding time window, beyond which additional traces will not be captured, and *(4)* a verbosity level to decide which instrumentation to execute for requests. Sampling methods may be random head-based sampling [34], in which requests are traced with a random probability, or adaptive sampling [33], in which the sampling probability is periodically changed to achieve a target rate of trace throughput.

Inferred service blocks: These blocks represent services that prematurely ended trace branches, either because of rate limiting or because they lacked tracing instrumentation. Inferred blocks are created during trace construction using information in parent services' message-send points. Inferred blocks may be named or unnamed. The former will be the case when parent points contain the necessary naming information.

Datasets used for this paper: Table 1 shows the datasets we use. For our topological analyses, we use logs describing service activity: history of deployments and deprecation, endpoints exposed by deployed services, and calls made from/to deployed services. For the workflow analyses, we use distributed traces collected by Canopy. The log data describes every deployed service, whereas traces are sampled using methods unique to Canopy, described above.

3 Topological Characteristics

We characterize Meta's current microservice topology as well as how it has evolved. Our analyses of the current topology uses the last (most recent) day of the *Service History* and

Dataset	Description	Format	Retention	Size	Period Used
Service History	Service deployment, lifetimes & interservice communication	*Service ID*s deployed each day	22 months	17 MB	All
Service Endpoints	Endpoints exposed by services	*Service ID* endpoints accessed each day	30 days	18.8 MB	1 day
Traces	Distributed traces	Canopy Trace Objects	30 days	13.1 PB	1 day (4.6 TB)

Table 1: **Datasets used for analyses.**

Endpoints datasets (2022/12/21). Our historical analyses use all 22-months of data available to us (2021/03/01 to 2022/12/21). We also use various dashboards w/statistics about services. The main findings are summarized below.

Finding F1 (All subsections): Meta's microservice topology contains three types of software entities that communicate within and amongst one another: *(1)* Those that represent a single, well-scoped business use case. *(2)* Those that serve many different business cases, but which are deployed as a single service (often from a single binary); *(3)* Those that are ill-suited to the microservice architecture's expectations that business use case is a sufficient partitioning on which to base scheduling, scaling, and routing decisions and to provide observability. These latter entities use *Service ID*s in custom ways, obfuscating their true complexity.

Finding F2 (§3.2): The topology is very complex in its current state, containing over 12 million service instances and over 180,000 communication edges between services. Individual services are mostly simple, exposing just a few endpoints, but some are very complex, exposing 1000s or more endpoints. The overall topology of connected services does not exhibit a power-law relationship typical of many large-scale networks. However, the number of endpoints services expose does show a power-law relationship.

Finding F3 (§3.3): The topology has scaled rapidly, doubling in number of instances over the past 22 months. The rate of increase is driven by an increase in number of services (i.e., new functionality) rather than increased replication of existing ones (i.e., additional instances). The topology sees daily fluctuations due to service creations and deprecations.

3.1 Existence of ill-fitting software entities

We discovered several anomalous patterns in the structure of service IDs within both datasets. For example, we found that on average, 60% of services observed on any single day of the 22-month period have *Service ID*s of the form `inference_platform/model_type_{random_number}`. We found that these services all expose a small number of endpoints with identical names. Meta's engineers informed us that these *Service ID*s are generated by a general-purpose platform for hosting per-tenant machine-learning models (called the Inference Platform). The platform serves a single business use case—i.e., serving ML models—but many per-tenant use cases. Platform engineers chose to deploy each tenant's model under a separate *Service ID* so that each can be deployed and scaled independently per the tenant's requirements by the scheduler.

Following our discovery of the Inference Platform, we investigated the most frequent *Service ID*s and those with the greatest number of service instances. We found two types of software entities that use *Service ID*s in custom ways: *(1)* platforms, such as the Inference Platform, for which multi-tenancy is an additional dimension that must be considered for scheduling, scaling, routing, or observability; *(2)* storage systems, which must take into account data placement in addition to their business use case(s).

We found that some entities, such as the Inference Platform, appear as many services where each service is a combination of the business use case and the additional dimension(s) of partitioning required. Other entities, such as databases and other platforms, appear as a single service and provide their own scheduling and observability mechanisms. Both types of entities' unique use of *Service ID*s masks their true complexity and skews service- and endpoint-based analyses of microservice topologies (ours and likely previous studies [20, 41]).

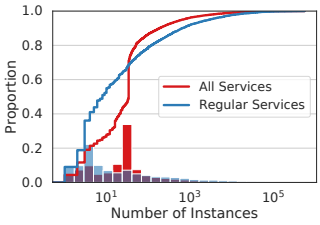
There is no systematic way to identify these ill-fitting software entities at Meta. To illustrate how they may affect *Service ID*-based analyses, we call out contributions by two entities that affect our results significantly. The first is the Inference Platform, which inflates the number of services observed. The second is the ML Scheduler, a scheduling platform for ML training jobs which chooses to appear as a single service and so inflates instance counts. We collectively refer to them and their services as *Ill-fitting services* and all other services as *Regular services*.

3.2 Analysis of the current topology

Scale is measured in millions of instances: On 2022/12/21, the microservice topology contained 18,500 active services and over 12 million service instances. Excluding the ill-fitting services, there are 7,400 services and 11.2 million instances.

The instance count is due to a few highly-replicated services: Figure 3 shows that the ill-fitting services greatly skew instance counts. Notably, the `ML scheduler` is replicated over 270,000 times, 2.2% of all instances. When these services are excluded, the median service's replication factor is only eight and the 99th percentile is 31,306. Frontend service `www` is the most replicated service (557,000 instances, 4.6% of all instances) as it handles most incoming requests.

Services are sparsely interconnected: We construct the service dependency diagram by connecting services that communicate with each other at least once with an edge. (Our dependency diagram is similar to that constructed by OpenTelemetry or Jaeger, except that it is constructed from a portion of the *Service History* dataset that captures commu-

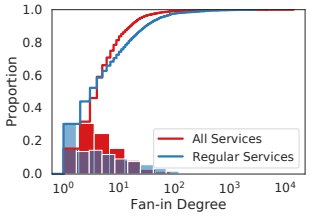


(a) CDF

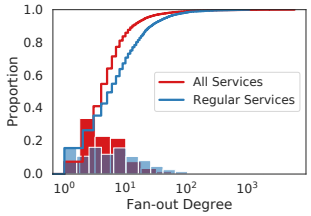
	# Instances	# Instances (Regular)
Min	1	1
Med	34	8
Avg	668	1,504
P99	9,220	31,306
Max	557,156	557,156

(b) Statistics

Figure 3: ***Service ID* replication factors.** The histogram is shown under the CDF. When the curves overlap, the colors are blended together.



(a) Fan-in CDF



(b) Fan-out CDF

Statistic	Fan-in	Fan-out	Fan-in (Reg)	Fan-out (Reg)
Min	1	1	1	1
Median	4	4	3	6
Average	14	12	19	15
P99	86	101	324	158
Max	14,084	5,865	2,968	1,069

(c) Statistics

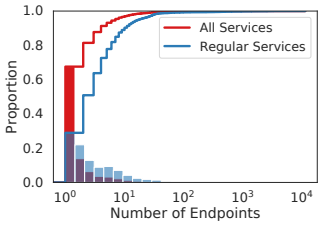
Figure 4: **Service fan-in and fan-out.**

nication between services, not raw traces.) There are 393,622 edges that connect services, which is much smaller than a fully connected topology (18,500^2 or 342 million edges).

Services are called by services more than they call other services: Continuing with the dependency diagram, Figure 4 shows CDFs and statistics about services fan-in (# of services that call them) and fan-out (# of services that they call) degrees. The median fan-in and fan-out are the same, but average and maximum fan-in is larger than fan-outs (14 vs 12 and 14,084 vs 5,865). Excluding the ill-fitting services, by removing all edges connected to ill-fitting services, decreases the median fan-in but increases the median fan-out. Excluding the ill-fitting services also increases the 99.9 percentile and decreases the maximum fan-in and fan-out values.

We investigated the services that have the highest fan-in and fan-out degrees. The former is a vault server storing credentials for use by other services. The latter is a service for querying hosts for arbitrary statistics. Both are used heavily by ill-fitting services, constituting 78% and 91% of the vault service's callers and the services called by the stats service respectively. When ill-fitting services are excluded, two other services rise to the highest fan-in and fan-out degrees respectively: a generic counting service used for various rate limiting mechanisms and a frontend service for internal applications.

Most services are simple, exposing only a few endpoints:



(a) CDF

	# Endpts	# Endpts (Regular)
Min	1	1
Med	1	2
Avg	5	14
P99	26	61
Max	11,359	11,359

(b) Statistics

Figure 5: **Number of endpoints exposed by services.**

Figure 5 shows a CDF and statistics of services' endpoints. Most services do not expose many endpoints (median: 1, P99: 26) and excluding ill-fitting services does not shift the statistics much. The service that exposes 11,359 endpoints is `www`, a frontend service which is used for many business use cases. It is deployed as a single binary from a large, well-engineered codebase that predates the microservice architecture.

Service complexity follows a power-law distribution: Service complexity, measured by number of unique endpoints in a service, follows a power law distribution ($\alpha = 2.23$, $R^2 = 0.99$), indicating that most services are simple with a long tail of more complex services. The power law does not hold for other measures of complexity. Despite there being a long tail of more complex services, the service dependency diagram does not follow a power law distribution ($R^2 = 0.62$). This means the services with more endpoints are not proportionally more connected to the topology than services with fewer endpoints. While there are some highly replicated services, the overall trend of instance counts does not follow a power law distribution either ($R^2 = 0.25$).

Sixteen different languages used to write services: Services can be written in many programming languages. There are currently 16 different programming languages in use at Meta, with the most popular being Hack (a version of PHP), measured by lines of code. Other popular languages include: C++, Python, and Java, with the rest forming a long tail.

3.3 Past growth & dynamism

The number of deployed service instances has almost doubled over the past 22 months: Figure 6 shows the percentage of deployed service instances each day as a percentage of the maximum value observed on 2022/12/21. We show different series for when all services are included, just the ill-fitting services, and only regular services. The slope when all services are considered is $s = 0.052\%$ per day (linear regression $R^2 = 0.95$). The slope when ill-fitting services are excluded is $s = 0.046\%$ per day ($R^2 = 0.95$).

The steady increase in instance counts reflects either an increase in hardware capacity over the time period or an increase in utilization of existing hardware. It cannot be explained by changes in instance sizes as they have remained mostly static.

Instances' rate of increase is due to new business use cases rather than increased scale: Figure 7 shows unique services deployed each day as a fraction of the maximum value

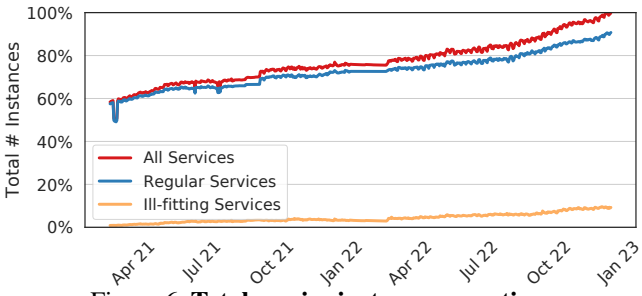


Figure 6: **Total service instances over time.**

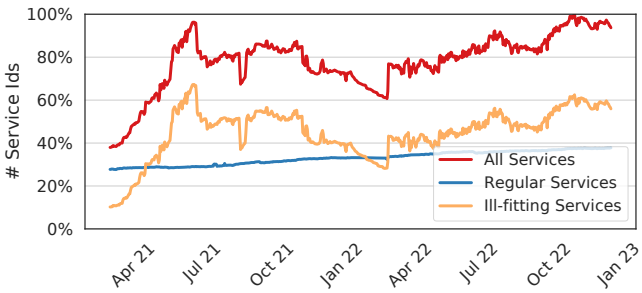


Figure 7: ***Service ID*s over time.**

observed on 2022/11/03. Note that the day with the maximum number of *Service ID*s is different from the day with the most instances. Almost all variability is explained by the ill-fitting services, specifically the Inference Platform, which launches and terminates services as per tenants' demands. The daily increase in services when ill-fitting services are excluded (slope of *Regular Services* series) is $s = 0.043\%$ ($R^2 = 0.98$). It is almost identical to the daily increase in instance counts when ill-fitting services are excluded, which was $s = 0.046\%$ (slope of the *Regular Services* series in Figure 6).

Lots of churn in services, with both long-lived and ephemeral ones: Over the 22-month time period, 180,000 new *Service ID*s were deployed, 89.7% of which were deprecated at some point. Figure 8 shows the number of services created and deprecated each day. Newly-created services are ones whose *Service ID*s were not observed previously during the 22-month period, whereas deprecated ones are services whose *Service ID*s are never observed again. For regular services, creation rates are slightly higher than deprecation rates. As expected, ill-fitting services have high churn.

We also computed the percentage of services observed over the entire period that were deprecated in less than one week (54% of ill-fitting Services, 7.7% of regular services) and the percentage that existed throughout the 22-month period (0% of ill-fitting services, 40% of regular services).

4 Request-workflow characteristics

We now analyze service-level properties of individual request workflows using traces collected by different profiles. We first discuss traces' general characteristics, such as size and width (§4.2). We then analyze whether specific elements of a single trace predict properties of other traces representing the same high-level behavior(s) (§4.3-§4.4). As with any large-scale

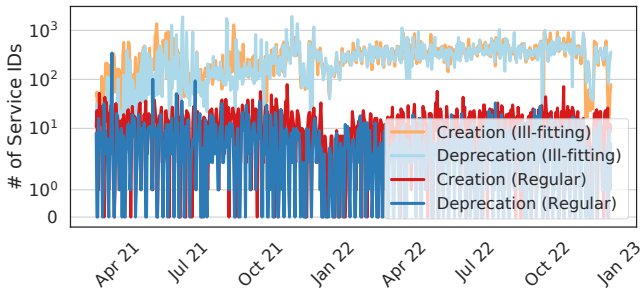


Figure 8: ***Service ID* creation & deprecation.**

tracing infrastructure, traces' visibility into request workflows may be limited due to dropped records, rate limiting, and non-instrumented services. We quantify the extent to which visibility into traces is obscured as a result of these factors in §4.5.

Methodology: For all experiments, we use traces collected on 2022/12/21 from three profiles monitoring important high-level business functionalities. Using a few profiles and a single day avoids factors that would otherwise obscure the interpretability of our results: the effects of analyzing traces using many sampling policies and code updates that change service behaviors. Focusing on important profiles increases the likelihood that traces are representative of their workflows: the services they access are likely to propagate context accurately and use descriptive *Service ID*s and endpoint names. Overall, we analyze 6.5 million traces representing 0.5% of traces collected on 2022/12/21 by all Canopy profiles. Though we do not report them, we observed similar trends to our results on different neighboring days to 2022/12/21 while refining our experiments.

For our predictability experiments, we conduct an ex-post-facto analysis of whether *Ingress ID*s, defined as a combination of *Service ID* and ingress endpoint name, predict properties of their children across many traces. We choose to use *Ingress ID*s because they are readily available in traces, are usually the primary means of understanding trace behaviors, and are location-independent so do not require alignment of traces starting at different (unknown) depths in the topology. We do not consider global characteristics of traces, such as size or width, for prediction experiments as they are not guaranteed to be comparable due to rate limiting or dropped trace records. In our predictability sections, we mean *Ingress ID*s when we refer to parents and children, as in "unique children."

We omit inferred calls (§2.3) from experiments that consider service names since names of inferred services are often unknown. Also, we omit *Ingress ID*s found fewer than 30 times within a profile to allow meaningful statistics to be calculated for the rest of the endpoints.

Our main findings are summarized below; we introduce the profiles afterward. Figure 9 describes the trace properties we analyze and predict.

Finding F4 (§4.2): We measure traces with regard to the number of service blocks they contain (recall from § 2.3 that a service block represents the time interval a service was executed; repeated invocations of the same service appear

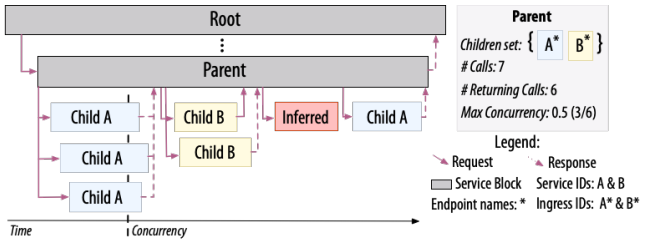


Figure 9: **Trace Characteristics.** Generic example trace showing attributes for a parent *Ingress ID*. Root service is either *www* or *RaaS*. *Inferred* services are represented as blocks of a fixed length since they do not contain notions of time or return edges. They are omitted from concurrency calculations.

as multiple service blocks.) Trace sizes vary depending on workflows' high-level behaviors, but most are small (containing only a few service blocks). Traces are generally wide (services call many other services), and shallow in depth (length of caller/callee branches).

Finding F5 (§4.3-§4.4): Root *Ingress ID*s do not predict trace properties. At the level of parent/child relationships, parents' *Ingress ID*s are predictive of the set of children *Ingress ID*s the parent will call in at least 50% of executions. But, it is not very predictive of parents' total number of RPC calls or concurrency among RPC calls. Adding children sets' *Ingress ID*s to parent *Ingress ID*s more accurately predicts concurrency of RPC calls.

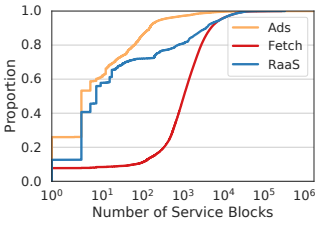
Finding F6 (§4.5): We observe that many call paths in the traces are prematurely terminated due to rate limiting, dropped records, or non-instrumented services. Few of these call paths can be reconstructed (those known to terminate at databases) while the majority are unrecoverable. Deeper call paths are disproportionately terminated.

4.1 Profile details

Ads: This profile represents a traditional CRUD web application focusing on managing customers' advertisements, such as getting all advertisements belonging to a customer or updating ad campaign parameters. The profile captures traces from 56-related endpoints exposed by the `www` frontend service. There are 3.2 million traces over the single-day period. This profile's sampling policy is random at 0.01% capped at 65 traces per second or 160 MB of trace data per minute.

Fetch: This profile represents deferred (asynchronous) work triggered by opening the notifications tab in Meta's client applications. Examples of work include updating the total tab badge count or retrieving the set of notifications shown on the first page of the tab. It captures traces from 91-related endpoints exposed by the `www` frontend service. There are 87,000 traces over the one-day period. This profile uses adaptive sampling with a target rate of 1 trace per second, capped at 20 MB of data per minute.

RaaS (Ranking-as-a-Service): This profile represents ranking of items, such as posts in a user's feed. The RaaS sampling policy is applied to the `RaaS` service, a non-frontend service

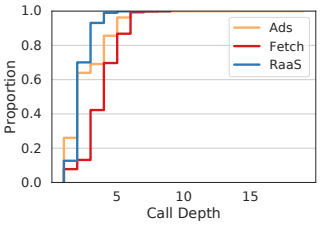


(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	498	4
Average	83	1,052	628
P99	1,233	8,911	10,667
Max	253,877	81,859	113,790

(b) Statistics

Figure 10: **Trace Size.** Service block counts per trace.



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	4	2
Average	2.6	3.8	2.3
P99	6	6	5
Max	19	9	9

(b) Statistics

Figure 11: **Call Depth.** Max depth of service blocks per trace.

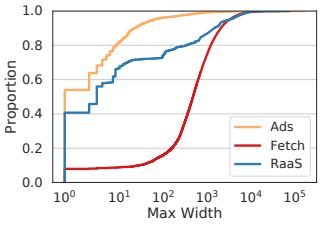
that is called by other services. As a result, traces from this profile always represent only portions of request workflows. Of the workflows we analyze, only those captured by Fetch call `RaaS`. Occasionally, a RaaS trace will be a portion of a Fetch trace, but such occurrences are rare because both Fetch and RaaS profiles use low sampling probabilities that are independent of each other. There are 3.3 million traces over the single-day period, from 4 different endpoints in `RaaS`. This profile uses adaptive sampling with a target rate of 25 traces per second.

4.2 General trace characteristics

There is significant diversity in trace sizes: Figure 10 shows CDFs and statistics of the number of service blocks in our traces. Traces collected by the Fetch profile are significantly larger than Ads and RaaS except at the tail, where Ads traces are largest.

Traces are shallow and wide: Figure 11 shows the maximum call depth in service blocks of our traces starting from trace roots (root is call depth 1). Figure 12 shows maximum trace width, which is the maximum number of calls made by all services at any depth level. (For example, 3 service blocks at one depth making 3 calls each results in a width of 9). We see that across all profiles, traces are much wider than deep: in Fetch profile the median depth is 4 vs. median width of 472, and P99 depth is 6 vs. P99 width of 7,400. We conclude that large traces are a result of the number of calls made by services, not depth of calls. This can be partially explained by the widespread use of data sharding where retrieving a collection of items requires fanning out requests across many storage service instances.

Service reuse within traces is high and occurs at many different call depths: Figure 13 shows CDFs and statistics of the number of services visited within individual traces. Comparing with trace sizes (Figure 10), traces generally contain more service blocks than unique services. At the median, traces visit be-

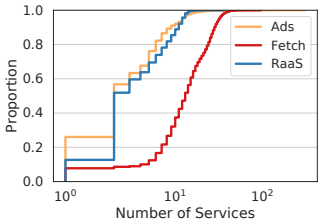


(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	1	472	3
Average	52	916	507
P99	774	7,395	7,885
Max	166,171	46,746	72,196

(b) Statistics

Figure 12: **Max Width.** Maximum number of service calls at a single call depth within a trace.



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	13	2
Average	4.1	14.6	4.5
P99	18	39	14
Max	307	96	36

(b) Statistics

Figure 13: **Unique Services.** Unique *Service ID*s in a trace.

Type	Ads	Fetch	RaaS
All	421	127	72
Leaf	168 (39.9%)	63 (49.6%)	35 (48.6%)
Single relay	58 (13.8%)	21 (16.5%)	17 (23.6%)
Variable relay	195 (46.3%)	43 (33.9%)	20 (27.8%)

Table 2: **Parent types.** The distribution of parents of each type within each profile.

tween 1x (2/2), 38x (498/13), and 2x (4/2) more service blocks than unique services across Ads, Fetch, and RaaS respectively; at P99 these ratios are 71x, 21x, and 810x respectively.

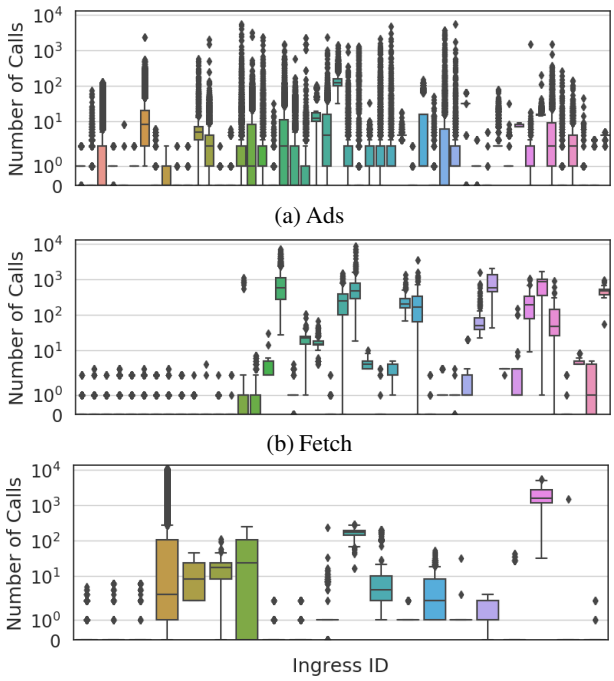
Most services are observed at more than one call depth in our profiles. We measured the number of call depths at which each service was observed. The services in Ads traces have high rates of appearing at multiple depths (median: 6, average: 7.3). Approximately 60% of Fetch and RaaS services are observed at multiple levels (median: 2, average: 2.6 for both profiles).

4.3 Predicting parent/child relationships

Parent *Ingress ID*s strongly predict whether services will have no children or only one child: Table 2 shows that such services, defined as *leaves* and *single relays*, make up from 53 to 73 percent of service executions in our profiles. We find that they are always databases or calls to databases.

***Ingress ID*s do not predict number of downstream calls**: Parents that make one or more downstream calls to children are called *variable relays* in Table 2, making up from 27 to 47 percent of *Ingress ID*s in our profiles. Figure 14 shows that variable relays exhibit a wide distribution in the number of children calls they make. Some *Ingress ID*s exhibit high variance in the number of children they call across different executions whereas others have very little variance (but it is always non-zero).

Variability in number of children calls is due to database calls for Fetch and RaaS: We find that at least 61.1% and



(a) Ads

(b) Fetch

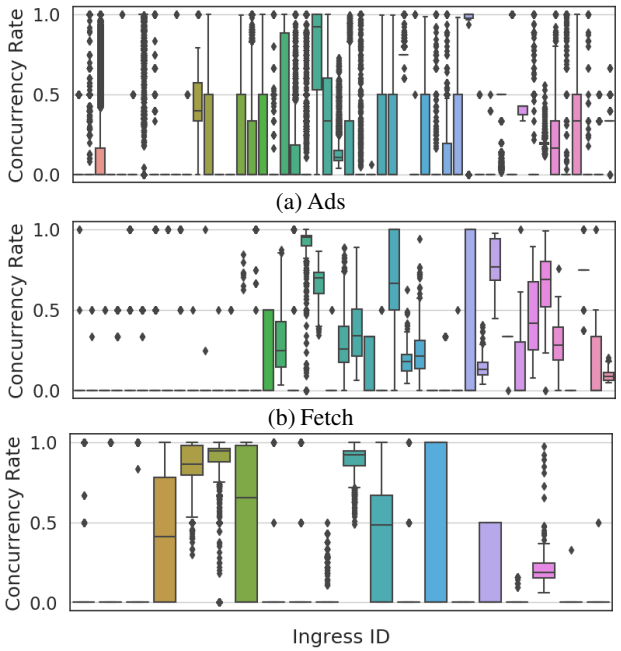
(c) RaaS

Figure 14: **Calls per parent.** Boxplots are shown for every Fetch and RaaS variable relay. Due to limited space, only the 50 variable relays with the greatest number of invocations are shown for Ads. Parent *Ingress ID*s are sorted in descending order by total number of invocations. Boxplot boundaries indicate P25-P75 and the horizontal line within boxes indicate medians. Lower and upper whiskers indicate the smallest/largest data values within 1.5 IQR below/above P25/P75 and dots are outliers.

72.1% of these variable relays' children calls are database accesses in Fetch and RaaS traces respectively. For Ads traces, only 35.7% of children calls are database accesses.

There is a dominant set of unique children per parent: When we ignore number of calls, we find that most single and variable relay parents call only a few *children sets*, where each set is defined as a combination of unique children *Ingress ID*s within a given invocation of the parent *Ingress ID*. For example, one children set may contain `memcache+read` and `database+write`, whereas another may contain `key_service+retrieve` and `database+write`. The average number of children sets called by a relay parent is 28 for Fetch & Ads and 12 for RaaS parents. Most parents have a *dominant children set* that they call in more than 50% of executions. Specifically, 71.9%, 80.2%, and 81.6% of Fetch, Ads, and RaaS relays have dominant children sets.

Non-dominant children sets contain mainly one off children *Ingress ID*s and are not a superset of the parent's dominant children set. On average, only 27% of children *Ingress ID*s called by a parent are in most (>50%) of the parent's children sets.



(a) Ads
(b) Fetch
(c) RaaS

Figure 15: **Parent concurrency.** Concurrency distribution for all invocations of a parent *Ingress ID*. Shows all variable relays for Fetch & RaaS, and the top 50 in Ads by invocation count. Boxplots are interpreted identically to figure 14.

4.4 Predicting children's concurrency

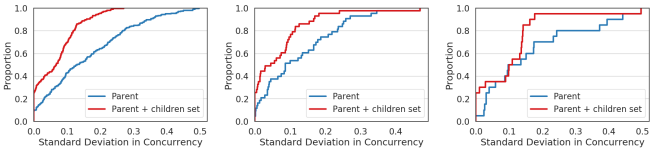
We define *maximum concurrency* as the maximum number of children calls executing concurrently by a parent at any point in time in its execution. More formally, a set of concurrent calls S_t at time t is all children calls with $t_{start} \leq t < t_{end}$, where all timestamps are measured at the parent, and the maximum concurrency C is computed as:

$$C = max(|S_t|), \forall t : t_{start}^{parent} \leq t < t_{end}^{parent} \quad (1)$$

We use a normalized measure of maximum concurrency, the *concurrency rate*, calculated as $C/num\_children$, to allow comparisons across different executions of the same parent (different numbers of children may be called in each execution) and to allow comparisons between different parents. *num_children* refers to children that have return edges and well-defined durations. We only consider variable relays since concurrency is ill-defined for leaves and single relays.

Parent *Ingress ID* does not predict whether children will execute concurrently or sequentially: Figure 15 shows boxplots of concurrency rates across executions for each parent *Ingress ID* observed in our traces. We see that there is a mix of high and low variation in concurrency rate across *Ingress ID*s.

The combination of parent *Ingress ID* and children set more accurately predicts concurrency rate: Figure 16 shows a CDF of the standard deviation in concurrency rate across all executions of parent *Ingress ID*s. To understand if children set adds predictability value, we calculate the standard deviation for each parent's children set and average them to obtain a



(a) Ads (b) Fetch (c) RaaS

Figure 16: **Standard deviation in concurrency rate.** *Parent* shows a CDF the standard deviation in concurrency rates for all executions of a parent. *Per-parent avg. children set* shows the average standard deviation per children set for each parent.

per-parent average. We plot this CDF of per-parent average. Intuitively, if children sets provide value, the per-parent average should decrease whereas if they do not, the data points will be randomly distributed and standard deviation will not decrease. Overall, we find that including children sets shifts the distributions to the left. The shift is most pronounced at the median for Ads and Fetch: 0.13, 0.09 vs. 0.04 and 0.02. Adding children set does not provide value in the tail for Fetch and RaaS.

We speculate the reduction in standard deviation is because children belonging to the same children set likely have well-defined control or data dependencies between each other. Reduction in variation due to control dependencies may be a result of custom threading models for different code logic blocks in parents (each responsible for a different behavior and thus children set). For data dependencies, consider the following examples. Children sets containing different cache services may have no dependencies and thus may be able to execute concurrently. In contrast, children sets comprised of a key server and a database service may have to execute sequentially: credentials may be required to access the database.

***Ingress ID* + children set calls display a range of dependency relationships**: We now quantify the strength of dependencies within *Ingress ID* + children set's calls. We use the maximum concurrency rate observed across *Ingress ID* + children set executions as a indicator of dependency. A maximum concurrency rate of 1 implies that there are no dependencies among children calls. A maximum observed concurrency rate of 0 builds confidence that the children calls are dependent and must execute sequentially. Figure 17 shows the results. Overall, we find that most *Ingress ID* + children set executions display weak dependencies (some concurrency) and there are a few strongly dependent (sequential) children sets.

4.5 Quantifying traces' observability loss

Most prematurely terminated call paths are unrecoverable: We plot the percent of branches that terminate prematurely (at an *inferred* service) at each call depth in our traces (Figure 18). Some branches terminate prematurely due to internal rate-limiting at databases, which are usually leaves in the traces. The shaded area in Figure 18 is the portion of inferred services that represent known databases. The distance between the curves are unknown inferred services, which make up the majority of inferred services. Using trace data alone, we cannot know what the unknown inferred services are

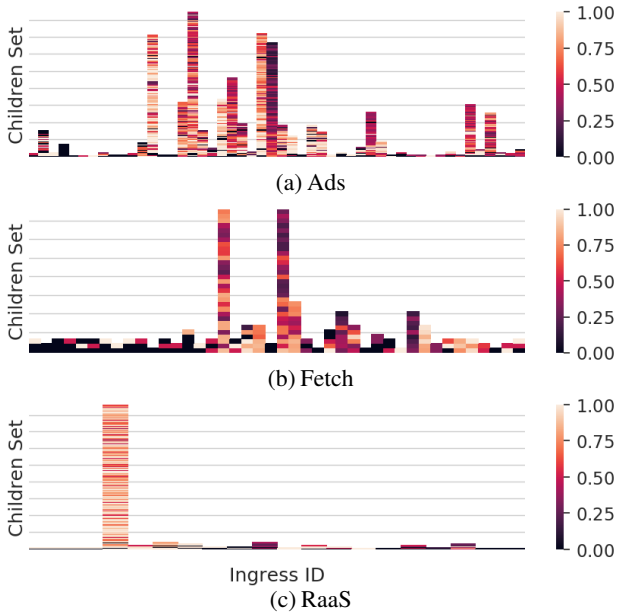


Figure 17: **Parent *Ingress ID* + children set max concurrency rate.**

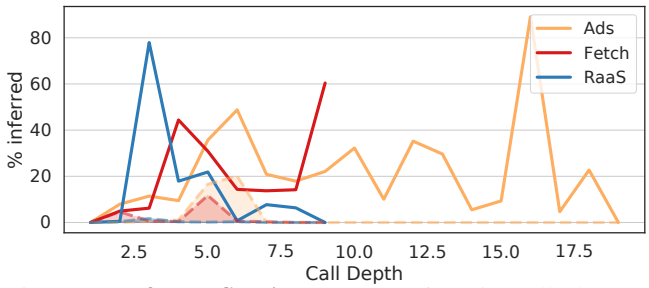


Figure 18: **Inferred Services.** Percent of service calls that are inferred at each call depth. The shaded region is the percent of inferred services that are known to be database calls.

(some may still be other databases we were not able to identify reliably) or the shape of the workflow from that point on.

Non-uniform probability a branch is terminated: Deep branches are disproportionately prematurely terminated. For RaaS traces, 80% of call paths that reach depth 3 are terminated with inferred nodes, none of which were identified as databases. However, as the average trace depth for RaaS is only 2.3 (Figure 11), the majority of RaaS traces are not affected by premature branch terminations. Similarly, Fetch and Ads traces are shallow and prematurely terminated branches mainly occur beyond the average trace depth.

5 Implications and opportunities

Implications for microservice testbeds: Existing testbeds [1, 13, 46] represent only single applications, whereas microservices within Meta serve many applications (§2.1). Previous studies state that existing open-source testbeds' topologies are lacking in scale compared to industrial microservices [20, 41]. Our results confirm these results (Finding F2) and add the following dimensions to consider in future testbeds: heterogeneity of services, churn, and growth. Specifically, we find that Meta's microservice architecture contains a mix of software entities that are deployed as services: complex ones that expose many endpoints and are likely more monolithic in nature, simple ones that expose just a few, and ill-fitting ones that require support beyond which the microservice architecture provides by default (Finding F1). We find that services are deployed and deprecated (at least) daily and that the shape of the communication topology is constantly growing and changing (Finding F3).

Luo et al. [20] state that request workflows within existing testbeds are too static. Many service-level workflow properties can be predicted from root endpoint alone. Our analyses show that future testbeds should include concurrency, number of children, and set of children that are executed as dimensions of variability in request workflows representing the same or similar high-level behaviors (Finding F5).

Implications for microservice tooling: Tools that use models of microservice topologies [12, 25, 44] should assume that its constituent services are always changing and that the topology itself is highly-dynamic (Finding F3). Periodic retraining may be necessary; mechanisms are needed to identify when predictions diverge from the ground truth due to stale topological information.

Tools that aggregate request-workflow traces for performance predictions, diagnosis, or capacity planning [7, 14, 28, 29, 45] must assume that there is significant diversity in workflows originating from the same root endpoints or groups of related root endpoints (Finding F5). Our studies show that many workflow properties can be predicted when they are broken down into fundamental building blocks (parent/child relationships) (Finding F5), perhaps a promising starting point for aggregation-based tools. However, capturing total orderings for entire traces [14] or even individual services may not scale due to parent *Ingress ID*s initiating large number of RPCs with high concurrency (§4.4).

Need for artificial microservice topology & workflow generators: Such generators are a necessity given the infeasibility of creating microservice deployments outside of industrial settings. The sole existing workflow generator [20] may be too specific to a single organizations' microservice design (that number of children depends on depth in trace) and generates stochastic workflows that do not represent any single request. Research is needed to identify: *(1)* which dimensions of microservice architectures are best explored in testbeds versus artificial topology or workflow generators; *(2)* how to ensure these dimensions are representative of a variety of large-scale organizations' characteristics. Our analysis shows that assuming topologies follow power-law relationships is insufficient for modeling microservice topologies (Finding F2).

How to better incorporate ill-fitting software entities into microservice architecture? Ill-fitting entities constitute a significant portion of Meta's microservice topology. Key questions include: *(1)* Should infrastructure platforms provide richer interfaces to allow scheduling, scaling, and observability based on additional dimensions rather than only one? *(2)* In

cases where ill-fitting-entities use custom techniques, what mechanisms are necessary to allow mapping them to standard service-level operations?

Naming & predicting missing elements of workflows: Our predictability results (Finding F5) indicate that well-defined service and endpoint names are important for predicting local workflow properties. Almost all tools that use distributed traces [7, 12, 14, 25, 28, 29, 42, 44, 45] assume descriptive names. But, naming quality can vary considerably, especially for services that satisfy many business use cases and for microservice architectures in which all instrumentation is done within proxies surrounding services [3]. Research into naming schemas that allow different parts of service behaviors to be differentiated based on parts of the name (or attached attributes) is needed. Research is also needed into how to automatically identify meaningful names and/or attributes that differentiate important within-service behaviors, and whether missing observability data (Finding F6) can be predicted based on other data already available.

Need for standardized methods to contrast different organizations' microservice architectures: Our original goal for this research was to compare characteristics of Meta's microservice architecture with previous studies of industrial microservice architectures. At 30,000 ft, we find that organizations' architectures have similar architectural diagrams (Figure 1) and use custom versions of the same architectural components or open-source versions [4, 19, 35]. Furthermore, similar to Meta, the traces used in Luo et al. [20] and Wen et al. [41] tend to be small. Large traces are wider than deep, indicating common use of data sharding. We also find some differences. Traces used in Zhang et al. [45] seem to be much deeper than those used in our analyses, perhaps due to their domain-oriented microservice strategy [15].

Unfortunately, we found more detailed quantitative comparisons to be impossible due to divergent (or ill-specified) definitions in previous studies and because different studies use custom measurement techniques specific to their observability frameworks. With regard to comparing scale and complexity, previous studies do not define the term *service*, describe individual service's complexity, or describe number of communication edges between services, or service instances. For request-workflow-based analyses, these studies do not identify tracing sampling rates and mechanisms, whether traces capture all of request workflows or only parts or whether dropped records or rate limiting impact their analyses. Similar to rich research into Internet measurement [2], we need to develop rich, well-accepted methodologies for collecting data about microservice architectures to understand and systematize similarities and differences across them.

6 Summary

The characteristics of large-scale microservice architectures are largely invisible outside of industrial organizations. We presented an analysis of Meta's microservice architecture to inform more robust assumptions for future microservices research and development.

7 Acknowledgments

We thank our shepherd, Jan Rellermeyer, the anonymous USENIX ATC conference reviewers, Lenni Kuff, Alex Knott, and members of Meta's observability team for their insightful feedback and support. We thank Theo Benson for suggesting examplars on which we could base this paper's structure.

References

[1] BookInfo application. URL: https://istio.io/latest/docs/examples/bookinfo/.

[2] IMC: Internet Measurement Conference. URL: https://dl.acm.org/conference/imc.

[3] ISTIO: Simplify observability, traffic management, security, and policy with the leading service mesh. URL: https://istio.io.

[4] OpenTelemetry. URL: https://opentelemetry.io/.

[5] Tracing in OpenTelemetry. URL: https://opentelemetry.io/docs/concepts/signals/traces/.

[6] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming systems: the what, where, when, and how of large-scale data processing*. O'Reilly Media, Inc., 2018.

[7] Vaastav Anand, Matheus Stolet, Thomas Davidson, Ivan Beschastnikh, Tamara Munzner, and Jonathan Mace. Aggregate-driven trace visualizations for performance debugging. *CoRR*, abs/2010.13681, 2020. URL: https://arxiv.org/abs/2010.13681, arXiv:2010.13681.

[8] Adrian Cockroft. The evolution of microservices, 2016. URL: https://learning.acm.org/techtalks/microservices.

[9] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Migrating towards microservice architectures: an industrial survey. In *ICSA'18: Proceedings of the International Conference on Software Architecture*. IEEE, 2018. doi:10.1109/ICSA.2018.00012.

[10] Rodrigo Fonseca, Michael J. Freedman, and George Porter. Experiences with tracing causality in networked services. In *INM/WREN'10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*, 2010.

[11] Martin Fowler. Microservices: a definition of this new architectural term. URL: https://martinfowler.com/articles/microservices.html.

[12] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[13] 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 & edge systems. In *ASPLOS'19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[14] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[15] Introducing Domain-Oriented Microservice Architecture. URL: https://www.uber.com/blog/microservice-architecture/.

[16] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *SOSP'17: Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[17] Justas Kazanavičius and Dalius Mažeika. Migrating legacy software to microservices architecture. In *eStream'19: Proceedings of the Open Conference of Electrical, Electronic and Information Sciences*. IEEE, 2019. doi:10.1109/eStream.2019.8732170.

[18] Tom Killalea. The hidden dividends of microservices. *Communications of the ACM*, 59(8):42–45, 2016.

[19] Kubernetes. URL: https://kubernetes.io.

[20] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[21] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. In *ICDE'20: Proceedings of the 36th International Conference on Data Engineering*, pages 1591–1602. IEEE, 2020.

[22] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2nd edition, 2021.

[23] HL Phalachandra, Pranav Bhatt, Ishitha Agarwal, and Rishith Bhowmick. Improving task scheduling in microservice environments by considering intra-job dependencies. In *ICICCS'22: Proceedings of the 6th International Conference on Intelligent Computing and Control Systems*, pages 568–573. IEEE, 2022.

[24] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *SCCC'19: Proceedings of the International Conference of the Chilean Computer Science Society*. IEEE, 2019. doi:10.1109/SCCC49216.2019.8966423.

[25] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *OSDI'20: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 805–825, 2020.

[26] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[27] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *SoCC'16: Proceedings of the Seventh Symposium on Cloud Computing*, 2016.

[28] Raja R. Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2466–2475, 2013.

[29] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[30] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *OSDI'23: Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[31] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R Sambasivan. [SoK] Identifying mismatches between microservice testbeds and industrial perceptions of microservices. *Journal of Systems Research*, 2(1), 2022. doi:http://dx.doi.org/10.5070/SR32157839.

[32] Yuri Shkuro. Jaeger: Evolving distributed tracing at Uber Engineering. URL: https://www.uber.com/blog/distributed-tracing/.

[33] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing, 2019.

[34] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.

[35] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 2007. URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[36] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[37] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[38] Mert Toslali, Srinivasan Parthasarathy, Fabio Oliveira, Hai Huang, and Ayse K Coskun. Iter8: Online experimentation in the cloud. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[39] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):1–44, 2021.

[40] Hao Wei, Joaquin Salvachua Rodriguez, and Octavio Nieto-Taladriz Garcia. Deployment management and topology discovery of microservice applications in the multicloud environment. *Journal of Grid Computing*, 19(1):1, 2021. doi:10.1007/s10723-021-09539-1.

[41] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. Characterizing and synthesizing the workflow structure of microservices in ByteDance cloud. *Journal of Software: Evolution and Process*, 34(8):1–18, 2022.

[42] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *ICSE'22: Proceedings of the 44th International Conference on Software Engineering*, 2022. doi:https://doi.org/10.1145/3510003.3510180.

[43] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3Milebeach: A tracer with teeth. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[44] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[45] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *ATC'22: Proceedings of the 2022 USENIX Annual Technical Conference*, 2022.

[46] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Poster: Benchmarking microservice systems for software engineering research. In *ICSE'18 Companion: Proceedings of the 40th Companion to the International Conference on Software Engineering*, 2018.