Offline Trace Generation for Microservice Observability

Dominik Ernst
Technische Universität Berlin
Berlin, Germany
dominik.ernst@tu-berlin.de

Stefan Tai
Technische Universität Berlin
Berlin, Germany
tai@tu-berlin.de

***Abstract*—Microservices are independently deployable and scalable architectural units owned by different teams, supporting continuous software engineering with increased team autonomy. Their distributed nature, heterogeneity and the shift towards decentralized responsibility, however, introduce difficulties for system-wide observability, making it harder to keep track of broader, strategic goals from an enterprise perspective. Further, different systems supporting runtime observability exist, including language-specific libraries, middleware for transportation, processing and storage, which tend to be complex and typically incur significant implementation and configuration overheads.**

To address system-scoped challenges of observability in microservice architectures, we propose an offline approach to distributed tracing. First, intra- and inter-service execution paths are described as a behavioral model of the microservices under observation. This model serves as input to an application workload generator, which produces realistic trace data comparable to trace data won through actual runtime observations from distributed tracing systems. Our approach thus allows to evaluate observability of microservices both in an offline and implementation-agnostic manner, which is less costly and serves as a complement and ex-ante perspective, facilitating a system-wide perspective. We present the idea, a workload generator design and a proof-of-concept prototype along with an initial evaluation.

1. Introduction

Long-term management and efficient operation of software relies on data collected at runtime and by different means, including monitoring, logging and distributed tracing. Today's software systems, in an enterprise context often embodied by microservices, are distributed, deployed on heterogeneous environments in the cloud, and under the pressure to constantly evolve. As software engineering strives to further automate the operation of such systems, *observability* draws increasing interest as a core property for effective control. *Observability systems*, however, for purposes of monitoring, logging and distributed tracing, are complex themselves and offer a plethora of choices regarding instrumentation, performance tuning, transportation, processing and analysis.

Designing and configuring observability systems and, in turn, observable microservices, is challenging for practitioners and researchers alike. First, single or different, possibly conflicting goals of observability, ranging from automated scaling to performance optimization to debugging [1], must be set. Second, observability methods in support of the goals must be chosen. For example, for (end-to-end) performance and failure diagnosis, distributed tracing is likely to the observability method of choice. Third, the instrumentation of the chosen observability methods and systems will become an integral part of microservices infrastructure.

Observability, in particular distributed tracing, also incurs a significant performance overhead. This manifests in system designs [2], [3], [4] relying on sampling to reduce resource footprints. In addition, middleware and storage for observations are required to support various observability goals in the long term.

Many of these challenges are crossing boundaries of individual services and require a system-wide perspective on observability. Aligning observability with strategic goals of the enterprise is impossible without going beyond the scope of individual services and requires a realistic assessment of long-term needs of relevant infrastructure. Niedermaier et al. [5], confirm this lack of a "holistic" perspective, which is further corroborated by the high intrusiveness of observability methods, in particular distributed tracing. Repeatedly assessing relevant, realistic scenarios for observable microservices is very costly, and doing so at runtime also carries a high risk. This risk is exacerbated for major changes to observability systems, for exampling switching backend systems or instrumentation libraries for an existing enterprise-scale application.

In addition, confirming or creating exceptional situations within the production system to stress-test observability systems is not desirable, but so far, alternatives are lacking. This calls for an offline approach, suitable for the repeated assessment of observability systems and supporting the experimental evaluation of architectural and deployment options.

Our contributions towards this fundamental issue are twofold. We define a behavioral model for microservices, inspired by the perspective of end-to-end distributed tracing, capable of representing complex architectures regarding their runtime-observable interactions. The model synthesizes

existing application-level trace data models and allows for the definition of trace data generated by microservices.

Second, we design and implement a trace generator, which takes model instances as input and generates traces targeting actual distributed tracing systems. We evaluate model and generator for a sample use-case and find that a reproducible production of traces is possible, but currently is limited by standardization efforts.

Summarizing, our contributions are as follows:

1) A behavioral microservice model in support of observability, allowing the description of runtime interactions within and between microservices and generated trace data.
2) Implementation of a workload generator for distributed tracing, enabling an economical and realistic generation of trace data using model instances as inputs.

2. Background: Observability of Microservices

Effective operational control at runtime requires observable systems. We differentiate three methods of observability, namely *distributed tracing*, *monitoring* and *logging*, which, depending on requirements and goals of various control mechanisms, serve multiple purposes.

Distributed tracing, at its core, constitutes the activity of recording timestamped, causally related events throughout the execution of a program. Each event captures a specific action of a component or service. For distributed tracing this includes calls to successive services. Events are often paired into so-called *spans*. By propagating uniquely generated random identifiers to subsequent services, relationships between caller and callee are recorded in a *trace context*. Collected traces are sent to a tracing backend, typically *out of band*, i.e. in a non-blocking fashion. All tracing systems rely on the instrumentation of traced components. Consequently, services, proxies or middleware libraries must be instrumented to handle generation and updates to the trace context, as well as the integration of a tracing system's API and backend connection.

Other methods of observability, such as logging and monitoring, traditionally don't provide a context with the same scope as distributed tracing. Recording only local context, they are used for different purposes. Logs serve as the foundation of functional analyses. Log parsing and analysis have found their way into most enterprise systems and are subject to a large body of scientific work. Monitoring in turn is often employed as a source of aggregate, quickly changing data. Resource monitoring helps with scaling operations and the identification of more fundamental issues of software.

In practice, as indicated by [5] and by the nature of systems evolving over a longer time, observability often is an afterthought. Gaining application- or system-wide insights can become challenging, especially as the result of a migration to microservices. Challenges, best practices and related trade-offs for implementing observability so far remain subject to only little research, e.g. [6] for monitoring or [1], [7] for distributed tracing.

One of the reasons why it is naturally difficult to generalize observability-related findings is the high degree of dependency between observability and the application. By providing an abstraction for applications, focused on the production of traces in a realistic fashion, we enable a reproducible, comparative assessment of design decisions related to observability systems.

3. Modeling Observable Execution Paths of Microservices

Microservices are commonly viewed as self-contained software services that implement scoped business functionality and communicate through well-defined interfaces. From a technical perspective, defining characteristics include (1) loose coupling and REST-style communication, (2) decentralized data management, and (3) independent deployment [8]. Microservices aim at breaking the traditional barrier between development and operations by promoting decentralized, continuous delivery following DevOps-like approaches.

Focusing on the core characteristics of microservices, there are two main microservice "workflows" that require attention: First, intra-service interactions, which reflect the functional implementation of the microservice as a layered architecture, possibly including a data management component. Second, inter-service interactions, that is, all calls between different microservices over a network that make up the microservices-based architecture.

3.1. Goals and Fundamental Concepts

To reason about observability of a microservice and a microservices-based architecture means to reason about observable data, their management, necessary development activities and, ultimately, economic implications and business opportunities. As motivated earlier, software engineering methods to approach these topics at an architectural or system-wide level, so far, are lacking.

We propose to elevate observability to a first-class concern in any microservices-based development and to inject an observability perspective early-on by modelling *observable behavior* and produced observations. This implies a logic-centric view of an application, which captures possible *execution paths* within and across services. The internal flow of logic of microservices is modelled as a set of *execution units* and their timings.

The purpose of this model is to allow reasoning about observability in an offline fashion, on an architectural level and with an application-wide scope. Specifically, our model (this section) and trace generator (Sect. 4) are aimed at producing realistic trace data, which can serve as an ex-ante or target view for system behaviour. Furthermore, the trace generator facilitates a benchmarking-based assessment of middleware and infrastructure for distributed tracing.

Existing structural models (e.g., UML component and deployment diagrams) and behavioral models (e.g., UML

behavioral diagrams) focus on abstractions in support of software development that separates design, implementation, deployment, and runtime concerns. The presented behavioral model for observability, on the other hand, is not aimed at serving model-driven implementation or deployment purposes. Instead, it provides an abstraction that allows for the emulation of a microservice architecture from an end-to-end perspective. In particular, target and current implementation can be compared in a reproducible fashion.

Because our model is aimed at describing trace data generation, the first step towards model definition is an analysis of existing trace data models.

3.2. Existing Application-level Trace Data Models

As summarized before, distributed tracing records events and their causality, resulting in a graph-structure, where each node is annotated with a timestamp and additional context information. *Distributed tracing systems* are responsible for the collection of events and aggregate them into said structure. Event data itself is collected by instrumentation libraries, which also include standards for the propagation and serialization of trace context across functions and services.

At the time of writing, two popular open source systems for distributed tracing exist: Jaeger[9] and Zipkin[10]. In addition, cloud providers' services, which constitute managed backends for tracing (Google Cloud Trace, AWS X-Ray), and many other commercial providers exist, with the latter sometimes running under the term "Application Performance Monitoring" (APM). Most of these systems adopt a model of a trace that is similar to the tree-structured model Google describes for Dapper [2].

OpenTracing and OpenCensus [11] are standards, aimed at creating interchangeability between different tracing backends by providing a generic API for source code instrumentation, with libraries available in many programming languages. At the time of writing they are being superseded by a joint community effort under OpenTelemetry [12], which aims to provide standardized APIs and data formats for both distributed tracing and monitoring across different languages and frameworks. The W3C consortium is also working on a standard for traces and the propagation of trace metadata using HTTP headers [13]. Finally, Zipkin implements its own data model (called B3) for context propagation.

The aforementioned standardization approaches rely on a very similar concept of spans and their correlation, but differ in their terminology and expressiveness. OpenTracing, OpenCensus and OpenTelemetry provide a technical specification for both the propagated trace metadata and the traces sent to the backend, while the W3C standard only targets propagated trace metadata.

Distributed tracing systems are also described in scientific literature, with more recent examples being Canopy [4] and PivotTracing [3]. Systems described in literature, however, either (a) are not open source, (b) implement lower, OS-level tracing or (c) are language- or runtime-specific. As far as they are described in the respective publications, we also consider data models from scientific publications for the definition of our model. For comparison and validation, we further take the work of Sambasivan et al. [14] into account, who lay out multiple design decisions and provide orientation for use-case-specific adoption of distributed tracing systems. In particular, the dimension *Relationship Indicators* is adopted from [14] as a superset of existing implementations.

A summary of examined trace data models and their differences in relevant dimensions is given in Table 1. In the following, we clarify the dimensions of comparison and provide a list of common attributes across all trace models. Our main focus for developing a workload model are the open source models [15], [11], [12], [10], since their data formats and API specification are supported by open source distributed tracing systems, cloud providers and commercial products alike.

Propagated data structure is the format available for the propagation of application-specific metadata alongside a trace context as it follows execution logic. OpenTracing, OpenCensus and OpenTelemetry support key-value data, while Zipkin's B3 format does not support such data. For Canopy, not much information about propagation is given, except that some services support the propagation of custom data, which is used for sampling purposes.

Out-of-band trace data structures are used to capture additional context information for spans. These are very similar for all of the trace models: log-like events are implemented by almost all models as a pair of a timestamp and a map-like structure. The three open source models also model key-value data without timestamps as out-of-band data, which are referred to as *tags*.

Relationship indicators capture the semantics of an edge in the trace graph. They vary significantly across all models. At minimum, a "depends on" relationship is modelled by all trace data models, with additional types of relationships available across other models. OpenCensus and OpenTelemetry allow to denominate "sender" and "receiver"-relationships, e.g., implemented by "client" and "server".

3.3. Synthesized Model of Observable Execution Paths

At its core, observability relies on events, produced by the execution of instrumented code. Each client interaction with a microservices-based application leads to multiple services being involved in answering a request. These individual interactions with the application reflect a specific *execution path*. In reality, a path may be the result of conditional branches and depends on the interaction with a client. We are looking for an abstraction that generalizes well enough to represent generated observations from possibly complex architectures, consisting of many services, without adding unnecessary overhead. Because produced events in distributed tracing are ordered temporally, we also need to represent *execution times* of such a path adequately.

TABLE 1: Overview of existing trace data models.

Model Name or Publication	Relationship Indicators	Propagated Data Structure	Out-of-band Data Structure	Special Characteristics
OpenTracing	parent, follows-from	key-value	key-value, events	
OpenCensus	server, client, unspec.	key-value	key-value, events	links to other traces, metrics
OpenTelemetry	server, client, consumer, producer, internal	key-value	key-value, events	links, metrics
Canopy	send, receive; additional event types	-	labels	multiple correlated APIs, e.g. metrics
W3C Tracecontext	parent	key-value	-	Propagation only
Zipkin (B3)	parent	key-value (static fields)	-	Propagation only

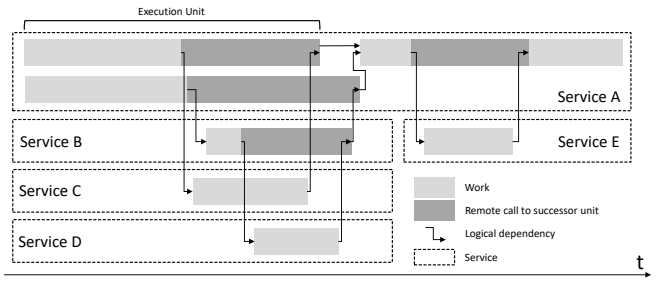


Figure 1: Example of execution units, in passage of time, with calls to other execution units as logical dependencies.

To accomplish this, we model a microservice architecture as set of services, which are subdivided into, so-called, *execution units*. This concept is shown in Fig. 1, for a simple example of five services. An execution unit combines a timing - a delay sampled from a given distribution, which we call *work*, shown as a light gray bar - and optionally a call to a *successor unit*, shown as a darker gray bar. Units can either be executed in sequence, waiting for the completion of previous units and representing a sequential, interdependent flow of logic, or fork out into multiple, parallel executions.

Execution units can reference execution units of other services as their successor and can also be the initiator of requests. In the example shown in Fig. 1, Service A acts as the entrypoint for visualization purposes. Other execution units could also, independently and in parallel, produce requests.

A *service*, as interpreted in the context of a microservice architecture, is thus represented as a combination of one or more execution units, which can be part of multiple execution paths.

Each execution unit serves as a frame for the creation of observable data, with start and end demarcating a span, part of a trace. Optionally, additional trace context data is generated at the start of this time window and added to the trace.

We refer to a called unit as a *successor*, with the calling unit as *inputs*. In the synthesized model, *relationship indicators* between units are integrated as metadata of a successor. It is possible to model forks and joins through multiple successors and inputs, respectively. We find, however, that existing distributed tracing systems cannot adequately represent joins, though, admittedly, these are rare outside of data processing pipelines.

Each execution unit in a path produces corresponding data, called *context*: a short-lived record of all data, created within delimiting events. Based on our analysis of existing trace models, the following mandatory attributes are stored by each context:

- (Gloabally unique) trace ID
- (Locally unique) context ID
- Reference to parent context (ID)
- Relationship indicator to parent context
- Operation or service name
- Timestamp demarcating context start
- Duration of context or timestamp for end of context

Execution of traced program logic - subdivided into multiple units locally and including remote calls - requires propagation of the trace ID and local context ID alongside the flow of execution. As distributed tracing supports the propagation of additional data, the synthesized model does so, too. Propagated trace metadata is called *baggage* and represented by key-value data.

Our analysis of existing APIs and data models also shows the inclusion of additional data with the trace context in two forms: as key-value data (often called *tags*) and timestamped event records, sometimes referred to as logs, which also come as key-value structures. Tags recorded as part of a context and key-value data propagated through baggage are assumed to follow the same structure.

For all key-value data, instead of supporting different types for values, we assume both keys and values for data generation to be string data.

TABLE 2: Synthesized Trace Context Model

Field Name	**Kind**	**Type**
Trace ID	implicit	-
Context ID	implicit	-
Parent Context ID	implicit	-
Relationship indicator (to parent)	modelled	child-of, follows-from
Service Name	modelled	String
Operation Name	modelled	String
Start Timestamp	implicit	-
Context Duration Distribution	modelled	Map <String,Float>
Baggage	modelled	Map<String,String>
Tags	modelled	Map<String,String>
Events (Timestamp)	implicit	-
Events (Data)	modelled	Map<String,String>

A summary of the trace data model is provided in Tab. 2. We differentiate between two *kinds* of data: *modelled* and *implicit*. For the data we need to model, data types

are described accordingly, with a few additional constraints. Implicit data are generated in a specific format by an instrumentation library/API.

The *relationship indicator* field is modelled as an enumerated data type consisting of two values. While we aim to support various types of APIs for instrumentation, modelling arbitrary relationship indicators would require changes to a model, depending on the tracing system to be assessed. The context duration is modelled as a distribution function. Being aware of limitations of libraries for statistical sampling, e.g., from a transformed standard normal distribution, we consequently chose float for parameter values. With constant, Gaussian and exponential distributions being frequent choices, we initially implement support for any required parameters accordingly.

Lastly, we have the requirement of generating textual key value data for multiple fields of the trace context. Ideally, we want to support both fixed value data, representing static context information, but also dynamically generated keys and or values, to represent execution-specific context information. For the latter, the length in bytes, templates or dictionaries could be modelled. Our first implementation of the model allows the definition of randomly generated fixed length strings.

4. t-race: Trace Generator Design

Towards the goal of this work, we present a design and prototype implementation of tool for offline trace generation (*t-race*). In the following, we describe design goals and considerations, as well as some details about the implementation. t-race takes instances of the previously described model as input and produces traces on an application layer. Using t-race as a benchmarking tool turns an instrumentation library and tracing backend system into the system under test (SUT). The prototype is available as an open source library[1].

4.1. Design Goals

To implement a generator for traces requires to emulate an application. The design goals for the t-race are thus derived from representing a microservices architecture adequately and from principles for implementing a benchmarking tool [16], [17]. Having defined an execution path and trace data model for microservices, the generator is developed to parse these definitions and produce traces accordingly. To emulate a realistic generation of traces, t-race must be highly scalable and represent an economic alternative to other types of evaluation. We also need to avoid bottlenecks in the generator itself to use t-race for benchmarking, so that a bottleneck occurs in the SUT first. The efficient, well-tested implementation of popular distributed tracing systems makes this a challenging task.

To cater for the assessment of a wide range of current and future observability systems, the generator should be extensible. Our implementation points at some challenges regarding this requirement, which are discussed later.

Lastly, for an adequate representation of microservice architectures and also related to scalability, the generator is designed to be portable, self-contained and agnostic regarding its runtime environment.

4.2. Architecture and SUT Connection

t-race is implemented as a distributed workload generator, which is deployed as one *coordinator* and multiple *workers*. An overview of the architecture is visualized in Fig. 2 for a sample deployment consisting of three services. The fundamental difference to other workload generators results from the requirement to emulate a microservice architecture. Each worker is responsible to emulate one service. Consequently, workers participate both actively and passively in workload generation. They initiate requests independently of each other if their execution units are configured to do so, and, in parallel, reacting to being invoked by predecessors.

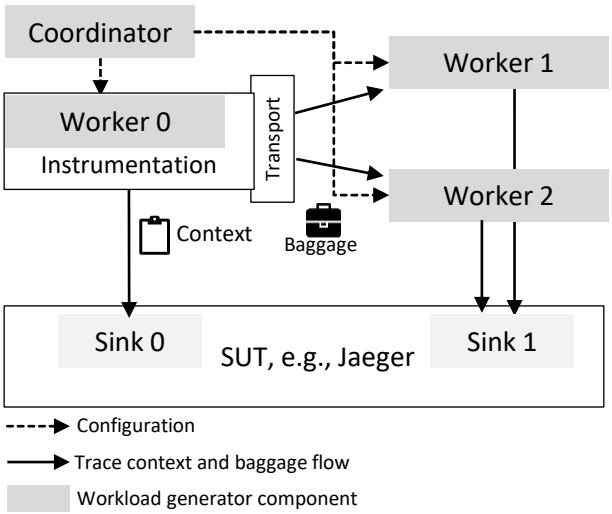


Figure 2: Architecture for a sample deployment of the workload generator.

Three interaction points of t-race with a tracing system exist. Workers need to connect to the SUT endpoints by implementing an API (1) of an instrumentation library, which generates traces during program execution. This API must also be integrated with the interface to successors for propagating trace baggage (2). Lastly, a connection to the deployment of a distributed tracing system must be managed (3). Popular open source systems support different types of deployment. Traces in microservice architectures can be either sent directly to a centralized backend directly or to intermediary receivers. These intermediaries in turn can be deployed either as, so-called, sidecars alongside every instance of a service, or as daemons, where one intermediary is run per host (e.g., a VM).

While implementing a first working solution and getting familiar with libraries for distributed tracing, we found that

1. https://github.com/dominik-/t-race

(3) also requires a configuration of the library for instrumentation regarding *sampling*. Unfortunately, sampling parameters, e.g., for percentage-based or rate-based sampling of traces, are tracing system-specific, even though they come into effect at instrumented source code.

4.3. Workload Abstraction and Load Generation

In benchmarking, workload generators, generally speaking, create artificial load for a SUT. Load is defined by payload characteristics (e.g., type of HTTP request) and a quantitative measure over time, such as a *target throughput*, and, optionally, a *runtime*.

For t-race, payload characteristics are given by the execution path model in the form of execution units, which are grouped by services. For load generation, we implement a configurable *target throughput per second* and runtime. Instead of configuring throughput for every execution unit, we extend the architectural model with *throughput ratios* for each unit. The default value for this ratio is zero, any value above zero represents a factor towards the target throughput and turns the execution unit "active", that is into a "root" span from the perspective of a trace. Throughput at root execution units represent client requests to the application. Two options present themselves for triggering trace generation: (a) using an existing load generator, e.g. for HTTP requests and implementing a respective receiver interface, (b) implementation of our own load generation mechanism.

Both options have advantages and disadvantages. Option (a) solves technical challenges of implementing a load generator, which in itself is a complex endeavor. Depending on the used load generator, we could also represent more sophisticated load scenarios, with changing behaviour over time. In turn, it requires the integration of the generator as a library into our tool, if possible, or the deployment of an additional component. Also, users of t-race would then have to model workload for this additional component, requiring knowledge about it. The requirement for distributed load generation and thus possibly a distributed deployment of such an external workload generator would further limit our options and make executions of a workload more difficult.

Option (b) leads to a more self-contained implementation of trace generation, improving reproducibility and simplicity, two sought after properties of benchmarks [17]. However, given the challenges of implementing a load generator, this option requires significant engineering effort and testing. Given the focus of t-race to serve as a trace generator resembling realistic observations from an end-to-end perspective and remaining portable, we decided for option (b). Portability was the deciding factor, because for option (a), a distributed deployment of multiple workers generating load independently would have added a significant overhead.

4.4. Trace Generator Prototype Implementation

For the prototype implementation of the trace data generator, three aspects were prioritized. First and foremost, the generator must be able to validate and run instances of our execution path model. Second, it should scale accordingly and remain resource-efficient. Lastly, the tool should offer a command line interface, which provides default values where possible and sensible feedback for users.

In turn, for the prototype, other aspects are deliberately regarded as out of scope. Notably, the deployment of worker instances and their allocation within a *run* of the generator are not done automatically, as this would require assumptions about a deployment environment. Given a large number of options for microservice communication, including REST, RPC frameworks and messaging middleware, the prototype is not able to flexibly accommodate for all of these. The choice of communication middleware, however, may be an important factor in the assessment of microservice observability and is discussed later.

As discussed in in Sect. 4.1, we decided to implement our own load generation mechanism. T-race generator was implemented in Golang and requires two inputs: an instance of the observable execution path model (in YAML format) and a deployment configuration file (JSON). The latter contains reachable IP addresses of sinks to send traces to and for each worker two IP addresses, which are used for connections from the coordinator and between workers, respectively.

For the prototype, we first targeted the OpenTracing API as instrumentation, using the official Golang SDK. Communication between coordinator and workers and between different workers is implemented using gRPC. There is much more to the implementation, interested readers are referred to the Github repository[2]. Given the nature of current state-of-the-art distributed tracing systems, a few limitations became apparent during implementation, which are also discussed in Sect. 5.

5. Evaluation

In order to evaluate our proposed approach, we first describe the usage of our execution model and workload generator for a simple example. Afterwards, we report on the fulfillment of design goals. Finally, we discuss the applicability of t-race in more realistic contexts and towards our goal of enabling offline trace generation.

5.1. Illustrative Example: Execution Unit Model and Deployment

For illustrative purposes, a very simple sample application, given in YAML format in Fig. 3a, is used. The resulting deployment architecture of the trace generator matching to this architecture model is given in Fig. 3b. It also contains the deployment of a tracing system. For this example, we use Jaeger [9] as the target system (our SUT) and the OpenTracing API [15] for instrumentation. For readability's sake, the YAML description of Fig. 3a is shortened and does, for example, not contain parameterization of *work*. The full YAML file can be found in our Git repository.

2. https://github.com/dominik-/t-race

For the given architecture, each Service contains only one execution unit, with ServiceA being the only active component, generating 100% of the target throughput. The SUT is also shown in a simplified fashion: the Jaeger backend includes a central "collector", a UI component and is deployed to a container-based environment.

The worker executing the unit with the ID *unitA1* is thus responsible for generating the initial trace context. The second worker, running the units of ServiceB handles incoming requests, extracting trace context data and baggage. It then adds a second span according to the definition of the successor unit (unitB1). The figure indicates context and baggage creation, where this is part of the formal model description. In reality, every execution unit leads to the creation of a context, which, at minimum, needs to contain references to the parent context.

Not shown in Fig. 3b is the coordinator, which is responsible for initial parsing of the architecture description, assigning services to workers and starting trace generation. During the prototype development, the coordinator also has been extended to collect status information on the progress of trace generation, as well as basic information about each produced trace context. This helps significantly with the assessment of performance-related goals and improves simplicity for benchmarking purposes.

5.2. Discussion of Design Goals

t-race was designed with three goals in mind: (1) being scalable and economical, (2) providing an adapter-based design, (3) providing a self-contained approach to trace generation.

While t-race is scalable and resource efficient, we so far did not systematically explore the limits of the load generator. Initial tests indicate that a single worker, without optimizations and running on a common desktop machine, is capable of comfortably producing well above 300 traces per second. Due to the design of t-race, it is also possible to split up trace generation across multiple workers by allocating execution units to "virtual" service. Nonetheless, we plan to explore resource configurations for different workload generator deployments and SUT setups in future work.

The second goal, implementing the workload generator in an extensible fashion, is currently only fulfilled to a limited extent. While we pursued an adapter-based design, the tool as of now only supports trace generation in the OpenTracing format, making it SUT-agnostic to only some extent. With Jaeger as the target SUT, we currently support one type of sink, but additional connectors such as for Zipkin, are straightforward to add.

Lastly, t-race is written in Golang, making it simple to compile for various environments. Given the design as CLI-tool which can run in both coordinator and worker modes without the need to run a separate workload generator makes deployment straightforward. Workload configuration, deployment description and architecture models can be represented as JSON or YAML files, making experiments reproducible and simple to automate.

5.3. Discussion of Broader Applicability

As of the time of writing, we are looking for opportunities to employ t-race in an setting with more practical relevance. In the meantime, we chose Google's microservices demo application, called Online Boutique [18] as a target architecture to help validate our approach. Google's demo application represents an online shop and is specifically implemented to showcase "cloud-native" technology, as well as corresponding tools and principles for observability – including distributed tracing. According to the documentation, most of the architecture is instrumented to support distributed tracing, using Google's OpenCensus libraries in various programming languages.

Running the demo application as-is, however, shows that not all services are instrumented and traces consequently do not provide the desired level of completeness for an insight into runtime behaviour. Thus, we decided to generate traces that represent the application based on given documentation and source code. For the architectural specification of the given microservice architecture as a set of execution paths its graphical representation (c.f., [18]) was used as a starting point. Given the lack of a formal description of the architecture, we looked into interface definitions and the implementation of services to discern execution units within services.

As an example, given many different paths based on a client's interaction with the system, the *frontend* service was modelled with three execution units: one representing a user browsing through the Online Boutique's inventory, one for shopping cart manipulation and one for the checkout process, where a user is placing an order.

Modeling the architecture of Online Boutique showed that while t-race does support the generation of realistic traces without any issue and in a reproducible fashion, it requires a relatively high effort to create a complete model of (intended) interactions of a more complex microservice architecture. We are currently exploring options towards a solution to this challenge, e.g., by automatically generating the execution model from existing traces or other architecture models.

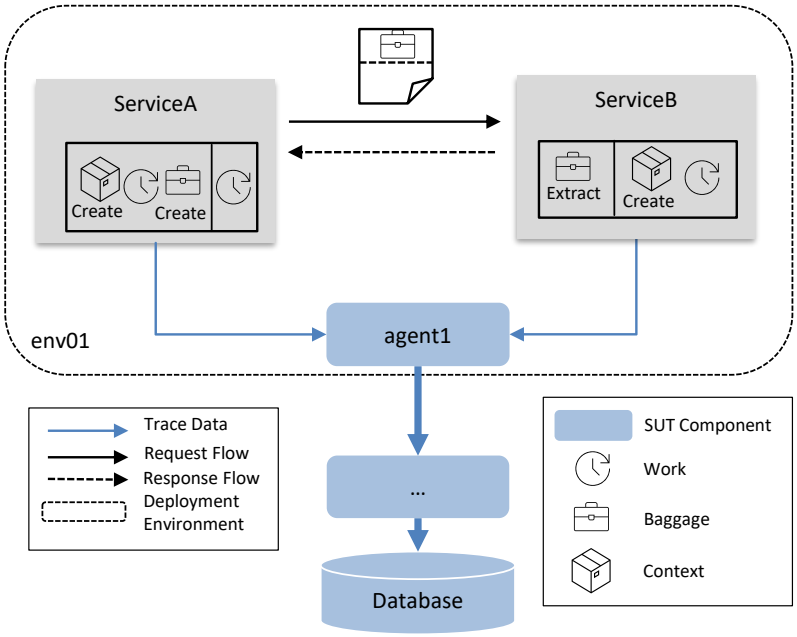
6. Related Work

Modelling of microservice architectures is done in the context of various goals. For example, models are focused on integration and deployment of microservices [19] and their performance evaluation [20]. This work, in contrast to these models, focuses on providing an abstraction for the evaluation of observable runtime behavior of microservice architectures. As such, the proposed model and more closely related work is not about improvement of a microservice architecture in itself, but the assessment of supportive infrastructure.

This problem has also previously been recognized by related work, e.g. Heinrich et al. [21], who, in identifying research challenges for the performance engineering of microservices, state that "The behavior of the dynamic

```yaml
services:
  - id: ServiceA
    envRef: env01
    sinkRef: agent1
    units:
      - id: unitA1
        work: work01
        ratio: 1.0
        successors:
          - svc: ServiceB
            unit: unitB1
            sync: true
        context:
          tags:
            - keyLength: 10
              valueLength: 40
          baggage:
            - keyStatic: clientId
              valueLength: 20
  - id: ServiceB
    envRef: env01
    sinkRef: agent1
    units:
      - id: unitB1
        rel: child
        work: work02
        context:
          tags:
          - keyLength: 10
            valueLength: 30
```

(a) YAML description.



(b) Deployed services/workers and SUT.

Figure 3: Simple example for a textual representation of the model and corresponding deployment.

components, in particular the aforementioned management tools, must also be learned" [21, p. 226]. We believe that our execution model and trace-generator could be highly useful towards this goal.

Düllman and van Hoorn [22] present the most closely related approach, by aiming to benchmark "resilience engineering approaches" for microservices, also stating the missing possibility of running experiments under controlled conditions. Different to our approach, however, their model is specific to REST-based microservices and reliability as a single property. It further does not cater to the requirement of generating configurable, observable output. Instead, their focus lies on representing deployment configurations with the model.

Focusing on challenges related to observability of microservices and other distributed systems, an interview study by Niedermaier et al. hints at the practical requirement for "a common and central view [..] which assists the implementation of a system-wide diagnosis and fault detection" [5, p. 45]. They also discovered that "careless deployment and configuration of monitoring agents" [5, p. 50] presents both a possible security issue and a reason of system instability.

In a similar notion, Haselböck et al. [6] propose a guided approach to microservice monitoring[3], a clear indicator of the broad design space of observability methods and systems.

7. Limitations and Future Work

The presented model and implementation have a few limitations, which we shortly want to address. One limitation, related to necessary development effort, is the adequate consideration of microservice communication middleware in the deployment of the workload generator. With the implementation of gRPC-based inter-service communication only a subset of commonly used middleware can be realistically emulated. As mentioned in Sect. 4, frameworks for instrumentation, communication and backends are often more closely intertwined than expected, making it difficult to build fitting abstractions. OpenTelemetry [12] as an emerging standard with widespread industry support may help towards this challenge.

Greater flexibility in the definition of execution units, e.g., also supporting generation of observable events in the form of monitoring and logging data and at arbitrary

3. While the authors refer to their examined systems and design decisions as monitoring, they include distributed tracing systems, indicating a broader scope.

points in time, would be another meaningful extension. Yet, again, this requires a significant implementation effort and standards for logging and monitoring are fragmented in a similar way to those for distributed tracing.

A conscious choice, but technically a limitation, was to implement our own load generation mechanism. Load is generated as if client-side requests were completely independent of each other. Focusing on the generation of traces and looking at ways to interact with produced data, this decision, however, also frees a user from the requirement to familiarize themselves with another dimension of the system.

Modelling microservices as a set of execution paths, while convenient for observability purposes because data is ordered chronologically, is not intuitive from a developers or architects point of view. A change to the model and load generator, which would allow to differentiate various execution units, for example by referencing to them by entrypoints, could simplify this. In turn, increasing model complexity and moving further away from a time-centric view requires additional validation and increases the number of possible errors. One way to approach this challenge, and planned future extension, is the inclusion of patterns, e.g. related to the deployment of microservices, as templates for services and execution units.

For future work, we are looking at a comparative assessment of different observability systems in a benchmark. As APIs are further standardized and more systems adopt these standards, such a benchmark would benefit both researchers and practitioners and was one of the motivations behind this work.

A fundamental challenge related to all model-based approaches is keeping models representative and correct regarding an existing implementation. As mentioned, an extension towards that end would be the automated generation of an execution path model, or a template for it, based on existing traces of a live system or a different model, e.g. describing the deployment of a microservice architecture. By doing so, differences between expected and actual observable behaviour or conflicting architectural goals would become visible and the assessment of observability systems, starting with distributed tracing, greatly simplified. Given the lacking standardization of trace data at rest, this remains an open challenge.

8. Conclusion

The reproducible assessment of complex architectures, such as microservices-based distributed architectures, is a fundamentally challenging problem of software engineering. By focusing engineering efforts on implementing independently scalable services, bounded by business functionality and driven by continuous delivery objectives, microservices shift responsibilities of deployment, observability and runtime management to underlying platforms and supportive systems. As the latter become indispensable parts of these microservice architectures, software engineering is becoming increasingly concerned with their assessment. The presented approach is concerned with the improvement of microservice observability through by exploring the generation of realistic traces. This could, for example, enable the assessment of observability systems, which impact a system as a whole.

Towards this goal, we propose to model microservices as observable execution paths. These serve as an abstraction for the reproducible generation of realistic trace data. By implementing a prototype tool for trace generation, we prove that offline trace generation is possible and economical, but also conclude that this currently requires significant manual effort.

An analysis of existing standards for distributed tracing and the prototype implementation also revealed remaining challenges for a more pervasive adoption of distributed tracing. In particular, standardization of instrumentation APIs and connectors to systems inevitably incurs high switching cost between different standards and backend systems.

Using our behavioral model and t-race as workload generator, complex microservice architectures can now be conveniently emulated, emphasizing observability in the engineering of microservices and allowing for reproducible assessment of distributed tracing systems.

References

[1] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. Santa Clara CA USA: ACM, Oct. 2016, pp. 401–414. [Online]. Available: https://dl.acm.org/doi/10.1145/2987550.2987568

[2] B. H. Sigelman, L. Andr, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," *Google Research*, no. April, p. 14, 2010. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf

[3] J. Mace, R. Roelke, and R. Fonseca, "PivotTracing: Dynamic Causal Monitoring for Distributed Systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 378–393.

[4] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An End-to-End Performance Tracing And Analysis System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. Shanghai, China: Association for Computing Machinery, Oct. 2017, pp. 34–50. [Online]. Available: https://doi.org/10.1145/3132747.3132749

[5] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner, "On Observability and Monitoring of Distributed Systems – An Industry Interview Study," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, Eds. Cham: Springer International Publishing, 2019, pp. 36–52.

[6] S. Haselböck and R. Weinreich, "Decision Guidance Models for Microservice Monitoring," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 54–61.

[7] J. Waller, F. Fittkau, and W. Hasselbring, "Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability," in *SoSP*, 2014.

[8] "Microservices," https://martinfowler.com/articles/microservices.html, last accessed: 2021-05-10.

[9] "Jaeger: open source, end-to-end distributed tracing," https://www.jaegertracing.io/, last accessed: 2021-05-03.

[10] "Zipkin," https://zipkin.io/, last accessed: 2021-05-03.

[11] "OpenCensus," https://opencensus.io/, last accessed: 2021-05-03.

[12] "OpenTelemetry," https://opentelemetry.io//, last accessed: 2021-05-03.

[13] "W3C Trace Context," https://www.w3.org/TR/trace-context-1/, last accessed: 2021-05-03.

[14] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16*. New York, New York, USA: ACM Press, 2016, pp. 401–414. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2987550.2987568

[15] "OpenTracing," https://opentracing.io/, last accessed: 2021-05-03.

[16] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai, "BenchFoundry: A Benchmarking Framework for Cloud Storage Services," in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds. Cham: Springer International Publishing, 2017, pp. 314–330.

[17] K. Huppler, "The Art of Building a Good Benchmark," in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30.

[18] "Online Boutique," https://github.com/GoogleCloudPlatform/microservices-demo, last accessed: 2021-05-10.

[19] V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, and M. Wurster, "Pattern-based Modelling, Integration, and Deployment of Microservice Architectures," in *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, Oct. 2020, pp. 40–50, iSSN: 2325-6362.

[20] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance Modeling for Cloud Microservice Applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 25–32. [Online]. Available: https://doi.org/10.1145/3297663.3310309

[21] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance Engineering for Microservices: Research Challenges and Directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 223–226. [Online]. Available: https://doi.org/10.1145/3053600.3053653

[22] T. F. Düllmann and A. van Hoorn, "Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 171–172. [Online]. Available: https://doi.org/10.1145/3053600.3053627