Blueprint: A Toolchain for Highly-Reconfigurable Microservices

Vaastav Anand, Deepak Garg, Antoine Kaufmann, Jonathan Mace
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Researchers and practitioners care deeply about the performance and correctness of microservice applications. To investigate problematic application behavior and prototype potential improvements, researchers and practitioners *experiment* with different designs, implementations, and deployment configurations. We argue that a key requirement for microservice experimentation is the ability to rapidly reconfigure applications and to iteratively Configure, Build, and Deploy (CBD) new variants of an application that alter or improve its design. We focus on three core experimentation use-cases: (1) updating the design to use different components, libraries, and mechanisms; (2) identifying and reproducing problematic behaviors caused by different designs; and (3) prototyping and evaluating potential solutions to such behaviors. We present Blueprint, a microservice development toolchain that enables rapid CBD. With a few lines of code, users can easily reconfigure an application's design; Blueprint then generates a fully-functioning variant of the application under the new design. Blueprint is open-source and extensible; it supports a wide variety of reconfigurable design dimensions. We have ported all major microservice benchmarks to it. Our evaluation demonstrates how Blueprint simplifies experimentation use-cases with orders-of-magnitude less code change.

ACM Reference Format:
Vaastav Anand, Deepak Garg, Antoine Kaufmann, Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservices. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 30 pages. https://doi.org/10.1145/3600006.3613138

1 Introduction

Modern cloud applications are increasingly developed as suites of loosely-coupled microservices [17]. The microservice architectural approach decomposes applications into smaller, modular services that communicate over the network. As a result of their success in enabling large-scale and continually-evolving applications, microservices have become ubiquitous among internet companies including Twitter [30], Netflix [16], Uber [29], and others [17].

Microservices are large and complex applications, composed of multiple pieces including frameworks, backends, and libraries. For any application, there are many possible designs, each with its own set of performance properties and issues. As a result, researchers are highly interested in studying, measuring, and improving the performance and correctness guarantees of microservice systems, and developing solutions to potential unwanted behavior when they arise. A salient example of unwanted behaviour are *emergent phenomena* [48] of microservice systems which include cascading failures [52, 70, 75], tail latency effects [18], cross-system consistency [3], and metastable failures [15, 33], among others. By analyzing these behaviors, researchers hope to improve the performance and correctness guarantees of microservice systems, and develop solutions to potential unwanted phenomena [15, 33, 43, 55, 61].

Towards this goal, researchers, both in academia and industry, need to be able to perform three basic actions: (i) reconfigure applications with new features, backends, and libraries to improve their performance and add new features; (ii) reproduce and discover potentially problematic emergent phenomena of applications; and (iii) develop and evaluate solutions on canonical microservice systems. The central requirement of these use-cases is for researchers to *easily explore the design space of microservices*, allowing them to move between different implementations and deployment configurations of the same application quickly and easily.

Enabling design space exploration is difficult for several reasons. First, the design space of microservice systems is enormous. Application design, configuration, and deployment choices vary along several dimensions: specific patterns in the flow of application logic (*e.g.* the presence of concurrent or asynchronous execution branches); the inclusion of particular features, components, or middleware (*e.g.* replicated services, autoscaling, and sharding); and component instantiations and their corresponding configuration (*e.g.* the specific RPC framework used and its timeout and retry mechanisms). Given the lack of standardization across these axes, it is not immediately clear how to support all possible application design dimensions and all possible instantiations for a given dimension in a systematic manner.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP '23, October 23–26, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613138

Second, existing microservice systems, both experimental and production-grade, are implemented as point solutions in this vast design space and are not designed to be reconfigurable or extensible. Thus, the entire burden for moving from one implementation to another falls on the researcher as they have to make deep modifications to microservice applications to fulfill their use-case. A high-level design change, such as replacing the RPC framework, typically translates into thousands of LoC of source-level implementation changes, dispersed across the many processes, components, and backends that comprise the microservice application (cf. §3.1). Implementing a design change is time-consuming, error-prone, and difficult, as it requires understanding the application at the source-code level; yet microservice implementations tightly couple concerns at the source-code level, *i.e.* application logic directly binding to libraries and middleware. Thus, changes that are conceptually simple – *e.g.* replacing an existing RPC framework with a different one – can require wide-ranging and complex manual modifications to many components. Most prior research work on microservices had no alternative to expending this developer effort, and consequently the majority of works deploy and evaluate solutions for only a single application (78%, cf. §B); anecdotally, the developer burden is the limiting factor.

Due to the large design space of microservice applications, there is a significant concern about the generalizability of research results derived from only a single application. A solution may have implicit dependencies on particular application design choices, or worse, it may be an application-specific solution that does not apply broadly at all. For example, Sage [25] assumes synchronous RPCs and Tprof's [34] layer4 grouping assumes assumes non-combinatorial explosion when grouping requests by visited services' execution order; both of which do not hold at Meta [35]. Under normal circumstances, a lack of broad evaluation would be considered a benchmarking crime [31]. However, for microservices it is the prevailing norm.

The goal of this work is to enable researchers to easily reconfigure microservice applications. Our solution, Blueprint, is a microservice development toolchain designed for rapidly *Configuring, Building, and Deploying* (CBD) microservices. Blueprint enables its users to easily *mutate* the design of an application and generate a fully-functional variant that incorporates their desired changes.

The key insight of Blueprint is that the design of a microservice application can be decoupled into (i) the application-level *workflow*, (ii) the underlying *scaffolding* components such as replication and auto-scaling frameworks, communication libraries, and storage backends , and (iii) the concrete *instantiations* of those components and their configuration. An application written using Blueprint avoids tightly coupling these concerns. Instead, these *design aspects are concisely declared* by the user using Blueprint's abstractions, namely, the *workflow spec* and the *wiring spec*. Blueprint's compiler combines these two abstractions to *automatically generate the system*. Changing any given aspect only requires the developer to revisit its declaration in Blueprint's abstractions and not the generated implementation. Moreover, Blueprint eliminates duplicated effort – scaffolding and instantiation logic are implemented once and integrated as Blueprint compiler extensions, to enable Blueprint to automatically change existing Blueprint applications with minimal effort. Concretely, using Blueprint, users can:

- mutate off-the-shelf microservice applications (*e.g.* an open-source benchmark) with just a few LoC, to swap an instantiation (*e.g.* RPC framework), enable or disable scaffolding (*e.g.* replication), or change backends (*e.g.* database).
- develop new application workflows and generate runnable systems. Instead of binding workflow code to specific scaffolding and instantiations (e.g. the choice of RPC framework), those are declared separately with 10s of LoC, and incorporated by Blueprint at compile time.
- introduce support for new instantiations (*e.g.* an experimental RPC framework) or scaffolding concepts (*e.g.* geo-replication) and transparently apply them to existing applications; these are implemented as compiler extensions that are independent and agnostic to specific application workflows.

We have implemented Blueprint in 11k LoC of Go and ported all major available microservice benchmarks to Blueprint (15k LoC), including the DeathStar benchmark [26], TrainTicket [76], and SockShop [47]. Our evaluation demonstrates the ease with which Blueprint enables changes to the design and features of an application; we reproduce known interesting behaviour from a number of prior works; and show that Blueprint is easily extensible with new features that can be reused across applications.

2 Microservice Design Space

The space of possible designs for microservice applications is enormous, and while there may be guiding high-level design principles, every application differs substantially from the next. In this section, we briefly elaborate three important dimensions for the design of a microservice application. These axes are useful both for motivating Blueprint's use cases (§3) and as insights into Blueprint's design (§4).

A microservice application's design can be principally decomposed along three major dimensions: the ***application-level workflow***; the lower-level ***scaffolding*** infrastructure on which the workflow executes; and ***instantiations*** for different infrastructure implementations.

Application-Level Workflow. Microservice applications vary widely in terms of their application-level logic and end-to-end flow of executions through the system's different components. Recent open-source microservice benchmarks cover diverse domains, such as e-commerce apps [28, 47, 71, 72], social networks [26], reservation systems [26, 76] and many others [2, 6, 26, 36, 66]. These applications differ in the number of services and APIs they use internally from only a few (SockShop [47]) to dozens (TrainTicket [76]). Even applications with similar high-level goals (*e.g.* TrainTicket [76] and DSB Hotel Reservation [26]) have vastly different workflows.

Scaffolding. Scaffolding refers to runtime components which are necessary for executing an application but orthogonal (and opaque) to the application's workflow. Scaffolding includes middleware, framework, libraries and backends that provide features such as RPCs, replication, load balancing, circuit breakers, and more [11]. Scaffolding can be changed without affecting the application's workflow and functional behavior. For example, the original DSB Social Network [26] uses one runtime instance of each service by default; however, a changed version of the application could modify its scaffolding to replicate service instances, without changing the end-to-end application workflow [42].

Instantiations. For every piece of scaffolding there may be different implementations to choose from, configuration dimensions of those implementations, and choices for configuration parameters, *e.g.* to enable RPC an application may use gRPC, Thrift, or some research prototype framework [39].

Overall, the choice of an application's workflow, scaffolding, and instantiations have different implications for the application's performance, correctness, and reliability.

3 Blueprint Use-Cases

This section outlines challenges faced by researchers today with respect to three core use-cases. Across the three use cases, we motivate a common need to iteratively Configure, Build, and Deploy (CBD) variants of microservice applications that have subtly different designs. Blueprint, which we will introduce in §4, is designed to address this need.

3.1 UC1: Mutating Applications

To investigate and experiment with changing workloads and deployment conditions, researchers may wish to *mutate* an application – *i.e.* change, reconfigure, or expand some aspect of its design. A mutation modifies the application along one or more of the aforementioned axes (§2). For example, changing the RPC framework from Thrift to gRPC is a mutation that only modifies instantiations; introducing replicated services and a load balancer is a mutation that modifies both scaffolding and instantiations; and refactoring the application workflow is a mutation that typically leaves scaffolding and instantiations untouched [19]. Ultimately, these changes modify the application to move its design from one point to another in the design space, creating a new variant of the application.

Ideally, users should be able to mutate an application with minimal effort. Yet today, a conceptually simple mutation may require far-reaching and time-consuming modifications to source code and configuration. For example, switching to a different RPC framework instantiation in an application with 30 services requires modifying all of those services. Replacing one instantiation with another is difficult because interfaces offered by instantiations of the same piece of scaffolding can differ widely, and existing systems tightly couple the application's workflow, scaffolding, and instantiations. Similarly introducing new scaffolding, such as enabling distributed tracing, entails exhaustively updating *all* services to support it; and changing an application's workflow requires binding the new or changed code to scaffolding and instantiation interfaces.

To understand the scope of mutations to existing microservice applications, we surveyed 464 forks of popular microservices benchmarks. We found a total of 146 application variants that apply mutations that include adding tracing, removing tracing, adding replication, adding georeplication, switching RPC frameworks, and more (cf. §B.3). As an example of the cost of manual mutations, an instantiation change in DSB Social Network [26] to support Sifter [40] required 1,289 lines of manual code change and took 2 weeks to complete based on commit timestamps (cf. §6.3).

3.2 UC2: Reproducing Emergent Phenomena

Emergent phenomena, or emergent behaviors [48], are aspects of the system's runtime behavior that are not localized to any one service or component, instead arising as the cumulative effect of interactions *between* components under a given workload and application design. Emergent phenomena encompass end-to-end performance, correctness, and reliability concerns of an application – notable examples include cascading failures [52, 70, 75], tail latency effects [18], cross-system consistency [3], laser of death [51], and metastable failures [15, 33]. See §B.1 for a detailed list of known emergent phenomena.

Left unchecked, emergent phenomena can lead to degraded service and even outages [46, 50, 51]; thus they are the focus of a range of recent work from both industry [61] and academia [15, 33, 43, 55]. In general, researchers need the ability to elicit emergent phenomena in microservice applications, to determine the conditions under which they emerge, and their effects on application behavior.

Few existing microservice systems exhibit emergent phenomena out-of-the-box, as they arise due to specific workflow, scaffolding, or instantiation choices, combined with workloads and deployment conditions. To study emergent phenomena, researchers must therefore mutate existing applications to find variants that exhibit the phenomena. For example, no out-of-the-box microservice benchmark exhibits cross-system inconsistency [43, 61] or metastable failures [15]; researchers studying these phenomena manually mutate existing microservice applications to elicit them [22, 42].

It is not straightforward to identify a design that exhibits an emergent phenomenon. Changes to the scaffolding or instantiations can affect performance and error-propagation characteristics non-linearly, making it difficult to predict the effects of even minor alterations. Moreover, certain emergent phenomena only manifest under specific workloads which are not known to developers a priori. Overall, this makes empirical exploration of the design space essential: researchers may need to mutate an application multiple times before finding a variant that clearly exhibits a given phenomenon.

3.3 UC3: Prototyping and Evaluating Improvements

Researchers often need to prototype and evaluate *improvements* to microservice applications. An improvement can include applying workflow design patterns [19], enabling novel scaffolding [42], and implementing instantiations with better properties than the existing ones [39]. Improvements typically target some performance, correctness, or reliability concern, *e.g.* the application's scalability, latency, or some emergent phenomenon. For example, FIRM [55] is an improvement for mitigating SLO violations; it leverages distributed tracing scaffolding and introduces a novel orchestration scaffolding and instantiation.

To develop and evaluate improvements, it necessary to mutate existing applications to incorporate the improvement. This entails the same degree of manual overhead described for UC1 and UC2, but is exacerbated in two ways: first, development may require multiple iterations of design to converge on appropriate interfaces, potentially entailing repeated mutations over time integrating successive versions of the improvement; second, best practices for rigorous evaluation demand that any improvement should be evaluated across multiple, diverse applications [31], thus requiring effort to mutate multiple applications, not just one.

Due to the high cost of mutating applications, most research works today evaluate on only a single microservice application (78%, cf. §B.2). Consequently it is difficult to distinguish whether a proposed improvement would be similarly effective for other applications, or just for the specific application selected. Moreover, an improvement might make assumptions about an application's design that restricts its broader applicability. For example, FIRM [55] assumes a deterministic critical path for each API, so might not apply to workflows with concurrent or branching RPC calls.

4 Design

Blueprint is a toolchain that offers first-class CBD support for microservice applications. Instead of directly implementing runnable application artifacts (*e.g.* code, container images, etc.), these are generated by Blueprint's compiler. Developers are still responsible for implementing application workflows (or re-using open-source workflows), and these must adhere to Blueprint's abstractions. Likewise developers must separately specify which scaffolding and instantiations to apply to the workflow. Blueprint's compiler will automatically generate the necessary artifacts (*e.g.* glue code, configuration, wrappers, scripts, and more) to produce a runnable variant.

Separation of Concerns. Blueprint's key insight is that an application's workflow, scaffolding, and instantiations are conceptually orthogonal and thus should be separable when specifying the application. The workflow of an application is independent of the specific choice of, *e.g.* RPC library, or the presence of particular scaffolding, *e.g.* replication. In today's applications these are tightly coupled, with application code that intertwines workflow, scaffolding, and instantiations, yet the interfaces between them are narrow because they are conceptually separate concerns and little information is needed of one to specify the other. Blueprint thereby only combines workflow, scaffolding, and instantiations at compile time, thus avoiding tight-coupling or hard-coding.

Compile-time integration. Despite a clean conceptual separation between workflow, scaffolding, and instantiations, in practice these manifest in diverse ways and at different granularities, *e.g.* application-level libraries, sidecar processes, container images, and orchestration framework configuration. Compile-time integration thus becomes necessary for Blueprint to abstract across diverse granularities. Blueprint's compiler encapsulates a wide range of concerns ranging from code, process, and container image generation, to templating, dynamic addressing, and configuration.

Examples. Blueprint enables users to:

- obtain variant applications by simply recompiling with different scaffolding and instantiation choices.
- mutate an application by changing as little as 1 LoC.
- change instantiations (*e.g.* RPC, database implementations) with as little as 1 LoC.
- develop or change workflows with less cognitive overhead, since workflow logic is not coupled with scaffolding or instantiations.
- integrate new instantiations and scaffolding concepts as one-shot compiler plugins, reusable by any existing or future application. Implementing a plugin does not require knowledge of or compatibility with other plugins.

4.1 Blueprint Applications

A Blueprint application consists of two key abstractions. The *workflow spec* abstraction is the implementation of the application's workflow. The *wiring spec* abstraction declares the scaffolding and instantiations to apply to the workflow. We describe each in detail.

Workflow Spec. The basic building block of a workflow is Blueprint's *service* abstraction: developers can declare an interface for the service with typed methods, and provide an implementation of those methods. Blueprint currently supports Go. Fig. 1 defines a ComposePost service from the DSB Social Media application [26] that enables callers to

```
1 type ComposePostService interface {
2  ComposePost(ctx context, userID int64, text postContent) error
3 }
4 type ComposePostImpl struct {
5  postStorageService PostStorageService
6  userService UserService
7 }
8 func NewComposePostImpl(ps PostStorageService, us UserService) *
       ComposePostService {
9  return &ComposePostImpl{ps, us}
10 }
11 func (c *ComposePostImpl) ComposePost(ctx context, userID int64, text
       postContent) error {
12  creator, err := c.userService.GetUser(ctx, userId)
13  post := Post{Creator: creator, Text: text}
14  return c.postStorageService.StorePost(ctx, post)
15 }
```

Fig. 1. Workflow Spec for DSB Social Network ComposePost.

```
type Cache interface {
 Put(key []byte, value []byte) error
 Get(key []byte) ([]byte, error)
}
```

Fig. 2. Built-in interface for a *cache* backend component.

```
1 normal_deployer : Modifier = Docker()
2 rpc_server : Modifier = GRPCServer()
3 tracer : Tracer = ZipkinTracer()
4 tracerModifier: Modifier = TracerModifier(tracer=tracer)
5 server_modifiers : List[Modifier] = [rpc_server, normal_deployer,
       tracerModifier]
6 post_cache := Memcached()
7 post_db = MongoDB()
8 user_db = MongoDB()
9 us = UserServiceImpl(user_db).WithServer(server_modifiers)
10 ps = PostStorageServiceImpl(post_cache,post_db).WithServer(
       server_modifiers)
11 c1 = Container(ps,post_cache)
12 cs = ComposePostServiceImpl(ps, us).WithServer(server_modifiers)
```

Fig. 3. Wiring Spec for three dependent DSB services. Zipkin tracing is enabled for all services; they are deployed in Docker containers; and communicate using gRPC. Cache and database instantiations are Memcached and MongoDB respectively.

upload new social media posts. Method implementations can be arbitrary and import arbitrary libraries.

Blueprint's *backend* abstraction offers interfaces for different kinds of backends such as caches, databases, and queues; Fig. 2 depicts a simple Cache interface. Unlike for services, a backend instantiation will have method implementations provided as part of its Blueprint compiler integration, e.g. memcached will provide a memcached client.

Blueprint imposes a *dependency injection* pattern on service implementations. A service can invoke the methods of another service or a backend (Line 12). However, a service is forbidden from instantiating other services or backends, which can only be received as constructor parameters (Line 8). Likewise, although invoking another service is simply a method call in the workflow spec (Line 12), the developer should not assume that other services and backends are running in the same address space, correspond to just a single instance, or are of a particular implementation. Services do not directly incorporate any scaffolding (*e.g.* configuring an RPC server) or bind specific instantiations (*e.g.* binding to a memcached client library) as these are automatically integrated later by Blueprint's compiler. Blueprint uses Go's error-handling conventions to wrap and encapsulate errors that may be introduced from scaffolding, and Go's context propagation for implementing scaffolding such as tracing.

The above restrictions are necessary for several reasons. First, the addressing scope of callee services can vary – they could be application-level instances in the same address space, or running in separate container instances in a different datacenter, requiring network calls and address translation. Second, scaffolding may interpose calls between services, *e.g.* to add functionality like tracing or to implement RPCs. Third, scaffolding may duplicate or replicate service instances in some way. In all cases, Blueprint's compiler is responsible for later generating the code that instantiates, configures, and addresses services, at the application, process, and container granularities. §4.2 relates the above service abstraction to corresponding compiler abstractions.

Blueprint includes numerous out-of-the-box open-source applications, and once a developer has implemented an application's workflow spec it only needs to be revisited if workflow logic needs to change. Changes to scaffolding and instantiations happen through Blueprint's wiring spec.

Wiring Spec. The *wiring spec* declares the topology of the application, applies scaffolding, and configures instantiations. Fig. 3 depicts a wiring spec for three dependent DSB services [26]. Wiring is declared using a strongly-typed DSL with C-style macro support (Fig. 3). The wiring spec declares instances of services named in the workflow spec (Line 9) and links instance dependencies (Line 12); it also declares and links instantiations of backends (Line 7) and scaffolding (Line 2). The wiring spec also groups instances into deployment units such as processes and containers.

To enable scaffolding, the user refers to it using unique keywords and syntactic sugar in the wiring file (Line 2, 4). A corresponding compiler plug-in will be invoked during compilation to generate and modify code and other artifacts. Scaffolding can potentially apply to service instances, processes, or container images, depending on the specific feature it enables. For example, Zipkin tracing (Line 3) applies to service instances by wrapping with proxy classes.

A typical wiring spec is concise – tens of LoC – and easily modified by other Blueprint users. Users do not need to touch the workflow spec to enact changes in the wiring spec. Blueprint will recompile an application variant, potentially generating substantially modified code artifacts, without requiring manual intervention from the user.

Compiler Plugins. Scaffolding and instantiations are implemented as *compiler plugins*. Most applications will make use of Blueprint's out-of-the-box compiler plugins; however a researcher wishing to prototype new functionality or improvements may wish to integrate that functionality with Blueprint by developing a compiler plugin. Compiler plugins integrate with Blueprint in three places. First, a plugin can introduce keywords and syntatic sugar to the wiring spec. Second, as described in the next section, a plugin can extend Blueprint's IR to add new node types or extend existing node types. Third, a plugin provides logic for generating code, configuration, and other artifacts specific to the plugin. For example, Blueprint's gRPC plugin invokes the Protocol Buffers compiler and generates client and server wrapper classes. Blueprint is designed in a way that implementing a plugin is independent of the implementation of any other plugins. Most core concepts of Blueprint are implemented as compiler plugins, *e.g.* application-level Go service instances, Go processes, and Docker containers.

4.2 Intermediate Representation (IR)

The canonical representation of a Blueprint application is the compiler's Intermediate Representation (IR). Blueprint's compiler takes as input an application's workflow spec, wiring spec, and enabled compiler plug-ins. The IR of an application is a verbose and well-structured graph that represents the concrete layout and hierarchy of components along with their interactions. The IR of an application depends on both the workflow and wiring spec, and a change to just the wiring spec (e.g. to add replication) will result in a different IR. The IR is designed to support the flexibility and extensibility of the compiler. Fig. 4 depicts the IR graph for the wiring spec outlined in Fig. 3.

Component Nodes. Components are entities that will ultimately be instantiated in the generated system; they are represented as nodes in the IR. All services defined in the workflow spec have corresponding component nodes; likewise all backends and instantiations. Component nodes can exist at different granularities, *e.g.* representing an application-level service instance, a pre-defined binary (*e.g.* a MongoDB instance), or a pre-built container image (*e.g.* a ZipkinServer). IR nodes are typed and plugins may introduce new IR node types and implementations.

Namespace Nodes. Components of the same granularity can be grouped under a namespace node to create a component of coarser granularity. For example, a Go namespace node groups together application-level instances (*e.g.* service instances) into a single Go process. Similarly, a process namespace can be grouped into a container, and a container namespace into a deployment. During compilation, namespace nodes generate runnable artifacts (*e.g.* code, container images) that instantiate their contained components. Typing on nodes ensures that namespace nodes can only contain children of a compatible granularity. Blueprint can be extended with new namespaces; *e.g.* support for georeplication would introduce a region namespace; supporting C++ workflows would introduce a Cpp namespace.

Dependencies. Services in the workflow spec can invoke other services and components; in the IR these dependencies are represented as edges between component nodes. RPC edges are directional indicating the caller-callee direction and declare the method signatures of the invocations.

Modifier Nodes. Scaffolding can interpose edges between components, *e.g.* to modify method signatures, add proxy functionality, or enable addressing across address space boundaries. We call these *modifier* nodes because they attach to component nodes and mutate the component's edges (e.g. to introduce client side and server side code). Modifiers must be opaque to the caller component whom expects a particular method signature from the callee; thus a modifier typically comprises a client-side transformation function and a corresponding server-side function that inverts the transformation (e.g. serialization and deserialization).

Visibility and Addressing. Dependent components can run in different processes, containers, or machines. For example, an application could be compiled as an all-in-one process, or using a container per service. Although there may be an edge between two components, it is possible that those components are not *visible* to each other, *e.g.* if a service has not been wrapped with an RPC server, it cannot receive remote invocations. Thus, edges between components are annotated with their visibility level. Nodes can expand the visibility of any edge traversing outside that node, *e.g.* an RPC modifier enables communication between processes.

Generators. A component declared in a wiring spec might correspond to a single concrete runtime instance (*e.g.* those in Fig. 3), or as a result of applying modifiers, to multiple runtime instances. For example, an autoscaling modifier might dynamically create and destroy multiple component instances at runtime. In general, *generator* nodes contain other nodes and represent instances that will be dynamically created at runtime. Generators restrict the visibility of contained nodes, since there will be multiple dynamically-generated instances of the contained nodes. Generators are typically coupled with functionality such as a load balancer to enable addressing of the dynamically created instances.

4.3 Compilation

Blueprint compiles an application in two steps. First, it processes the wiring spec and workflow spec to instantiate the specific IR graph representing the application and its topology. Second, it invokes compiler passes and scaffolding-specific plugins to generate the runnable artifacts. We explain both steps in detail below.

4.3.1 Wiring & Workflow Spec to IR

Blueprint parses the workflow spec to identify all workflow services that have been defined, and loads the definitions of standard library backends that can be instantiated. Next, Blueprint processes the wiring spec to extract the list of components instantiated in the wiring spec, creating the appropriate IR nodes for each. Blueprint applies modifiers

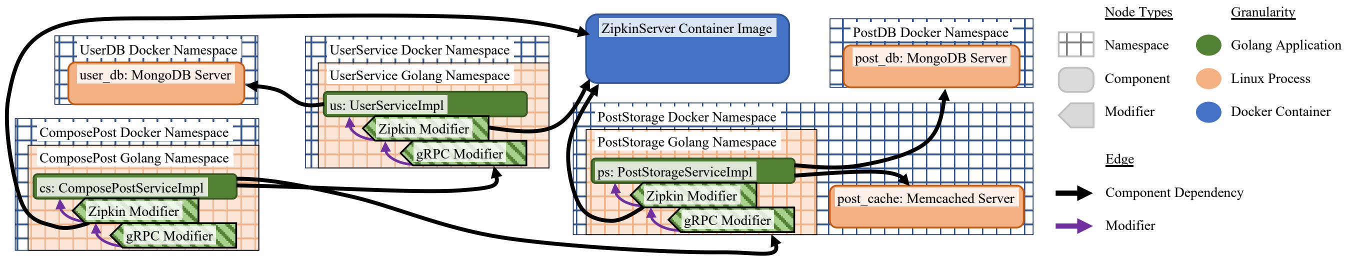


Fig. 4. A depiction of Blueprint's IR for the three DSB services outlined in Fig. 3. Node shape is based on node type, and nodes are colored according to their granularity. Edges are color-coded according to the type of relationship.

to components by creating additional IR nodes representing the scaffolding. Blueprint then creates directed edges between components to encode the dependencies defined in the wiring spec. Blueprint then extracts the various component groupings and granularities to generate the namespace nodes as well as to add the visibility attributes to the dependency ages between the component nodes. Lastly, Blueprint performs a pass on the IR graph to allow modifier nodes to add, delete, or change nodes in the IR graph. For example, a replication modifier could duplicate the IR nodes representing a component, and insert a load balancer node.

4.3.2 IR to Implementation

Once the IR graph is constructed, Blueprint checks the visibility of edges, *i.e.* that any component that calls another component will be capable of doing so. Blueprint then proceeds to the artifact generation step. Each node of the IR graph will have plugin-specific logic for generating its own code, configuration, or artifacts needed for instantiating the component in the system. Blueprint traverses the IR graph, invoking plugins at IR nodes and collecting the artifacts that are generated.

Artifact Generation. Blueprint generates code and artifacts in a hierarchical manner, starting from the innermost nodes in the IR graph. For service nodes defined in the workflow spec, no extra code generation is required and only dependencies are gathered. For modifier nodes, the compiler invokes the plugin that corresponds to the node. The output of the previous node is passed as input to the plugin, allowing the plugin to potentially generate code that wraps, extends, or changes the previous output. For namespace nodes such as go processes or docker containers, generating code entails packaging code generated by the c ontained nodes, and generating code that instantiates those nodes.

As an example, consider the generation process for the ComposePostService in Fig. 3. The generation procedure starts at the ComposePostServiceImpl node, `cs`, in Fig. 4. This node simply gathers the code dependencies directly from the workflow spec where it is defined. The compiler then steps outwards to the ZipkinModifier which inspects the method signature list of `cs` and generates a wrapper class that adds trace contexts to all methods. Next, the gRPC plugin generates protobuf RPC message definitions for the expanded `cs` methods, invokes the gRPC compiler, then generates client and server instantiation code. The compiler proceeds in inverse topological order and next invokes the Go Namespace plugin to package all contained code and generate a main method that instantiates the service, wrappers, RPC server, and clients to dependencies. The compiler proceeds similarly for process nodes and container nodes.

Resolving Dependencies. As part of the generation process, Blueprint gathers code dependencies across namespaces from the IR graph to ensure that remote components are addressable. For example, if a service invokes another over RPC, running in a different container, it must therefore include client code for the target service and its modifiers. Any node crossed by this edge must receive and forward the target service's address as an argument (*i.e.* so that the target service binds to an address that the source service can dial, and so that docker containers publicly expose the relevant ports). If the remote components are not addressable by a service, Blueprint's compiler will return an error citing that the edge between the two services lacks the necessary visibility.

5 Implementation

Blueprint is implemented in Go in 10,892 LoC, which includes Blueprint's compiler, the wiring spec DSL, component interfaces and their implementations, debugging and logging features, and other features implemented as modifiers.

Blueprint's compiler is implemented in 4062 LoC. Blueprint provides first-class support for Go workflow specs. We selected Go because it is designed specifically for high-performance RPC services, and has convenient mechanisms for handling concurrency, errors, and context propagation. Blueprint is not constrained to Go; the abstractions of Blueprint enable extension to multiple languages with no additional difficulty. Blueprint's wiring spec is currently a Python-based DSL that also allows C-style macros; this is 771 LoC, and we are considering more flexible programmatic approaches for future work.

We reimplemented five applications from three microservice benchmark suites described in the literature: the SocialNetwork, Media, and HotelReservation applications from the DeathStar Benchmark (DSB) [26], TrainTicket [76], and the SockShop benchmark [47]. We present an analysis of the LoC effort required for porting these applications in §6.1. We additionally outline the features currently supported by Blueprint in §6.5 and the LoC of implementation required to implement the compiler plug-ins.

6 Evaluation

Our evaluation of Blueprint seeks to answer the following:

- Does Blueprint reduce effort for design space exploration (UC1)? (§6.1)
- Can Blueprint help reproduce emergent phenomena in microservice applications (UC2)? (§6.2)
- Does Blueprint reduce effort for prototyping improvements (UC3)? (§6.3)
- Are Blueprint-generated systems realistic? (§6.4)
- Is Blueprint easy to extend with new scaffolding and instantiations? (§6.5)
- What is the cost of Blueprint's abstractions? (§6.6)

Experimental setup. All experiments use a cluster comprising eight machines, each with four Intel Xeon E7-8857V2 processors, 48 cores and 1.5 TB RAM. We deploy each service in a separate container. We use a simple open-loop workload generator that can be configured to exercise APIs of the generated system with a specified request rate and API distribution; this runs on a separate machine.

6.1 UC1: Design Space Exploration

Reducing Implementation Effort. To demonstrate that Blueprint makes it easy to implement realistic microservice systems not specifically designed for our evaluation, we have re-implemented five existing microservice applications in Blueprint. We selected these systems based on their popularity in microservice research as highlighted in §B.2. Tab. 1 shows the LoC needed to implement the workflow spec and wiring file for each application in Blueprint. We compare the LoC needed to those in the original implementations. Blueprint reduces the code footprint by 5–7× for each application by eliminating the need to implement scaffolding and instantiations alongside workflow code. In the original implementations, scaffolding was tightly coupled with workflow code, thus inflating the amount of code that a user needed to write. By decoupling the workflow specification from the scaffolding code and moving scaffolding code generation to the compiler, Blueprint reduces the volume of code required to implement a workflow. One beta user of Blueprint noted that this decoupling also made it easier for them to understand and implement the workflow specification.

System	Orig. (LoC)	Blueprint (LoC)		Reduction
		Spec	Wiring	
DSB SocialNetwork	8 209	1 478	57	5.4×
DSB Media	7 794	1 401	42	5.4×
DSB HotelReservation	5 160	679	63	7.0×
TrainTicket	54 466	9 639	166	5.6×
SockShop	13 987	2 261	40	6.1×

Tab. 1. Lines of code of original and Blueprint implementations of popular open-source microservice systems

Changing workflow specs. We compare the LoC required to make a change to the design of the application in the original system and compare that to the effort to implement the same change in the Blueprint implementation of the application. In pull request #101 of DSB SocialNetwork [19], the authors inverted caller-callee relationships between `ComposePostService` and `TextService`, `UniqueIDService`, `UserService`, and `MediaService` to improve application performance. They modified 5,140 LoC. We implemented the same change in the Blueprint version of the system by modifying 288 LoC of workflow spec, and 7 LoC of wiring spec — a 17× reduction. The substantial difference in code changes arises due to Blueprint's separation of concerns: in the original implementation, changing interfaces between services required changing scaffolding and instantiation code, which was tightly coupled with workflow code. However, with Blueprint, scaffolding and instantiation code changes are handled by the compiler, and only the workflow specification required manual changes.

Changing scaffolding and instantiations. Blueprint makes it easier to enable or disable new scaffolding in an application. Based on our survey in §B.2, we found that a popular change in existing microservice systems is to enable or disable tracing [5, 13, 32, 37]. For example, disabling tracing in a variant of DSB SocialNetwork required 418 LoC [37]. In contrast, the same change required 5 LoC wiring spec change for the Blueprint implementation of DSB Social Network. This small wiring spec change automatically removes ~2 KLoC from the generated system, including all tracing source code modifications and configuration files needed to enable tracing in the runtime.

Performance-Driven Design Exploration. Finally we perform a study requiring many configure, build, and deploy iterations. We first study the performance impact of different choices in DSB HotelReservation and DSB SocialNetwork [26]. Fig. 5 shows the performance impact of changes along two dimensions: (i) the RPC framework (gRPC or Thrift); and (ii) the size of the clientpool (relevant only for Thrift, as gRPC multiplexes connections on a single connection). We find that gRPC outperforms Thrift for both applications and find marginal differences when varying the clientpool size.

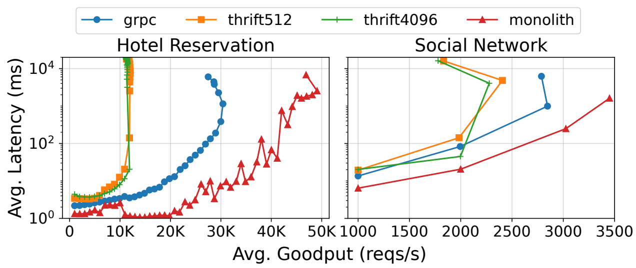


Fig. 5. Blueprint enables easy performance-driven design exploration.

Next, we use Blueprint to generate monolithic versions of both applications, where all services run in a single process and communicate directly through function calls. This allows us to quantify the performance cost of breaking the application down into a microservice architecture. In both cases, we run all services on a single machine. The monolith line in Fig. 5 shows that the monolith version outperforms the microservice version of the application. This enables an empirical decision for when the complexity of a microservice architecture is justified from a performance point of view.

To generate these variants, we only needed to modify 5–10 LoC in the wiring spec, illustrating how Blueprint enables design space exploration with minimal manual effort.

6.2 UC2: Eliciting Emergent Phenomena

Through two case-studies we demonstrate that Blueprint is capable of generating systems for exploring emergent phenomena, namely *metastability failures* [33] and *cross-system inconsistency* [61]. We modify the Blueprint implementations of DSB HotelReservation and DSB SocialNetwork to exhibit these specific emergent phenomena; for readability we refer to these simply as HotelReservation and SocialNetwork.

6.2.1 Case Study I: Metastability Failures

We adapt HotelReservation and SocialNetwork to showcase the four different kinds of metastability failures [33]. The required changes described below are to the wiring spec and amount to at most 3 LoC per failure type.

Load spike trigger workload amplification (Type 1). We modify HotelReservation to add a 500 ms timeout to every inter-service RPC. We also modify the services to retry up to 10 times on error. We start with a 10 K requests/s (Rps) workload for 60 s, then increase to 30 KRps for 30 s, and then revert to 10 KRps. Fig. 6a shows the mean and 99^{th} percentile service latencies over time. At the 1-minute mark, the sudden workload increase triggers the majority of requests to time out, in turn causing more requests to be generated due to retries. This trigger keeps the system in a metastable state and even after decreasing the load, the system does not recover to a stable state.

Load spike trigger capacity degradation amplification (Type 2). To induce this type of metastability failure, the authors [33] limited the maximum service heap size. We experiment with HotelReservation's `ReservationService`. As Go offers no direct control over heap size, we instead increase the garbage collection (GC) frequency by causing the GC to trigger whenever the heap is 75% full instead of the default 100% (for this, we set the environment variable `GOGC` to 75). We run a 20 KRps workload for 10 mins; at the 5 min mark we introduce low-level CPU contention for 30 s using FIRM's anomaly injector [55]. Fig. 6b shows mean service latency over time. Here, the CPU contention acts as a trigger by increasing the GC duration, which results in frequent stop-the-world GC phases, causing other requests to start timing out and generate more retries. This metastable state also persists after the CPU contention disappears.

Capacity decreasing trigger workload amplification (Type 3). To induce this metastability failure, we first modify HotelReservation to have 1 s timeouts and up to 10 retries on every RPC. We run a 24 KRps workload for 2 mins. After 1 min, we inject low-level resource contention with FIRM's [55] anomaly injector for 30 s to decrease available CPU capacity. Fig. 6c shows the mean and 99^{th} percentile service latencies over time. CPU contention starting at 1 min causes timeouts leading to retries that overload the system and keep it in a metastable state after CPU contention disappears.

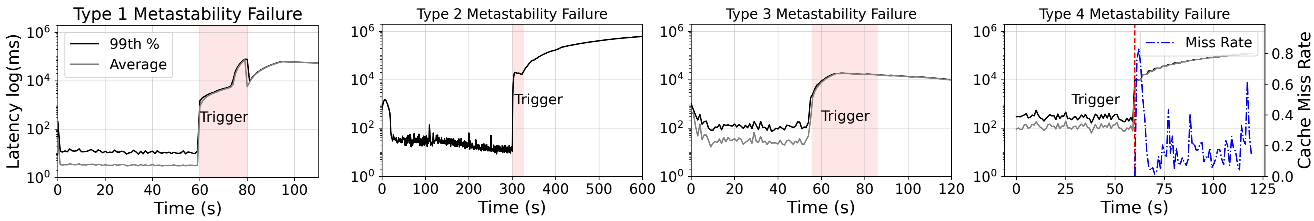
Fig. 7 illustrates vulnerability for different request rates, trigger durations, and maximum retries. At higher request rates, even a short trigger can cause the system to move into a metastable failure state. In contrast, at lower request rates, short triggers only cause transient issues. Fewer retries only minimally increase the tolerable trigger duration.

Capacity decreasing trigger capacity degradation amplification (Type 4). We modify SocialNetwork with an internal 1 s timeout and up to 10 retries. We run this experiment in two phases. First, we fill the `UserTimelineCache` with all content from the `userTimelineDatabase`. In the second phase, we run a 3 KRps workload for 2 mins. At 1 min, we flush the `UserTimelineCache` which then causes future requests to query the database. Fig. 6d again shows the mean and 99^{th} percentile service latencies along with the observed cache miss rate. The cache flush at 1 min overloads the database and causes cascading timeouts and retries. This overload prevents the cache from repopulating and keeps the system in a metastable failure state.

In all cases, Blueprint enables rapid exploration of different designs, such as adding timeouts and retries, which, in turn, enables the reproduction and analysis of metastable failures.

6.2.2 Case Study II: Cross-System Inconsistency

We add replication scaffolding to SocialNetwork to elicit cross-system inconsistencies. Cross-system inconsistencies



(a) Type 1 in HotelReservation (b) Type 2 in HotelReservation (c) Type 3 in HotelReservation (d) Type 4 in SocialNetwork

Fig. 6. Four metastability failure types [33] demonstrated in the Blueprint versions of DSB HotelReservation and DSB SocialNetwork.

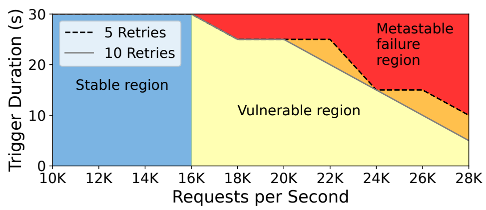


Fig. 7. Metastability Vulnerability Analysis for HotelReservation.

occur when delays in synchronization of replicated data stores such as databases and caches cause read requests for an object to return incorrect results. By default SocialNetwork has no replication, so we add replication to `userTimelineDatabase` and `UserTimelineService`, and modify the `GatewayService` to use the replicated version of the service. These changes only touch 4 LoC in the wiring spec.

We compare the replication-enabled SocialNetwork to the initial Blueprint SocialNetwork. We use a 100 Rps workload consisting of 100 % `ComposePost` requests. For each successful request, we read the user timeline of the post creator after a wait time ranging between 0 s and 1 s at 100 ms intervals. A response without the new post is a cross-system inconsistency. Fig. 8 shows the measured fraction of inconsistencies with increasing wait times for the original SocialNetwork and the replicated version. As expected, the non-replicated version always reads consistent results, whereas the replicated version incurs a small fraction of inconsistencies [12].

Overall, this case study demonstrates how Blueprint applications are amenable to change and enable users to modify existing applications to reproduce emergent phenomena with minimal effort.

6.3 UC3: Prototyping Improvements

In this section we evaluate how amenable Blueprint is for supporting prototyping and evaluation of improvements in two ways: (i) reproducing the prototyping required for a solution performed in existing research; and (ii) prototyping a new solution for an emergent phenomenon.

Reproducible Research. To understand how Blueprint can aid researchers in making changes, we select a mutation that was manually added by researchers to an existing microservice application, and reproduce that mutation in the Blueprint implementation of the application. Concretely, Sifter [40] manually mutates the DSB Social Network application to add X-Trace [23]. X-Trace is a distributed tracing framework, but it does not conform to OpenTelemetry APIs and cannot reuse the existing Jaeger instrumentation of DSB SocialNetwork. Consequently, the authors of Sifter manually extended DSB SocialNetwork to add X-Trace support. This comprised 1,289 LoC changed over a 2 week period, based on commit timestamps.

We implemented the same change in Blueprint, which required (1) extending Blueprint's compiler to support X-Trace (a 1-time task); and (2) enabling X-Trace for the Blueprint SocialNetwork application. The latter required 3 LoC change to the wiring spec of the SocialNetwork application. This reduction occurred because the vast majority of code to support X-Trace is templatable scaffolding code which can be easily incorporated in the compiler. Implementing X-Trace within Blueprint's compiler required 433 LoC.

To evaluate if Blueprint's modifications to systems yield the same results as the modifications to original systems, we contacted the authors of Sifter to obtain their experiment code and reproduced Figure 6 from the Sifter paper [40] using the Blueprint-generated SocialNetwork application. In the original experiment, the authors recorded the loss and sampling probability for a sequence of 1000 ComposePost API requests, and at five separate instances they had inserted anomalous requests. Similarly, in our experiment we generated anomalous requests with the Blueprint generated DSB-SN and repeated the above experiment. In Fig. 9, the spikes of high probability of selection correspond to the anomalous requests. This shows that Blueprint generated systems can reproduce the same results as the original systems.

Prototyping New Solutions. In this experiment we demonstrate how Blueprint can help in prototyping and integrating novel solutions in applications. We particularly focus on the Type 1 Metastable Failure first introduced in §6.2.

To address the Type 1 Metastable Failure, we prototype a solution based on *circuit-breakers*. A circuit-breaker prevents new requests from being sent if the moving-average failure rate of requests is higher than a specified threshold. We

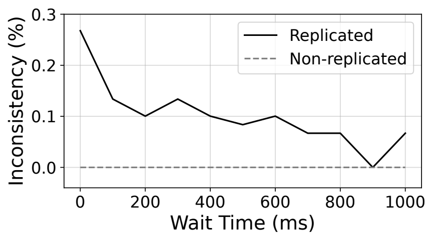


Fig. 8. Cross-system inconsistencies arise in SocialNetwork when enabling replication.

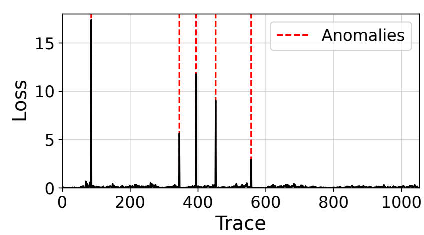


Fig. 9. Blueprint's reproduction of Fig. 6 from the Sifter paper [40] in SocialNetwork.

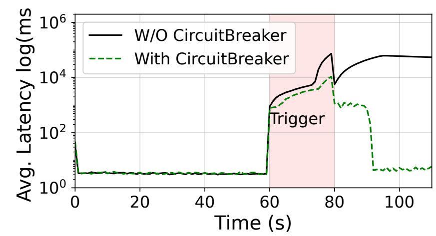


Fig. 10. Prototype Solution for Type 1 Metastability failure in HotelReservation.

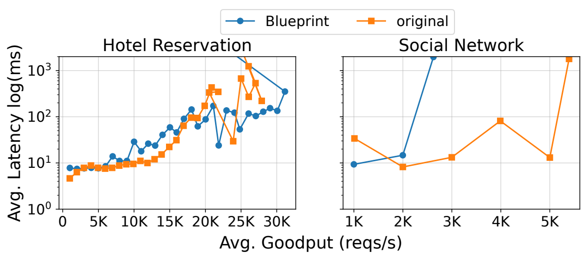


Fig. 11. Performance comparison of Blueprint systems with their original counterparts

Backend	Interface	Compiler
Cache	12	0
NoSQLDB	27	0
RelDB	22	0
Queue	12	0
Tracer	45	0
Deployer	3	46
RPC	11	152
HTTP	11	146

Tab. 2. Lines of Code required for adding a backend interface.

implement a circuit-breaker feature and introduce it to Blueprint's compiler as a new type of scaffolding. This required 126 LoC. Enabling circuit breakers for the HotelReservation application required only 2 LoC change to its wiring spec.

Fig. 10 shows how adding circuit breakers can potentially prevent the system from going into a metastable failure state. The circuit-breaker-enabled version of the application experiences the same latency increase at $t = 60$, but due to the circuit-breaker triggering, the application is able to avoid entering a metastable failure state and returns to normal at $t \approx 90$. Overall, this example demonstrates how Blueprint can be useful in prototyping novel solutions for phenomena.

6.4 Generating Realistic Systems

We now evaluate Blueprint's ability to generate real systems useful for meaningful evaluation. For this we measure and compare the performance of two of the Blueprint-generated systems applications above to that of the original systems.

HotelReservation. We compare the performance of the Blueprint implementation of HotelReservation to the original DSB implementation. The original DSB HotelReservation application is implemented in Go, enabling a direct comparison between it and the Blueprint-generated application. To exercise the systems, we run a mixed workload of 60% `hotels`, 38% `recommendations`, 1% `user`, and 1% `recommend` requests. We run the workload for different request rates ranging from 1 Krps to 30 KRps for a duration of 1 min each. Fig. 11 shows the latency-throughput profile of both systems. The Blueprint generated system has a similar performance to the original manually implemented system.

SocialNetwork. We also compare the performance of the Blueprint implementation of SocialNetwork to the original DSB implementation. The original DSB implementation uses a mix of C++ and Lua with an nginx gateway web server whereas the Blueprint implementation uses Go's default HTTP web server. To exercise both systems, we run a mixed workload of 60% `ReadHomeTimeline`, 30% `ReadUserTimeline`, and 10% `ComposePost` requests. We run the workload for different request rates ranging from 1 KRps to 6 KRps for a duration of 1 min each. Fig. 11 shows the latency-throughput profile for both systems under the aforementioned workload. In this scenario, the original system outperforms the Blueprint generated system. We attribute this to two compounding factors. First, the original system is implemented in C++ whereas the Blueprint version is in Go. Go incurs garbage collection overhead and relies on different libraries for many core microservices building blocks. Second, the Blueprint implementation does not use any Redis-specific specialized array operations. This illustrates a drawback of Blueprint– it requires interacting with backends through common interfaces.

Overall, these results illustrate that Blueprint can generate microservice systems whose performance compares closely to that of handwritten systems.

6.5 Extensibility of Blueprint

Adding backends and instantiations. Tab. 2 shows the LoC in the interface of various Blueprint backends and

Type	Instantiation	Impl	Compiler
Cache	Redis	76	140
Cache	Memcached	76	142
NoSQLDB	MongoDB	288	140
RelDB	MySQL	91	140
Queue	RabbitMQ	50	111
Tracer	Jaeger	28	145
Tracer	Zipkin	28	145
Deployer	Docker	74	0
Deployer	Kubernetes	45	0
Deployer	Ansible	439	0
RPC	GRPC	673	0
RPC	Thrift	636	0
HTTP	Go's net/http	271	0

Tab. 3. Lines of Code required for adding a new instantiation.

Plugin	Compiler (LoC)	Stdlib (LoC)
Retry	123	0
Tracing	284	45
p-Replication	52	0
ClientPools	145	55
X-Trace	364	69
CircuitBreaker	126	0
LoadBalancer	208	19

Tab. 4. Lines of code required for adding a new compiler plugin.

Tab. 3 shows the LoC for the instantiations currently available for each backend. Adding a new backend generally requires <100 LoC for implementing the lightweight interface. Adding an instantiation also requires a small amount of code (<200 LoC) in the compiler. Each of these is a one-time effort, that can be leveraged by subsequent Blueprint applications.

Instantiations of certain backends require more code than others. For example, instantiations of the RPC and backends require more code than average as these instantiations must correctly generate the code for communicating over the network, establishing connections, and running servers.

Adding new plugins. Blueprint's modifier abstraction allows developers to add new scaffolding. Tab. 4 shows the plugins currently available in Blueprint and the LoC in their implementations. Some plugins require changes only in the Blueprint toolchain whereas others also require an additional runtime library component. The LoC vary across features from 46 to around 440.

Adding new plugins is harder than directly implementing the scaffolding hard-coded within an application. Nonetheless, this addition is a one-time cost that amortizes the effort of reimplementing scaffolding in all later systems.

6.6 Cost of Blueprint

Compilation time. Tab. 5 shows the time taken by Blueprint to generate systems. Blueprint can generate small to medium sized system within seconds. As there are no existing large open-source microservice systems, we generated a large-scale microservice application using the Alibaba service topology in the Alibaba trace dataset [44]. For this, we omitted the caches and databases and only focused on stateless services. In total, the resulting workflow and wiring spec had 2.8K service instances. To generate a variant implementation, Blueprint took around 12 minutes. Overall, the compilation time is proportional to the number of service instances in the wiring spec and the density of the service topology. These results demonstrate that Blueprint enables researchers and practitioners to quickly (re-)compile applications, thus supporting rapid Configure, Build, and Deploy cycles. These results may be further improved through compiler optimizations and incremental compilation.

System Name	Gen Time(s)	Num Services
DSB-SN	1.172	28
DSB-MM	1.698	33
DSB-HR	1.281	18
TrainTicket	3.723	67
SockShop	0.925	14
Alibaba-TraceSet	707	2882

Tab. 5. Time taken by Blueprint for generating the system including services, caches, databases, queues, tracers

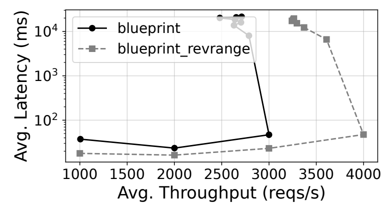


Fig. 12. DSB-SN cache choice exploration

Cost of abstractions. For each type of backend supported by Blueprint, services interact with the backend through a generalized interface. The interface is selected such that backend instances can be opaquely reconfigured. Yet many backends do provide broader interfaces with specialized operations that are more efficient for certain workloads. Blueprint's approach discourages services from using such operations, in the interest of reconfiguration.

To demonstrate the impact of using more general but less efficient APIs, we implement an extended Cache interface that provides access to Redis' specialized cache operations. We use this extended interface in the SocialNetwork application, we execute a `100% ReadHomeTimeline` workload for 1 minute for request rates ranging from 1 KRps to 6 KRps. With the extended interface, the application observes a 33% increase in throughput as shown in Fig. 12.

7 Discussion

Debugging. Debugging generated code and the workflow specification can be a challenging task. To aid developers in debugging workflow specification code, Blueprint provides default implementations of the various components called *null implementations*. These implementations provide a basic interface against which the core application specification can be tested without worrying about the deployment of the system. These implementations are attached to the workflow specification in the wiring spec and can be iteratively replaced with the real choices once the developer is confident in the correctness of their application code.

Language Heterogeneity. Usually, microservice systems contain services implemented in different languages. Currently, Blueprint generates services only in Go. Generating services in other languages is purely an implementation challenge and we believe that the current design can be extended to generate code in other languages. This would require compiler plugins to support generation of scaffolding in multiple languages in order to correctly mix code of different languages in an application.

Generating Production Systems. Blueprint targets microservice experimentation and prototyping use cases, rather than generating production-ready microservice applications. It remains an open question whether Blueprint is a suitable toolchain for developing production-ready microservice applications. Currently, Blueprint's approach stands at odds with the microservices architectural approach: microservices are typically developed by highly distributed teams operating independently; yet Blueprint imposes a centralized configuration and deployment step through its wiring spec. We are currently exploring approaches to decomposing and distributing the wiring spec. A further concern is the cost of Blueprint's abstractions for backends, which might be too high a price to pay for production systems. However, we believe that Blueprint could be used in production to quickly home in on a concrete set of choices that satisfies the developer's requirements. The Blueprint-generated system could then further be fine-tuned manually to obtain a production-grade system.

8 Related Work

Microservice benchmark suites have gained considerable popularity in recent years [26, 66, 71, 76]. Each system corresponds to only one concrete point in the design space and as they were not designed to be configurable, they are inadequate for tasks that would require exploring the design space of microservices.

Several existing tools support generation of microservice systems by providing a DSL or some other programming language [4, 21, 27, 38, 49, 53, 56–59, 62–64, 67, 68, 73].

However, the tools are designed for one-shot generation of microservices and select a single dimension to provide flexibility. The most prominent example is Google's ServiceWeaver [27], which provides flexibility along the deployment dimension allowing users of the tool to deploy the same application as a suite of microservices or as a monolithic application. However, like the other tools, ServiceWeaver is a poor fit CBD use cases that require modifying the application along dimensions other than the deployment dimension.

Some existing tools, such as SpringBoot [65] or Dapr [1] aid in developing microservice systems by separating out reusable infrastructure components from the core implementation of the application, allowing users to select infrastructure building blocks that can be applied to an application. However, the SpringBoot framework makes binding choices along the deployment and communication dimensions of the microservice design space making SpringBoot a poor fit for CBD use cases. Unlike the SpringBoot framework, Dapr allows users to switch between the deployment and communication dimensions but the user must change them manually resulting in high manual effort.

Two of the key features that enables Blueprint to quickly reconfigure applications are the notions of reusability and dependencies. First, the idea of reusability has been inspired from the The Flux OSKit [24]. Similar to Flux OSKit's suite of reusable OS components, Blueprint also uses commonly used microservice libraries and components as reusable building blocks for microservice applications. In contrast, Blueprint does not require glue code to be handwritten for each component for each new application, instead relying on its compiler abstractions and code generation to correctly handle composition of components. Second, Blueprint uses Dependency Injection [54] to correctly generate applications by not allowing instantiation of dependent components in the workflow specification of applications. The dependencies of any given service in a Blueprint application are generated at compilation time.

9 Conclusion

We have introduced Blueprint, a toolchain for developing highly reconfigurable microservice applications. We have demonstrated Blueprint for several use cases that involve configuring, building, and deploying variants of microservice applications with modified designs. Compared to existing benchmark applications, Blueprint substantially eases development and reconfiguration by providing a strong separation of concerns between an application's workflow, scaffolding infrastructure, and implementation choices. Blueprint is open-source, and we hope that its adoption will make it easier for future work to understand and improve the performance and correctness guarantees of microservice systems. Blueprint is available at https://gitlab.mpi-sws.org/cld/blueprint/blueprint-compiler.

Acknowledgements

We would like to thank our shepherd, Rebecca Isaacs and our anonymous reviewers for helping us shape up the final version of the paper. We would also like to thank and acknowledge the efforts of the anonymous evaluators of the Artifact Evaluation Committee. We would like to thank Gerd Alliu, for helping with an initial Blueprint versions of the TrainTicket and SockShop applications. We would also like to thank Rodrigo Fonseca, Roberta De Viti, and Matheus Stolet for providing us with feedback.

References

[1] Dapr: Distributed application runtime. https://dapr.io/.

[2] AcmeAir. Acmeair. https://github.com/acmeair.

[3] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[4] I. K. Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan. A model-driven architecture for automated deployment of microservices. *Applied Sciences*, 11(20):9617, 2021.

[5] V. Anand and J. Mace. Deathstarbench - modified with x-trace. https://github.com/JonathanMace/DeathStarBench/.

[6] asc lab. Asclab micronaut poc - lab insurance sales portal. https://github.com/asc-lab/micronaut-microservices-poc/.

[7] Azure. Retry storm antipattern. Retrieved September 2022 from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/, 2021.

[8] Azure. Chatty i/o. Retrieved September 2022 from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/, 2022.

[9] Azure. No caching antipattern. Retrieved September 2022 from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/no-caching/, 2022.

[10] Azure. Noisy neighbour. Retrieved Septemeber 2022 from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor, 2022.

[11] Azure. Technology choices for azure solutions. Retrieved September 2022 from https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/technology-choices-overview, 2022.

[12] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[13] Barber0. Deathstarbench no tracing. Retrieved August 2022 from https://github.com/Barber0/DeathStarBench/tree/no_tracing, 2021.

[14] R. Blum and R. Singh. Data integrity: What you read is what you wrote. Retrieved September 2022 from https://sre.google/sre-book/data-integrity/.

[15] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 221–227, 2021.

[16] A. Cockcroft. The evolution of microservices. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[17] A. Cockcroft. Microservices workshop: Why, what, and how to get there. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, 2016.

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] delimitriou. Deathstarbench social network application refactor for better performance. Retrieved August 2022 from https://github.com/delimitrou/DeathStarBench/pull/101, 2021.

[20] M. Dinga. An empirical evaluation of the energy and performance overhead of monitoring tools on docker-based systems. Retrived August 2022 from https://github.com/MadalinaDinga/thesis-2022-monitoring-tools-integration-with-train-ticket-replication-package/, 2022.

[21] 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*, pages 171–172, 2017.

[22] J. Faro. Deathstarbench metastability failure fork. Retrieved August 2022 from https://github.com/jfaro/DeathStarBench, 2022.

[23] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[24] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.

[25] Y. Gan, S. Dev, D. Lo, and C. Delimitrou. Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices. In *Workshop on ML for Computer Architecture and Systems (MLArchSys)*, June 2020.

[26] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[27] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

[28] GoogleCloudPlatform. Online boutique, fka hipstershop. https://github.com/GoogleCloudPlatform/microservices-demo.

[29] E. Haddad. Service-oriented architecture: Scaling the uber engineering codebase as we grow. (September 2015). Retrieved October 2020 from https://eng.uber.com/service-oriented-architecture/, 2015.

[30] M. Hashemi. (January 2017). Retrieved February 2021 from https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html, 2017.

[31] G. Heiser. System benchmarking crimes. https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html.

[32] Hilbert-Yaa. Deathstarbench local tracing. Retrieved August 2022 from https://github.com/Hilbert-Yaa/DeathStarBench/tree/proto-tracing, 2021.

[33] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[34] L. Huang and T. Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021.

[35] D. Huye, Y. Shkuro, and R. R. Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[36] istio. Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/.

[37] ivanium. Deathstarbench social network no tracing. Retrieved August 2022 from https://github.com/ivanium/DeathStarBench/commit/01360df6653e12982335838c877cc4178a518987, 2021.

[38] jhipster. Jhipster. Retrieved August 2022 from https://github.com/jhipster/generator-jhipster, 2019.

[39] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[40] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[41] A. Lesniak, R. Laigner, and Y. Zhou. Enforcing consistency in microservice architectures through event-based constraints. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 180–183, 2021.

[42] J. Loff. Deathstarbench inconsistency. Retrieved August 2022 from https://github.com/jfloff/antipode-deathstarbench, 2022.

[43] J. Loff, D. Porto, C. Baquero, J. Garcia, N. Preguiça, and R. Rodrigues. Transparent cross-system consistency. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2017.

[44] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[45] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 589–603, 2015.

[46] A. Mcdade. Significant outage for amazon web services stalls netflix, delta airlines, others. Retrieved September 2022 from https://www.newsweek.com/significant-outage-amazon-web-services-stalls-netflix-delta-airlines-others-1657077, 2021.

[47] microservices demo. Retrieved August 2022 from https://github.com/microservices-demo/microservices-demo, 2016.

[48] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[49] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[50] S. Needleman. Amazon outage disrupts lives, surprising people about their cloud dependency. Retrieved September 2022 from https://www.wsj.com/articles/amazon-outage-disrupts-lives-surprising-people-about-their-cloud-dependency-11638972001, 2021.

[51] L. Nolan. Why health checks are like sidewalks. (December 2022). Retrieved April 2023 from https://www.usenix.org/publications/loginonline/why-health-check-sidewalk, 2022.

[52] J. Park, B. Choi, C. Lee, and D. Han. Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[53] K. Perera and I. Perera. A rule-based system for automated generation of serverless-microservices architecture. In *2018 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–8. IEEE, 2018.

[54] D. R. Prasanna. Dependency injection. 2009.

[55] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 805–825, 2020.

[56] F. Rademacher, S. Sachweh, and A. Zündorf. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 IEEE International conference on software architecture (ICSA)*, pages 21–30. IEEE, 2019.

[57] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf. Towards a viewpoint-specific metamodel for model-driven development of microservice architecture. *arXiv preprint arXiv:1804.09948*, 2018.

[58] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf. Specific model-driven microservice development with interlinked modeling languages. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 57–5709. IEEE, 2019.

[59] F. Rademacher, J. Sorgalla, P. N. Wizenty, S. Sachweh, and A. Zündorf. Microservice architecture and model-driven development: Yet singles, soon married (?). In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–5, 2018.

[60] M. R. S. Sedghpour, C. Klein, and J. Tordsson. Service mesh circuit breaker: From panic button to performance management tool. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, pages 4–10, 2021.

[61] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson. {FlightTracker}: Consistency across {Read-Optimized} online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423, 2020.

[62] J. Sorgalla. Ajil: A graphical modeling language for the development of microservice architectures. In *Extended Abstracts of the Microservices 2017 Conference*, 2017.

[63] J. Sorgalla, F. Rademacher, S. Sachweh, and A. Zündorf. Model-driven development of microservice architecture: An experiment on the quality in use of a uml-and a dsl-based approach. 2020.

[64] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):1–25, 2021.

[65] spring.io. Spring boot. Retrieved August 2022 from https://spring.io/projects/spring-boot.

[66] A. Sriraman and T. F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[67] A. Suljkanović, B. Milosavljević, V. Indjić, and I. Dejanović. Developing microservice-based applications using the silvera domain-specific language. *Applied Sciences*, 12(13):6679, 2022.

[68] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, and I. Luković. Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018.

[69] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G. E. Boussaidi, J. Privat, and Y.-G. Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–13, 2020.

[70] M. Ulrich. Addressing cascading failures. Retrieved September 2022 from https://sre.google/sre-book/addressing-cascading-failures/.

[71] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[72] S. Waterworth. Stan's robot shop – a sample microservice application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/, 2018.

[73] P. Wizenty, J. Sorgalla, F. Rademacher, and S. Sachweh. Magma: Build management-based generation of microservice infrastructures. In *Proceedings of the 11th European conference on software architecture: companion proceedings*, pages 61–65, 2017.

[74] xiling42. Zl-ldfi with train ticket. Retrieved September 2022 from https://github.com/xiling42/train-ticket-LDFI, 2020.

[75] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[76] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Poster: Benchmarking microservice systems for software engineering research. In

2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 323–324. IEEE, 2018.

A Appendix: Hierarchical Code Generation

In this section, we expand upon the source code generation process of Blueprint's toolchain, Blueprint. Recall that the service instance of the `ComposePostService` in the IR was modified with two modifiers - `ZipkinTracer` and `GRPC`. As a result, Blueprint generates two different classes that wrap around the workflow specification from Fig. 1. Fig. 13 shows the generated code. Blueprint generates code in a hierarchical fashion. As tracing must happen before the start of the actual processing of the function call, Blueprint wraps the workflow specification of the service with a generated object that starts a new span every before calling into the function. Subsequently, as the next level in the hierarchy is the GRPC node, Blueprint wraps the tracing code with GRPC-specific code and creates a function to start the GRPC server. The next node in the hierarchy, according to the IR, is the process node. Fig. 14 shows the process code generated for the service. In lines 2-15, the client objects are constructed and then passed as parameters to the constructor for the `ComposePostService` defined in Fig. 1. The tracer and grpc objects for the service are then constructed and the server is started. This hierarchical code generation pattern repeats for every service instance defined in the wiring specification of the application.

B Phenomena, Systems Usage, & Modifications

B.1 Known Emergent Phenomena

In this section, we list and define known phenomena. While, our list of phenomena is by no means exhaustive, the list is detailed enough to provide an insight into the kinds of phenomena exhibited by microservice systems and what are the requirements for a system to exhibit phenomena. We hope that researchers find this information useful while using Blueprint to modify systems to understand this phenomena.

Retry storm metastable failure. in which a system is pushed into a prolonged failure state due to repeated RPC retries and timeouts [7, 15, 33]. Retry storms arise when the timeout configuration of services are too low, causing RPCs to to be aborted part-way through execution, wasting work. The system must be modified such that each service has timeouts as well as retry mechanisms where a service generates a retry for a request on failure.

Load Spike Resource Contention metastable failure. This is type 2 of the four types of metastability failures [33]. In this phenomenon, some bookkeeping function competes with the actual job for processor resources. For example, a sudden spike in the load could trigger frequent Garbage Collection(GC) cycles which compete with actual job processing for processor resources leading to a build up of unprocessed requests leading to a cascade of backed up requests. For a system to exhibit this phenomenon, the system must have some kind of an auxiliary bookkeeping subroutine – 100% sampling for traces, garbage collection.

Low-Level Resource Contention metastable failure. This is type 3 of the four types of metastability failures [33]. In this phenomenon, low-level resource contention outside the scope of the application causes the system to slow down. For example, co-location of two processor intensive services might cause those services to drastically impact each other. For a system to exhibit this phenomenon, the system should additionally have timeouts and retries to trigger a cascading effect so that the system stays in a metastable state.

Low-Level Resource Contention QoS violations. Similar to the previous phenomenon, this phenomenon manifests as QoS (Quality of Service) violations caused due to some low-level resource contention. However, in this phenomenon, the violations only last a short duration and the system does not enter into a metastable state. For a system to exhibit this phenomenon, the system needs some kind of low-level resource contention that violates a QoS metric. An example would be FIRM's anomaly injector [55] that injects low level anomalies to cause QoS violations.

Capacity Degradation Trigger Capacity Degradation Amplification metastable failure, This is type 4 of the four types of metastability failures [33]. In this phenomenon, a trigger decreases the capacity of the system to process requests which causes failures leading to the capacity of the system never reverting back to the normal state. For example, a system with a cache in front of a database can process 2 Krps but if the cache fails then the capacity of the system to process requests decreases. This can cause failures and prevent the cache from ever getting filled back and can overload the database. For a system to exhibit this phenomena, the system needs two different paths - a fast path and a slow path and timeouts and retries. Momentary failure of the fast path could lead to timeouts causing a retry storm and putting the system into a metastable failure state.

Cascading QoS violations in which application quality of service deteriorates due to increased latency and/or failures in downstream services [52, 75]. This is part of a broader class of *cascading effects* where failures in one component drastically combine with error-handling and availability mechanisms in another component to ultimately bring down the system [70]. For a system to exhibit this phenomenon, there must be a chain of services of significant length as well as timeouts and retries.

QoS violations from repeated expensive I/O. In this phenomenon, repeated execution of expensive I/O can lead to QoS violations. For example, repeatedly fetching the same information from a resource that is expensive to access, in terms of I/O overhead or latency and not caching the data can cause QoS violations [9]. Another example would be services not re-using clients while making requests to other

```go
type ComposePostTracer struct {
 service *ComposePostImpl
 tracer blueprint.Tracer
 service_name string
}

func NewComposePostTracer(service *ComposePostImpl,tracer blueprint.
      Tracer,service_name string) *ComposePostTracer {
 return &ComposePostTracer{service: service, tracer: tracer,
      service_name: service_name}
}

func (t *ComposePostTracer) ComposePost(ctx context.Context, reqID
      int64, username string, userID int64, text string, mediaIDs []
      int64, mediaTypes []string, post_type services.PostType,
      zipkinTracer_trace_ctx string) error {
 if zipkinTracer_trace_ctx != "" {
  span_ctx_config, _ := blueprint.GetSpanContext(
      zipkinTracer_trace_ctx)
  span_ctx := trace.NewSpanContext(span_ctx_config)
  ctx = trace.ContextWithRemoteSpanContext(ctx, span_ctx)
 }
 tp, _ := t.tracer.GetTracerProvider()
 tr := tp.Tracer(t.service_name)
 ctx, span := tr.Start(ctx, "ComposePost")
 defer span.End()
 err := t.service.ComposePost(ctx, reqID, username, userID, text,
      mediaIDs, mediaTypes, post_type)
 if err != nil {
  span.RecordError(err)
 }
 return err
}
```

(a) Generated code for adding tracing to ComposePostService

```go
type ComposePostGRPC struct {
 service *ComposePostTracer
 socialnetwork.UnimplementedComposePostServiceImplServer
}

func NewComposePostGRPC(old_handler *ComposePostServiceImpl) error {
 addr := os.Getenv("composePostService_ADDRESS")
 port := os.Getenv("composePostService_PORT")
 if addr == "" || port == "" {
  return errors.New("Address or Port were not set")
 }
 lis, err := net.Listen("tcp", addr + ":" + port)
 if err != nil {
  return err
 }
 handler := &ComposePostGRPC{service:old_handler}
 grpcServer := grpc.NewServer()
 socialnetwork.RegisterComposePostServiceImplServer(grpcServer,
      handler)
 return grpcServer.Serve(lis)

}

func (rpchandler *ComposePostGRPC) ComposePost(ctx context.Context,
      request *socialnetwork.
      ComposePostServiceImpl_ComposePostRequest) (*socialnetwork.
      BaseRPCResponse, error) {
 var arg7 services.PostType
 copier.Copy(&arg7,request.PostType)
 ret0 := rpchandler.service.ComposePost(ctx,request.ReqID,request.
      Username,request.UserID,request.Text,request.MediaIDs,request.
      MediaTypes,arg7,request.ZipkinTraceCtx)
 response := &socialnetwork.BaseRPCResponse{}
 return response,ret0
}
```

(b) Generated code for runnin ComposePostService as a GRPC server

Fig. 13. Generated Server side code for the ComposePostService defined in §4

```go
func RuncomposePostService() error {
    zipkin := NewZipkinTracer()
    userserviceimplrpcclient_netclient, err :=
      NewUserServiceGRPCClient()
    if err != nil {
        return err
    }
    userservicetracerclient := NewUserServiceZipkinClient(zipkin,
      userserviceimplrpcclient_netclient)
    userserviceimplclient := NewUserServiceClient(
      userservicetracerclient)

 poststorageserviceimplrpcclient_netclient, err:=
      NewPostStorageServiceGRPCClient()
 if err != nil{
        return err
    }
 poststorageserviceimpltracer :=
      NewPostStorageServiceZipkinClient(zipkin,
      poststorageserviceimplrpcclient_netclient)
 poststorageserviceimplclient := NewPostStorageServiceClient(
      poststorageserviceimpltracer)
 spec_handler := NewComposePostImpl(poststorageserviceimplclient,
       userserviceimplclient)
    composeposttracer := NewComposePostZipkin(spec_handler)
 err_new := NewComposePostServiceImplHandler(composeposttracer)
 if err_new != nil {
  return err_new
 }
 return nil
}
```

Fig. 14. Generated Server side main function for the ComposePost-Service defined in §4

services and re-establishing a connection on every call. For a system to exhibit this phenomenon, the system must perform an expensive operation in a repeated fashion either in the workflow spec or in the scaffolding.

Reduced Availability due to saturation. Services can get saturated and overwhelmed when placed under significant load. However, even in the absence of retry storms, a service can get oversaturated. In this phenomenon, only one service is impacted while the rest of the system remains healthy. This can happen in multiple scenarios. One such scenario is if the service has a high fan in number *i.e.* multiple services call into this service. Another scenario is when one big request is broken down into multiple smaller requests which can fill up queues at the called service [8]. DSB-SN v1 suffered from this exact phenomenon via a combination of the above scenarios where multiple services called `ComposePostService` and the items to ComposePostCache were broken down into multiple pieces and accessed via multiple set/get requests. The application was changed to remove this phenomenon [19].

Reduced Availability due to QoS violation prevention schemes. In this phenomenon, application availability is impacted by mechanisms such as circuit breakers [60] in place to prevent QoS violations at others services. This incurs a high error rate or high end-to-end latencies to prevent QoS violations at others services. For a system to exhibit this phenomenon, the system must have QoS prevention mechanisms in different parts of the system.

Cross-System Inconsistency. This phenomenon occurs when requests write to multiple eventually-consistent datastores, establishing an ordering for read operations across those datastores. Each datastore is individually consistent, yet readers may experience inconsistent reads, such as a notification arriving before post content has finished replicating [3]. For a system to exhibit this phenomenon, the system must at least have replicated datastores. For a stronger manifestation of this phenomenon, geo-replication is recommended [42].

User-View Data Inconsistency. This phenomenon is a less stricter version of Cross-System Inconsistency. For a Cross-System Inconsistency, the replicated datastore is inconsistent from the viewpoint of the system itself such that a value that was committed in the datastore has not propagated to all replicas. In contrast, for a user-view data inconsistency, the value does not have to be committed to a datastore; rather, it must *seem* to be committed from the user but the update has not reached all parts of the system due to asynchrony [41]. Prime example of this phenomenon is observed almost on a daily basis by users of social-media applications where the user gets a notification but clicking on the notification does not show the update simply because the update is being applied asynchronously as the data policy is of eventual consistency. The key to replicating this phenomenon is to have asynchronous updates to datastores leading to inconsistent view of the system from the point of view of the user. This phenomenon falls under the larger class of phenomena related to Data Integrity violations [14].

QoS violations from excessive repeated calls. In this phenomenon, repeatedly contacting a datastore or a service for the same piece of information can overwhelm the downstream service resulting into a cascading failure. For a system to exhibit this phenomenon, it must make repeated calls for the same data and must not use a cache [9].

QoS violations from excessive data demand. This is the dual of the previous phenomenon. In this phenomenon, contacting a service for more data than required can cause the downstream service to get blocked as it now needs to process and return a large amount of data. This can also create contention on the network and bog down concurrent requests. For a system to exhibit this phenomenon, the workflow spec must have a service that asks for excessive data.

Performance Degradation due to Disruptive neighbours. In this phenomenon, the performance of one application/-tenant is drastically impacted by the co-located application/-tenant due to the co-tenant using a disproprtionate amount of resources [10, 45]. This scenario only happens in situtations when multiple applications are co-located on the same node and compete for resources.

The Laser of Death. In this phenomenon, the availability of the system is hampered by a load balancer overloading replicas of a service [51]. Specifically, a load balancer performs periodic health checks to find a healthy subset of replicas and routes requests to this healthy subset. If one of the replicas gets overloaded then its health check would fail which would cause the load balancer to stop routing requests to that replica. This in turn can overload the new subset of healthy replicas as each healthy replica now has to handle an increasing workload. Essentially, the load balancer begins acting as a Laser of Death by overloading healthy replicas due to health checks. This scenario only happens in applications with replicated service instances that are fronted by a load balancer making routing decisions based on health checks of instances.

The Killer Health Check. In this phenomenon, health check based instance replacement automation might cause warmed up hosts to get replaced which can prevent the application from servicing requests at the same rate [51]. The underlying cause of this is the fact that the health check signal does not actually specify if a problem is instance-specific. The problem might exist in some downstream services or might be a problem that is affecting all services. This scenario can only happen in applications with replicated service instances that allow for automatic creation and deletion of instances based on health check signals.

Incorrect Microservice Granularity. This is a phenomenon pertinent exclusively to the hierarchy and grouping of microservices. Mega Microservices can suffer the same problems that monolith applications suffer whereas Nano Microservices can incur a high development and maintenance and can lead to cycles [69]. The correct granularity of microservices has also been of interest to practitioners. This interest led to Uber switching to a less granular microservice architecture in which microservices are grouped together into domains [167].

Multi-version Errors. This is a phenomenon in which multiple versions of the same microservice co-exist and are deployed at the same time [69]. This can cause a lot of errors if the requests meant for service:v1 end up at service:v2. For a multi-version error to occur, the system could have service discovery mechanisms that link callers to incorrect version due to stale information. Another way of exhibiting this phenomena in this system is to have the same service instance simultaneously service requests from both versions with clients making calls to version 1 with data for version 2 of the function.

B.2 Existing Use of Microservice Systems

To understand how microservice systems are used by researchers, we conducted a literature survey to analyze which systems were popular. To find the necessary papers, we

System	Count	Papers
DSB-SN [26]	51	[34, 40, 52, 55, 82, 84, 85, 95, 102, 103, 107, 108, 126, 129, 133, 149, 156, 157, 160, 162, 191, 196, 201–203, 206, 209, 228, 252, 270, 273, 276–278, 285–287, 304, 322, 323, 334, 351, 353, 371, 374–377]
DSB-MM [26]	18	[55, 103, 107, 108, 111, 131, 133, 156, 162, 209, 225, 252, 273, 286, 287, 305, 322, 337]
DSB-HR [26]	17	[55, 103, 126, 156, 162, 188–190, 205, 209, 217, 246, 302, 314, 375, 384, 388]
DSB-Swarm [26]	1	[163]
μSuite [66]	2	[253, 328]
TrainTicket [76]	52	[34, 55, 77, 81, 88, 95, 109, 110, 113, 127, 130, 131, 134, 137, 156, 177, 182, 188–192, 197, 203, 234, 235, 237, 239, 246, 247, 257, 296, 297, 304, 305, 311, 322, 329, 338, 339, 347–349, 351, 370, 378–381]
SockShop [47]	96	[83, 86, 87, 91, 93, 100, 105, 106, 114, 115, 117, 119, 121, 124, 134, 135, 139, 141, 146, 150, 155, 171, 172, 176, 178–183, 188–190, 204, 207–211, 215, 216, 218–222, 229–232, 237, 244, 246, 247, 249, 250, 254, 256, 258–262, 268, 271, 280, 281, 283, 286, 292, 294–296, 300, 301, 311, 313, 319–321, 331–333, 335, 336, 340, 343, 350, 354–357, 360–362, 365, 372, 380, 383]
AcmeAir [2]	26	[79, 96, 97, 104, 125, 136, 148, 173, 174, 195, 196, 204, 214, 255, 269, 289, 293, 310, 311, 330, 341, 342, 358, 363, 364]
TeaStore [71]	32	[92, 112, 129, 132, 143, 145–147, 154, 166, 168, 175–177, 184, 185, 187, 194, 198, 198, 213, 233, 248, 286, 303, 306, 307, 317, 318, 326, 344, 359]
HipsterShop/OnlineBoutique [28]	34	[52, 94, 95, 101, 116, 118, 120, 122, 123, 158, 159, 170, 186, 212, 217, 227, 231, 232, 237, 245, 250, 267, 290, 296–299, 302, 321, 366–369, 384]
AliBaba [44]	2	[242, 373]
Stan's RobotShop [72]	9	[52, 98, 123, 206, 236, 259, 266–268]
BookInfo [36]	12	[52, 80, 90, 98, 123, 231, 232, 236, 236, 243, 263, 296, 312]
eShopOnContainer [140]	13	[153, 164, 165, 200, 223, 247, 260, 265–268, 315, 324, 346]
LakeSideMutual [275]	10	[41, 64, 144, 164, 165, 266, 282, 315, 316, 385–387]
eShoppers [308]	3	[127, 128, 339]
PitStop [142]	2	[232, 264]
Lab Insurances Portal [6]	2	[206, 240]
Overleaf [272]	1	[206]
Retwis [288]	3	[163, 217, 309]

Tab. 6. Microservice systems used in the literature.

started with a list of seed microservice systems (DeathStarBench [26] and TrainTicket [76]) and shortlisted the papers citing the seed systems that we found using Google Scholar. During the analysis of the shortlisted papers, if we found a previously unseen system used by the paper then we modified our list of systems to include the newly found system. When we added a new system, we increased the shortlisted papers with the papers citing the new system. Some of the systems did not have an accompanying research paper so it was not straightforward to find the papers using those systems via Google Scholar. To overcome this, we used the url of the system as a query to Google Scholar search to find candidate papers.

To decide if a paper used a particular system, we analyzed the paper's evaluation section. If the paper performed at least one experiment using a given system, the paper was deemed to have used the system. In total, we found 290 research papers that altogether used 20 different systems. Six of the twenty systems, DSB-SN, DSB-HR, DSB-Swarm, DSB-MM, TrainTicket, and TeaStore, were described by the authors of the systems as microservice benchmarks, eleven systems were demonstration apps by a specific framework or a library to showcase some feature of their library or feature in the context of microservice systems, two systems were constructed based on public trace datasets, and the final system, Overleaf [272] was the sole microservice system that is currently deployed and used publicly. While our survey found 20 systems that are used by researchers in the literature, there exist a lot more open source microservice systems [284].

Tab. 6 shows the breakdown of papers found in the survey categorized by the open-source system they used. Out of all the used systems, SockShop [47] was used by the most number of papers despite its small number of services. We found it surprising that SockShop was used in more papers than any of the systems labeled as 'microservice benchmarks'. We believe that this is probably due to the fact that the system has been around for a longer period of time as compared to the benchmarks and is enabled with a plethora of orchestration and monitoring features that are desirable to researchers. This indicates that researchers want systems that have a variety of features.

While researchers use microservice systems to analyze and solve specific emergent phenomena, other use-cases of microservices include using them as sources of log and trace data, or as individual datapoints in a larger dataset of microservice systems for source code analysis. The use-cases of microservice systems are diverse and require a diverse set of systems that vary across a variety of dimensions. Thus, it is impossible that any single implementation of a microservice system can be used as the ideal benchmark system for all possible research scenarios.

System	Total Forks	Forks w/ Modifications
DeathStarBench	240	85
TrainTicket	130	25
TeaStore	96	36

Tab. 7. Modified Forks for each Microservice Benchmark

B.3 Modifications to open-source systems

In addition to the literature survey, we also analyzed forks of popular microservice systems to figure out what modifications were being made to the systems. We chose to anlayse the forks of the systems labeled as microservice benchmarks as these systems were designed as microservice benchmarks and are the ideal targets for researchers. We analyzed 240 forks of DeathStarBench, 130 forks of TrainTicket, and 96 forks of TrainTicket. Out of the total 464 forks, 146 forks made modifications to the benchmark systems. Tab. 7 shows the breakdown of the modified forks by system.

We found multiple different types of modifications made to the systems. The most common modification to systems were to either enable or disable tracing and other monitoring tools. This included switching on/off tracing, metrics, and logging. Another common modification was to generate the manifest files for deploying systems on a specific framework such as Kubernetes or OpenShift. There were multiple forks that modified systems such that they exhibited a specific emergent phenomena. We found one fork of DeathStarBench that modified the system to induce a metatstable failure [22, 152] into the system and another fork that modified the DeathStarBench system to include geo-replication to analyze cross-system inconsistencies [42]. Modifications were not restricted to emergent phenomena. Some other forks modified the systems to understand the impact of various of design choices- for eg, thrift vs grpc [99], energy efficiency of monitoring tools [20], among others. We also found a fork that modified TeaStore to study, reproduce, and exploit CVE-2021-44228 [291]. Researchers have also modified systems to support their experiments [5, 20, 74].

In conclusion, researchers modify systems for one main reason that can manifest as different types of modifications. We found that the reason for modifications was to include some feature that is absent from the system but is necessary for research.

Appendix References

[77] A. S. Abdelfattah. Student research abstract: Microservices-based systems visualization. *arXiv preprint arXiv:2207.12998*, 2022.

[78] AcmeAir. Acmeair. https://github.com/acmeair.

[79] S. Agarwal, R. Sinha, G. Sridhara, P. Das, U. Desai, S. Tamilselvam, A. Singhee, and H. Nakamuro. Monolith to microservice candidates using business functionality inference. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 758–763. IEEE, 2021.

[80] S. Allen, M. Toslali, S. Parthasarathy, F. Oliveira, and A. K. Coskun. Tritium: A cross-layer analytics system for enhancing microservice rollouts in the cloud. In *Proceedings of the Seventh International Workshop on Container Technologies and Container Clouds*, pages 19–24, 2021.

[81] N. Alshuqayran. *Static Microservice Architecture Recovery Using Model-Driven Engineering*. PhD thesis, University of Brighton, 2020.

[82] M. Amaral, T. Chiba, S. Trent, T. Yoshimura, and S. Choochotkaew. Microlens: A performance analysis framework for microservices using hidden metrics with bpf. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 230–240. IEEE, 2022.

[83] L. An, A.-J. Tu, X. Liu, and R. Akkiraju. Real-time statistical log anomaly detection with continuous aiops learning. In *CLOSER*, pages 223–230, 2022.

[84] V. Anand, M. Stolet, T. Davidson, I. Beschastnikh, T. Munzner, and J. Mace. Aggregate-driven trace visualizations for performance debugging. *arXiv preprint arXiv:2010.13681*, 2020.

[85] V. Anand and J. Wonsil. Tracey-distributed trace comparison and aggregation using nlp techniques.

[86] M. V. d. A. ANDRADE. Migração de microsserviços containerizados em tempo de execução. Master's thesis, Universidade Federal de Pernambuco, 2018.

[87] T. Angerstein, C. Heger, A. v. Hoorn, D. Okanović, H. Schulz, S. Siegl, and A. Wert. Continuity-automatisiertes performance-testen in der kontinuierlichen softwareentwicklung: Abschlussbericht. 2020.

[88] V. Arya, K. Shanmugam, P. Aggarwal, Q. Wang, P. Mohapatra, and S. Nagar. Evaluation of causal inference techniques for aiops. In *8th ACM IKDD CODS and 26th COMAD*, pages 188–192. 2021.

[89] asc lab. Asclab micronaut poc - lab insurance sales portal. https://github.com/asc-lab/micronaut-microservices-poc/.

[90] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 229–236, 2021.

[91] W. K. Assunção, J. Krüger, and W. D. Mendonça. Variability management meets microservices: six challenges of re-engineering microservice-based webshops. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–6, 2020.

[92] S. Athlur, N. Sondhi, S. Batra, S. Kalambur, and D. Sitaram. Cache characterization of workloads in a microservice environment. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 45–50. IEEE, 2019.

[93] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In *European Conference on Software Architecture*, pages 159–174. Springer, 2018.

[94] A. Aznavouridis, K. Tsakos, and E. G. Petrakis. Micro-service placement policies for cost optimization in kubernetes. In *International Conference on Advanced Information Networking and Applications*, pages 409–420. Springer, 2022.

[95] A. F. Baarzi and G. Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441, 2021.

[96] A. Baluta, J. Mukherjee, and M. Litoiu. Machine learning based interference modelling in cloud-native applications. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 125–132, 2022.

[97] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.

[98] G. C. Baptista et al. *Failure injection in microservice applications*. PhD thesis, Universidade de Coimbra, 2021.

[99] Barber0. Deathstarbench social network grpc. Retrieved August 2022 from https://github.com/Barber0/DeathStarBench/tree/sn_grpc.

[100] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano. Neptune: Network-and gpu-aware management of serverless functions at the edge. *arXiv preprint arXiv:2205.04320*, 2022.

[101] J. Berg, F. Ruffy, K. Nguyen, N. Yang, T. Kim, A. Sivaraman, R. Netravali, and S. Narayana. Snicket: Query-driven distributed tracing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 206–212, 2021.

[102] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020.

[103] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[104] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien. Migration from monolith to microservices: Benchmarking a case study. 2020.

[105] M. Bogo, J. Soldani, D. Neri, and A. Brogi. Fine-grained management of cloud-native applications, based on tosca. *Internet Technology Letters*, 3(5):e212, 2020.

[106] A. Brogi, A. Corradini, and J. Soldani. Estimating costs of multi-component enterprise applications. *Formal Aspects of Computing*, 31(4):421–451, 2019.

[107] R. Brondolin and M. D. Santambrogio. A black-box monitoring approach to measure microservices runtime performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.

[108] R. Brondolin and M. D. Santambrogio. Presto: a latency-aware power-capping orchestrator for cloud-native microservices. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 11–20. IEEE, 2020.

[109] V. Bushong, D. Das, A. Al Maruf, and T. Cerny. Using static analysis to address microservice architecture reconstruction. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1201. IEEE, 2021.

[110] V. Bushong, D. Das, and T. Cernỳ. Reconstructing the holistic architecture of microservice systems using static analysis. In *CLOSER*, pages 149–157, 2022.

[111] A. Byrne, S. Nadgowda, and A. K. Coskun. Ace: Just-in-time serverless software component discovery through approximate concrete execution. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 37–42, 2020.

[112] S. Caculo, K. Lahiri, and S. Kalambur. Characterizing the scale-up performance of microservices using teastore. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 48–59. IEEE, 2020.

[113] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 29–39, 2022.

[114] M. Camilli, A. Janes, and B. Russo. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software*, 187:111225, 2022.

[115] M. Camilli and B. Russo. Modeling performance of microservices systems with growth theory. *Empirical Software Engineering*, 27(2):1–44, 2022.

[116] L. Cao and P. Sharma. Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 168–174, 2021.

[117] W. Cao, Z. Cao, and X. Zhang. Research on microservice anomaly detection technology based on conditional random field. In *Journal of Physics: Conference Series*, volume 1213, page 042016. IOP Publishing, 2019.

[118] V. Cardellini, E. Casalicchio, S. Iannucci, M. Lucantonio, S. Mittal, D. Panigrahi, and A. Silvi. An intrusion response system utilizing deep q-networks and system partitions. *arXiv preprint arXiv:2202.08182*, 2022.

[119] A. Carrusca, M. C. Gomes, and J. Leitão. Microservices management on cloud/edge environments. In *International Conference on Service-Oriented Computing*, pages 95–108. Springer, 2019.

[120] C. Cassé, P. Berthou, P. Owezarski, and S. Josset. Using distributed tracing to identify inefficient resources composition in cloud applications. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 40–47. IEEE, 2021.

[121] N. Chen, H. Tu, X. Duan, L. Hu, and C. Guo. Semisupervised anomaly detection of multivariate time series based on a variational autoencoder. *Applied Intelligence*, pages 1–25, 2022.

[122] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran. {CLARION}: Sound and clear provenance tracking for microservice deployments. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3989–4006, 2021.

[123] B. Choi, J. Park, C. Lee, and D. Han. phpa: A proactive autoscaling framework for microservice chain. In *5th Asia-Pacific Workshop on Networking (APNet 2021)*, pages 65–71, 2021.

[124] N. Chondamrongkul, J. Sun, and I. Warren. Formal security analysis for software architecture design: An expressive framework to emerging architectural styles. *Science of Computer Programming*, 206:102631, 2021.

[125] S. Choochotkaew, T. Chiba, S. Trent, and M. Amaral. Run wild: Resource management system with generalized modeling for microservices on cloud. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 609–618. IEEE, 2021.

[126] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu. Deeprest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 181–198, 2022.

[127] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *Journal of Systems and Software*, 183:111084, 2022.

[128] V. Cortellessa and L. Traini. Detecting latency degradation patterns in service-based systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 161–172, 2020.

[129] C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson. Automated performance prediction of microservice applications using simulation. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.

[130] C. Cui, G. Wu, W. Chen, J. Zhu, and J. Wei. Feedback-based, automated failure testing of microservice-based applications. *arXiv preprint arXiv:1908.06466*, 2019.

[131] J. Curtis. On language-agnostic abstract-syntax trees: student research abstract. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1619–1625, 2022.

[132] Q. N. Dao and T. Hey. *Coreference Resolution for Software Architecture Documentation.* PhD thesis, Informatics Institute, 2022.

[133] N. Daw, U. Bellur, and P. Kulkarni. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 585–599, 2021.

[134] E. De Angelis, A. Pellegrini, and M. Proietti. Automatic extraction of behavioral features for test program similarity analysis. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 129–136. IEEE, 2021.

[135] A. De Iasio and E. Zimeo. A framework for microservices synchronization. *Software: Practice and Experience*, 51(1):25–45, 2021.

[136] U. Desai, S. Bandyopadhyay, and S. Tamilselvam. Graph neural network to dilute outliers for refactoring monolith application. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 72–80, 2021.

[137] D. Di Pompeo, M. Tucci, A. Celi, and R. Eramo. A microservice reference case study for design-runtime interaction in mde. In *STAF (Co-Located Events)*, pages 23–32, 2019.

[138] M. Dinga. An empirical evaluation of the energy and performance overhead of monitoring tools on docker-based systems. Retrived August 2022 from https://github.com/MadalinaDinga/thesis-2022-monitoring-tools-integration-with-train-ticket-replication-package/, 2022.

[139] P. Dogga, K. Narasimhan, A. Sivaraman, S. Saini, G. Varghese, and R. Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. *Proceedings of Machine Learning and Systems*, 4:601–622, 2022.

[140] dot-net architecture. eshoponcontainers. https://github.com/dotnet-architecture/eShopOnContainers.

[141] T. F. Düllmann, A. van Hoorn, V. Yussupov, P. Jakovits, and M. Adhikari. Ctt: Load test automation for tosca-based cloud applications. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pages 89–96, 2022.

[142] EdwinVW. Pitstop: Garage management system. https://github.com/EdwinVW/pitstop/.

[143] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn. Microservices: A performance tester's dream or nightmare? In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 138–149, 2020.

[144] A. El Malki and U. Zdun. Evaluation of api request bundling and its impact on performance of microservice architectures. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 419–424. IEEE, 2021.

[145] S. Elflein, I. S. Kounev, J. Grohmann, and S. Eismann. Automatic identification of parametric dependencies for performance models.

[146] M. Elsaadawy, M. Younis, J. Wang, X. Hou, and B. Kemme. Dynamic application call graph formation and service identification in cloud data centers. *IEEE Transactions on Network and Service Management*, 2022.

[147] D. Elsner, D. Bertagnolli, A. Pretschner, and R. Klaus. Challenges in regression test selection for end-to-end testing of microservice-based software systems. In *International Conference on Automation of Software Test (AST'22)*, 2022.

[148] A. Erradi, W. Iqbal, A. Mahmood, and A. Bouguettaya. Web application resource requirements estimation based on the workload latent features. *IEEE Transactions on Services Computing*, 2019.

[149] A. M. Espinoza, R. Wood, S. Forrest, and M. Tiwari. Back to the future: N-versioning of microservices. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–427. IEEE, 2022.

[150] A. Farley. Continuous performance testing of faas and microservices based on tosca topology and orchestration specifications. B.S. thesis, 2020.

[151] J. Faro. Deathstarbench metastability failure fork. Retrieved August 2022 from https://github.com/jfaro/DeathStarBench, 2022.

[152] J. Faro. Metastable failure vulnerability testing for deathstarbench's social network micro-service benchmark. Retrieved August 2022 from https://github.com/jfaro/metastable-failures, 2022.

[153] K. Fedyuk. *Sheik: Dynamic location and binding of microservices for cloud/edge settings.* PhD thesis, NOVA University of Lisbon, 2020.

[154] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes. My services got old! can kubernetes handle the aging of microservices? In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 40–47. IEEE, 2021.

[155] J. Forough, M. Bhuyan, and E. Elmroth. Detection of vsi-ddos attacks on the edge: A sequential modeling approach. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.

[156] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840, 2021.

[157] A. Fuerst, A. Ali-Eldin, P. Shenoy, and P. Sharma. Cloud-scale vm-deflation for running interactive applications on transient servers. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 53–64, 2020.

[158] N. Fukuda, C. Wu, S. Horiuchi, and K. Tayama. Fault report generation for ict systems by jointly learning time-series and text data. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2022.

[159] S. Galantino, M. Iorio, F. Risso, and A. Manzalini. Raygo: Reserve as you go. In *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 269–275. IEEE, 2021.

[160] Y. Gan, M. Liang, D. L. Sundar Dev, and C. Delimitrou. Sage: Practical & scalable ml-driven performance debugging in microservices.

[161] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[162] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[163] D. Garg, N. C. Narendra, and S. Tesfatsion. Heuristic and reinforcement learning algorithms for dynamic service placement on mobile edge cloud. *arXiv preprint arXiv:2111.00240*, 2021.

[164] P. Genfer and U. Zdun. Identifying domain-based cyclic dependencies in microservice apis using source code detectors. In *European Conference on Software Architecture*, pages 207–222. Springer, 2021.

[165] P. Genfer and U. Zdun. Avoiding excessive data exposure through microservice apis. In *European Conference on Software Architecture*, pages 3–18. Springer, 2022.

[166] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 150–160, 2020.

[167] A. Gluck. Introducing domain-oriented microservice architecture. (July 2020). Retrieved October 2020 from https://eng.uber.com/microservice-architecture/, 2020.

[168] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian. A holistic machine learning-based autoscaling approach for microservice applications. In *CLOSER*, pages 190–198, 2021.

[169] GoogleCloudPlatform. Online boutique, fka hipstershop. https://github.com/GoogleCloudPlatform/microservices-demo.

[170] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach. Befaas: An application-centric benchmarking framework for faas platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–8. IEEE, 2021.

[171] M. Grambow, E. Wittern, and D. Bermbach. Benchmarking the performance of microservice applications. *ACM SIGAPP Applied Computing Review*, 20(3):20–34, 2020.

[172] D. Granata, M. Rak, and G. Salzillo. Metasend: A security enabled development life cycle meta-model. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[173] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302. IEEE, 2017.

[174] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53. IEEE, 2017.

[175] J. Grohmann, S. Eismann, S. Elflein, J. v. Kistowski, S. Kounev, and M. Mazkatli. Detecting parametric dependencies for performance models using feature selection techniques. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 309–322. IEEE, 2019.

[176] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones. Monitorless: Predicting performance degradation in cloud applications with machine learning. In *Proceedings of the 20th international middleware conference*, pages 149–162, 2019.

[177] J. Grohmann, M. Straesser, A. Chalbani, S. Eismann, Y. Arian, N. Herbst, N. Peretz, and S. Kounev. Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 165–176, 2021.

[178] Z. Guan, J. Lin, and P. Chen. On anomaly detection and root cause analysis of microservice systems. In *International Conference on Service-Oriented Computing*, pages 465–469. Springer, 2018.

[179] C. Guerrero, I. Lera, and C. Juiz. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*, 16(1):113–135, 2018.

[180] C. Guerrero, I. Lera, and C. Juiz. Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. *Future Generation Computer Systems*, 97:131–144, 2019.

[181] C. Guerrero, I. Lera, and C. Juiz. A lightweight decentralized service placement policy for performance optimization in fog computing. *Journal of Ambient Intelligence and Humanized Computing*, 10(6):2435–2452, 2019.

[182] A. Guerriero and L. Giamattei. Microsertest: a flexible tool for automated testing of microservices applications.

[183] D. Guhathakurta, P. Aggarwal, S. Nagar, R. Arora, and B. Zhou. Utilizing persistence for post facto suppression of invalid anomalies using system logs. 2022.

[184] A. Hakamian. Engineering resilience: Predicting the change impact on performance and availability of reconfigurable systems. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 143–146. IEEE, 2020.

[185] W. Hasselbring. Benchmarking as empirical standard in software engineering research. In *Evaluation and Assessment in Software Engineering*, pages 365–372. 2021.

[186] Z. He, P. Chen, X. Li, Y. Wang, G. Yu, C. Chen, X. Li, and Z. Zheng. A spatiotemporal deep learning approach for unsupervised anomaly

detection in cloud systems. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
[187] A. Horn, H. M. Fard, and F. Wolf. Multi-objective hybrid autoscaling of microservices in kubernetes clusters. In *European Conference on Parallel Processing*, pages 233–250. Springer, 2022.
[188] M. R. Hossen and M. A. Islam. Practical efficient microservice autoscaling.
[189] M. R. Hossen and M. A. Islam. Towards efficient microservices management through opportunistic resource reduction.
[190] M. R. Hossen and M. A. Islam. A lightweight workload-aware microservices autoscaling with qos assurance. *arXiv preprint arXiv:2202.00057*, 2022.
[191] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo. Ant-man: towards agile power management in the microservice era. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1098–1111. IEEE Computer Society, 2020.
[192] X. Hou, J. Liu, C. Li, and M. Guo. Unleashing the scalability potential of power-constrained data center in the microservice era. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.
[193] L. Huang and T. Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021.
[194] L. Ifflän der, N. Rawtani, L. Beierlieb, N. Fella, K.-D. Lange, and S. Kounev. Attack-aware security function chain reordering. *arXiv preprint arXiv:2005.08364*, 2020.
[195] T. Inagaki, Y. Ueda, T. Nakaike, and M. Ohara. Profile-based detection of layered bottlenecks. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 197–208, 2019.
[196] T. Inagaki, Y. Ueda, M. Ohara, S. Choochotkaew, M. Amaral, S. Trent, T. Chiba, and Q. Zhang. Detecting layered bottlenecks in microservices. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 385–396. IEEE, 2022.
[197] M. R. Islam, A. Al Maruf, and T. Cerny. Code smell prioritization with business process mining and static code analysis: A case study. *Electronics*, 11(12):1880, 2022.
[198] A. A. T. Isstaif. Self-managed services using mirageos unikernels. In *Proceedings of the 21st International Middleware Conference Doctoral Symposium*, pages 35–37, 2020.
[199] istio. Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/.
[200] J. Ivers, C. Seifried, and I. Ozkaya. Untangling the knot: Enabling architecture evolution with search-based refactoring. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 101–111. IEEE, 2022.
[201] S. Jacob, Y. Qiao, and B. Lee. Detecting cyber security attacks against a microservices application using distributed tracing. In *ICISSP*, pages 588–595, 2021.
[202] S. Jacob, Y. Qiao, Y. Ye, and B. Lee. Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks. *Computers & Security*, 118:102728, 2022.
[203] S. Ji, W. Wu, and Y. Pu. Multi-indicators prediction in microservice using granger causality test and attention lstm. In *2020 IEEE World Congress on Services (SERVICES)*, pages 77–82. IEEE, 2020.
[204] T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang, and Y. Liu. Machine deserves better logging: a log enhancement approach for automatic fault diagnosis. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 106–111. IEEE, 2018.
[205] Z. Jia and E. Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices extended abstract.
[206] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu. Lessons learned from migrating complex stateful applications onto serverless platforms. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 89–96, 2021.
[207] D. Johansson. Model-driven development for microservices: A domain-specific modeling language for kubernetes, 2022.
[208] C. T. Joseph and K. Chandrasekaran. Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments. *Journal of Systems Architecture*, 111:101785, 2020.
[209] H. Kadiyala, A. Misail, and J. Rubin. Kuber: cost-efficient microservice deployment planner. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 252–262. IEEE, 2022.
[210] M. Kang and J. Kim. Comparison of service description and composition for complex 3-tier cloud-based services. *Proceedings of the Asia-Pacific Advanced Network*, 44:37–40, 2017.
[211] M. Kang, J.-S. Shin, and J. Kim. Protected coordination of service mesh for container-based 3-tier service traffic. In *2019 International Conference on Information Networking (ICOIN)*, pages 427–429. IEEE, 2019.
[212] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth. Automated testing and resilience of microservice's network-link using istio service mesh. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 79–88. IEEE, 2022.
[213] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolek. Trace link recovery for software architecture documentation. In *European Conference on Software Architecture*, pages 101–116. Springer, 2021.
[214] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. De Lara. {JITServer}: Disaggregated caching {JIT} compiler for the {JVM} in the cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, 2022.
[215] P. Kokić. *Unaprijedjenje sigurnosti pomoću mikroservisa*. PhD thesis, University of Zagreb. Faculty of Organization and Informatics. Department of …, 2018.
[216] J. Kosińska and K. Zieliński. Autonomic management framework for cloud-native applications. *Journal of Grid Computing*, 18(4):779–796, 2020.
[217] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, et al. Apiary: A dbms-backed transactional function-as-a-service framework. *arXiv preprint arXiv:2208.13068*, 2022.
[218] N. Kratzke. About the complexity to transfer cloud applications at runtime and how container platforms can contribute? In *International Conference on Cloud Computing and Services Science*, pages 19–45. Springer, 2017.
[219] N. Kratzke. Smuggling multi-cloud support into cloud-native applications using elastic container platforms. In *CLOSER*, pages 29–42, 2017.
[220] N. Kratzke. About an immune system understanding for cloud-native applications. *CLOUD COMPUTING 2018*, page 41, 2018.
[221] N. Kratzke. About being the tortoise or the hare?-a position paper on making cloud applications too fast and furious for attackers. *arXiv preprint arXiv:1802.03565*, 2018.
[222] N. Kratzke and P.-C. Quint. Project cloudtransit.
[223] R. Laigner, Y. Zhou, and M. A. V. Salles. A distributed database system for event-based microservices. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 25–30, 2021.
[224] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.
[225] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory

reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[226] A. Lesniak, R. Laigner, and Y. Zhou. Enforcing consistency in microservice architectures through event-based constraints. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 180–183, 2021.

[227] J. Levin and T. A. Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[228] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[229] R. Li, M. Du, H. Chang, S. Mukherjee, and E. Eide. Deepstitch: Deep learning for cross-layer stitching in microservices. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, pages 25–30, 2020.

[230] R. Li, M. Du, Z. Wang, H. Chang, S. Mukherjee, and E. Eide. Longtale: Toward automatic performance anomaly explanation in microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 5–16, 2022.

[231] X. Li, Y. Chen, and Z. Lin. Towards automated inter-service authorization for microservice applications. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 3–5, 2019.

[232] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3971–3988, 2021.

[233] Y. Li, T. Li, P. Shen, L. Hao, W. Liu, S. Wang, Y. Song, and L. Bao. Sim-drs: a similarity-based dynamic resource scheduling algorithm for microservice-based web systems. *PeerJ Computer Science*, 7:e824, 2021.

[234] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, et al. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.

[235] Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui, et al. Actionable and interpretable fault localization for recurring failures in online service systems. *arXiv preprint arXiv:2207.09021*, 2022.

[236] H. Lim, Y. Kim, and K. Sun. Service management in virtual machine and container mixed environment using service mesh. In *2021 International Conference on Information Networking (ICOIN)*, pages 528–530. IEEE, 2021.

[237] F. Liu, Y. Wang, Z. Li, R. Ren, H. Guan, X. Yu, X. Chen, and G. Xie. Microcbr: Case-based reasoning on spatio-temporal fault knowledge graph for microservices troubleshooting. In *International Conference on Case-Based Reasoning*, pages 224–239. Springer, 2022.

[238] J. Loff. Deathstarbench inconsistency. Retrieved August 2022 from https://github.com/jfloff/antipode-deathstarbench, 2022.

[239] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei. Fitness-guided resilience testing of microservice-based applications. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 151–158. IEEE, 2020.

[240] N. D. S. Loureiro. *Compositional analysis of vulnerabilities in microservice-based applications*. PhD thesis, 2021.

[241] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[242] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[243] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, and C. Lin. Microservice deployment in edge computing based on deep q learning. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[244] J. Mace and R. Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–18, 2018.

[245] N. Marie-Magdelaine and T. Ahmed. Proactive autoscaling for cloud-native applications using machine learning. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–7. IEEE, 2020.

[246] D. R. Mathews, M. Verma, J. Lakshmi, and P. Aggarwal. Towards more effective and explainable fault management using cross-layer service topology. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 94–96. IEEE, 2022.

[247] A. P. Matthews. *Microservice Pattern Identification from Recovered Architectures of Orchestrated Systems*. University of California, Irvine, 2021.

[248] M. Mazkatli, D. Monschein, J. Grohmann, and A. Koziolek. Incremental calibration of architectural performance models with parametric dependencies. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 23–34. IEEE, 2020.

[249] D. Mealha, N. Preguiça, M. C. Gomes, and J. Leitão. Data replication on the cloud/edge. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–7, 2019.

[250] W. D. Mendonça, W. K. Assunção, L. V. Estanislau, S. R. Vergilio, and A. Garcia. Towards a microservices-based product line with multi-objective evolutionary algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[251] microservices demo. Retrieved August 2022 from https://github.com/microservices-demo/microservices-demo, 2016.

[252] A. Mirhosseini, S. Elnikety, and T. F. Wenisch. Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 442–457, 2021.

[253] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198. IEEE, 2019.

[254] H. Mohamed and O. El-Gayar. End-to-end latency prediction of microservices workflow on kubernetes: A comparative evaluation of machine learning models and resource metrics. In *Proceedings of the 54th Hawaii International Conference on System Sciences*, page 1717, 2021.

[255] J. Mukherjee, A. Baluta, M. Litoiu, and D. Krishnamurthy. Rad: Detecting performance anomalies in cloud-based web services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 493–501. IEEE, 2020.

[256] G. Muntoni, J. Soldani, and A. Brogi. Mining the architecture of microservice-based applications from their kubernetes deployment. In *European Conference on Service-Oriented and Cloud Computing*, pages 103–115. Springer, 2020.

[257] A. Nadeem and M. Z. Malik. A case for microservices orchestration using workflow engines. *arXiv preprint arXiv:2204.07210*, 2022.

[258] S. Nagar, S. Samanta, P. Mohapatra, and D. Kar. Building golden signal based signatures for log anomaly detection. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 203–208. IEEE, 2022.

[259] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin. {BASTION}: A security enforcement network stack for container networks. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 81–95, 2020.

[260] A. R. Nasab, M. Shahin, S. A. H. Raviz, P. Liang, A. Mashmool, and V. Lenarduzzi. An empirical study of security practices for microservices systems. *arXiv preprint arXiv:2112.14927*, 2021.

[261] T. Nebaeus. Optimal scaling configurations for microservice-oriented architectures using genetic algorithms, 2019.

[262] C. Nguyen, A. Mehta, C. Klein, and E. Elmroth. Why cloud applications are not ready for the edge (yet). In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 250–263, 2019.

[263] H. X. Nguyen, S. Zhu, and M. Liu. Graph-phpa: Graph-based proactive horizontal pod autoscaling for microservices using lstm-gnn. *arXiv preprint arXiv:2209.02551*, 2022.

[264] S. Nijboer. Traffic analysis of a service mesh. Master's thesis, 2021.

[265] E. T. Nordli, P. H. Nguyen, F. Chauvel, and H. Song. Event-based customization of multi-tenant saas using microservices. In *International Conference on Coordination Languages and Models*, pages 171–180. Springer, 2020.

[266] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, and W. Hasselbring. Detector-based component model abstraction for microservice-based systems. *Computing*, 103(11):2521–2551, 2021.

[267] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *International Conference on Service-Oriented Computing*, pages 580–596. Springer, 2020.

[268] E. Ntentos, U. Zdun, J. Soldani, and A. Brogi. Assessing architecture conformance to coupling-related infrastructure-as-code best practices: Metrics and case studies. In *European Conference on Software Architecture*, pages 101–116. Springer, 2022.

[269] M. S. Nyfløtt. Optimizing inter-service communication between microservices. Master's thesis, NTNU, 2017.

[270] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, 2022.

[271] A. Osman, J. Born, and T. Strufe. Mitigating internal, stealthy dos attacks in microservice networks. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 500–504. Springer, 2021.

[272] overleaf. Overleaf. https://github.com/overleaf/overleaf.

[273] M. Pancholi, A. D. Kellas, V. P. Kemerlis, and S. Sethumadhavan. Timeloops: System call policy learning for containerized microservices. *arXiv preprint arXiv:2204.06131*, 2022.

[274] J. Park, B. Choi, C. Lee, and D. Han. Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[275] M. A. Patterns. Lakeside mutual. https://github.com/Microservice-API-Patterns/LakesideMutual.

[276] H. Phalachandra, P. Bhatt, I. Agarwal, and R. Bhowmick. Improving task scheduling in microservice environments by considering intra-job dependencies. In *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 568–573. IEEE, 2022.

[277] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1216, 2020.

[278] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2021.

[279] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 805–825, 2020.

[280] P.-C. Quint and N. Kratzke. Towards a description of elastic cloud-native applications for transferable multi-cloud-deployments. *Small*, 147:34, 2017.

[281] P.-C. Quint and N. Kratzke. Towards a lightweight multi-cloud dsl for elastic and transferable cloud-native applications. *arXiv preprint arXiv:1802.03562*, 2018.

[282] F. Rademacher, S. Sachweh, and A. Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020.

[283] J. Rahman and P. Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.

[284] M. I. Rahman, S. Panichella, and D. Taibi. A curated dataset of microservices-based systems, extended version. Vol-2520, 2019.

[285] K. Ramasubramanian, A. Raina, J. Mace, and P. Alvaro. Act now: Aggregate comparison of traces for incident localization. *arXiv preprint arXiv:2205.06933*, 2022.

[286] V. Rao, V. Singh, K. Goutham, B. U. Kempaiah, R. J. Mampilli, S. Kalambur, and D. Sitaram. Scheduling microservice containers on large core machines through placement and coalescing. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 80–100. Springer, 2021.

[287] K. Ray and A. Banerjee. Horizontal auto-scaling for multi-access edge computing using safe reinforcement learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(6):1–33, 2021.

[288] redis. Redis patterns example. https://redis.io/docs/reference/patterns/twitter-clone/.

[289] J. S. Rellermeyer, M. Amer, R. Smutzer, and K. Rajamani. Container density improvements with dynamic memory extension using nand flash. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.

[290] F. G. O. Risso and S. Galantino. Enabling job aware scheduling on kubernetes cluster.

[291] rnp68. Teastore cve-2021-44228. Retrieved September 2022 from https://github.com/rnp68/TeaStore, 2021.

[292] O. R. C. Rodríguez, C. Pahl, N. El Ioini, H. R. Barzegar, et al. Improvement of edge computing workload placement using multi objective particle swarm optimization. In *2021 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 1–8. IEEE, 2021.

[293] Y. Rouf, J. Mukherjee, M. Litoiu, J. Wigglesworth, and R. Mateescu. A framework for developing devops operation automation in clouds using components-off-the-shelf. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 265–276, 2021.

[294] W. Runan, G. Casale, and A. Filieri. Service distribution estimation for microservices using markovian arrival processes.

[295] E. Rwemigabo. *Autonomic application topology redistribution*. PhD thesis, 2018.

[296] V. Sachidananda and A. Sivaraman. Learned autoscaling for cloud microservices with multi-armed bandits. *arXiv preprint arXiv:2112.14845*, 2021.

[297] M. Sakai and K. Takahashi. Constructing a service process model based on distributed tracing for conformance checking of microservices. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE, 2022.

[298] M. Sakai, K. Takahashi, and S. Kondoh. Method of constructing petri net service model using distributed trace data of microservices. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 214–217. IEEE, 2021.

[299] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 17–27, 2022.

[300] A. R. SAMPAIO JUNIOR. Runtime adaptation of microservices. 2018.

[301] M. Santana, A. Sampaio Jr, M. Andrade, and N. S. Rosa. Transparent tracing of microservice-based applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1252–1259, 2019.
[302] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot. Lukewarm serverless functions: characterization and optimization. In *ISCA*, pages 757–770, 2022.
[303] M. Schiedermeier. A concern-oriented software engineering methodology for micro-service architectures. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, 2020.
[304] M. Schiewe. Bridging the gap between source code and high-level concepts in static code analysis: student research abstract. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1615–1618, 2022.
[305] M. Schiewe, J. Curtis, V. Bushong, and T. Cerny. Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761, 2022.
[306] N. Schmitt, L. Iffländer, A. Bauer, and S. Kounev. Online power consumption estimation for functions in cloud applications. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 63–72. IEEE, 2019.
[307] Y. R. Schneider and A. Koziolek. Towards reverse engineering for component-based systems with domain knowledge of the technologies used.
[308] SEALABQualityGroup. E-shopper. https://github.com/SEALABQualityGroup/E-Shopper.
[309] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan. Pmnet: in-network data persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2021.
[310] K. Sellami, M. A. Saied, and A. Ouni. A hierarchical dbscan method for extracting microservices from monolithic applications. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, pages 201–210, 2022.
[311] P. V. Seshadri, H. Balagopal, A. Nayak, A. P. Kumar, and P. Loyola. Konveyor move2kube: A framework for automated application replatforming. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 115–124. IEEE, 2022.
[312] R. Sharma and A. Mathur. Traefik as kubernetes ingress. In *Traefik API Gateway for Microservices*, pages 191–245. Springer, 2021.
[313] O. Shchur, A. C. Turkmen, T. Januschowski, J. Gasthaus, and S. Günnemann. Detecting anomalous event sequences with temporal point processes. *Advances in Neural Information Processing Systems*, 34:13419–13431, 2021.
[314] J. Shi, J. Wang, K. Fu, Q. Chen, D. Zeng, and M. Guo. Qos-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 324–334. IEEE, 2022.
[315] A. Singjai and U. Zdun. Conformance assessment of architectural design decisions on the mapping of domain model elements to apis and api endpoints. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 76–83. IEEE, 2022.
[316] A. Singjai, U. Zdun, O. Zimmermann, M. Stocker, and C. Pautasso. Patterns on designing api endpoint operations. 2021.
[317] D. Sokolowski. Deployment coordination for cross-functional devops teams. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1630–1634, 2021.
[318] D. Sokolowski. Infrastructure as code for dynamic deployments. 2022.
[319] J. Soldani and A. Brogi. Automated generation of configurable cloud-native chaos testbeds. In *European Dependable Computing Conference*, pages 101–108. Springer, 2021.
[320] J. Soldani, G. Montesano, and A. Brogi. What went wrong? explaining cascading failures in microservice-based applications. In *Symposium and Summer School on Service-Oriented Computing*, pages 133–153. Springer, 2021.
[321] J. Soldani, G. Muntoni, D. Neri, and A. Brogi. The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621, 2021.
[322] G. Somashekar, A. Dutt, R. Vaddavalli, S. B. Varanasi, and A. Gandhi. B-meg: Bottlenecked-microservices extraction using graph neural networks. 2022.
[323] G. Somashekar and A. Gandhi. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 7–14, 2021.
[324] H. Song, P. H. Nguyen, and F. Chauvel. Using microservices to customize multi-tenant saas: From intrusive to non-intrusive. In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[325] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):1–25, 2021.
[326] S. Speth. Semi-automated cross-component issue management and impact analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1090–1094. IEEE, 2021.
[327] A. Sriraman and T. F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.
[328] A. Sriraman and T. F. Wenisch. μtune: Auto-tuned threading for {OLDI} microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 177–194, 2018.
[329] C.-a. Sun, J. Wang, Z. Liu, and Y. Han. A variability-enabling and model-driven approach to adaptive microservice-based systems. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 968–973. IEEE, 2021.
[330] H. Sun, D. Bonetta, C. Humer, and W. Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.
[331] Y. Sun, L. Zhao, Z. Wang, D. Cui, Y. Yang, and Z. Gao. Fault root rank algorithm based on random walk mechanism in fault knowledge graph. In *2021 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6. IEEE, 2021.
[332] Y. Tang, N. Chen, Y. Zhang, X. Yao, and S. Yu. Fine-grained diagnosis method for microservice faults based on hierarchical correlation analysis. In *CCF Conference on Computer Supported Cooperative Work and Social Computing*, pages 14–28. Springer, 2022.
[333] P. Tennage, S. Perera, M. Jayasinghe, and S. Jayasena. An analysis of holistic tail latency behaviors of java microservices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 697–705. IEEE, 2019.
[334] V. Thrivikraman, V. R. Dixit, V. K. Gowda, S. K. Vasudevan, and S. Kalambur. Misertrace: Kernel-level request tracing for microservice visibility. *CoRR*, 2022.
[335] O. Tomarchio, D. Calcaterra, G. Di Modica, and P. Mazzaglia. Torch: a tosca-based orchestrator of multi-cloud containerised applications. *Journal of Grid Computing*, 19(1):1–25, 2021.
[336] K. A. Torkura, C. Meinel, and N. Kratzke. Don't wait to be breached! creating asymmetric uncertainty of cloud applications via moving target defenses. *arXiv preprint arXiv:1901.04319*, 2019.
[337] M. Toslali, E. Ates, A. Ellis, Z. Zhang, D. Huye, L. Liu, S. Puterman, A. K. Coskun, and R. R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with

vaif. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–75, 2021.

[338] M. Toslali, S. Parthasarathy, F. Oliveira, H. Huang, and A. K. Coskun. Iter8: Online experimentation in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 289–304, 2021.

[339] L. Traini and V. Cortellessa. Delag: Detecting latency degradation patterns in service-based systems. *arXiv preprint arXiv:2110.11155*, 2021.

[340] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018.

[341] T. Ueda, T. Nakaike, and M. Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016.

[342] T. N. Y. U. T. Ueda and M. Ohara. Performance characteristics of linux for java workloads oversubscribing memory.

[343] A. van Hoorn and T. F. Düllmann. Continuous testing tool i. 2020.

[344] W. Viktorsson, C. Klein, and J. Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–4. IEEE, 2020.

[345] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[346] H. Vural and M. Koyuncu. Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*, 9:32721–32733, 2021.

[347] A. Walker, D. Das, and T. Cerny. Automated microservice code-smell detection. In *Information Science and Applications*, pages 211–221. Springer, 2021.

[348] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 142–150. IEEE, 2020.

[349] Q. Wang, L. Shwartz, G. Y. Grabarnik, V. Arya, and K. Shanmugam. Detecting causal structure on cloud application microservices using granger causality models. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 558–565. IEEE, 2021.

[350] R. Wang, G. Casale, and A. Filieri. Service demand distribution estimation for microservices using markovian arrival processes. In *International Conference on Quantitative Evaluation of Systems*, pages 310–328. Springer, 2021.

[351] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, and M. Guo. Exploring efficient microservice level parallelism. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[352] S. Waterworth. Stan's robot shop – a sample microservice application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/, 2018.

[353] M. Wawrzoniak, I. Müller, R. Bruno, A. Klimovic, and G. Alonso. Short-lived datacenter. *arXiv preprint arXiv:2202.06646*, 2022.

[354] N. Wenzler. Automated performance regression detection in microservice architectures. B.S. thesis, 2017.

[355] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao. Performance diagnosis in cloud microservices using deep learning. In *AIOPS 2020-International Workshop on Artificial Intelligence for IT Operations*, 2020.

[356] L. Wu, J. Tordsson, A. Acker, and O. Kao. Microras: Automatic recovery in the absence of historical failure data for microservice systems. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 227–236. IEEE, 2020.

[357] L. Wu, J. Tordsson, E. Elmroth, and O. Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[358] X. Wu, S. Cheng, S. Yuan, and Z. Wang. A parallelism-based earliest finish time (pbeft) algorithm for workflow scheduling in clouds. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 22–26. IEEE, 2022.

[359] Y. Xia. *Reducing the Length of Field-replay Based Load Testing*. PhD thesis, Concordia University, 2021.

[360] R. Xin, P. Chen, and Z. Zhao. Causalrca: Causal inference based precise fine-grained root cause localization for microservice applications. *arXiv preprint arXiv:2209.02500*, 2022.

[361] J. Xu, P. Chen, L. Yang, F. Meng, and P. Wang. Logdc: Problem diagnosis for declartively-deployed cloud applications with log. In *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, pages 282–287. IEEE, 2017.

[362] M. Xu and R. Buyya. Brownoutcon: A software system based on brownout and containers for energy-efficient cloud computing. *Journal of Systems and Software*, 155:91–103, 2019.

[363] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic. Lessons learned from hyper-parameter tuning for microservice candidate identification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1141–1145. IEEE, 2021.

[364] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic. Partitioning cloud-based microservices (via deep learning). *arXiv preprint arXiv:2109.14569*, 2021.

[365] K. Yin and Q. Du. On representing resilience requirements of microservice architecture systems. *International Journal of Software Engineering and Knowledge Engineering*, 31(06):863–888, 2021.

[366] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, pages 3087–3098, 2021.

[367] G. Yu, P. Chen, and Z. Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75. IEEE, 2019.

[368] G. Yu, P. Chen, and Z. Zheng. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Transactions on Cloud Computing*, 2020.

[369] U. Zdun, P.-J. Queval, G. Simhandl, R. Scandariato, S. Chakravarty, M. Jelic, and A. Jovanovic. Microservice security metrics for secure communication, identity management, and observability. *ACM Transactions on Software Engineering and Methodology*, 2022.

[370] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. 2022.

[371] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1187–1204, 2020.

[372] K. Zhang, A. Filieri, and R. Chatley. Kubedim: Self-adaptive service degradation of microservices-based systems. 2021.

[373] L. Zhang, V. Anand, Z. Xie, Y. Vigfusson, and J. Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. *arXiv preprint arXiv:2202.05769*, 2022.

[374] Y. Zhang, Y. Gan, and C. Delimitrou. μqsim: Enabling accurate and scalable simulation for interactive microservices. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 212–222. IEEE, 2019.

[375] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou. Sinan: Ml-based & qos-aware resource management for cloud microservices.

[376] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[377] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[378] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.

[379] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*, 2019.

[380] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 683–694, 2019.

[381] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding. Delta debugging microservice systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 802–807. IEEE, 2018.

[382] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Poster: Benchmarking microservice systems for software engineering research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 323–324. IEEE, 2018.

[383] L. Zhu. Decomposition tool i. 2019.

[384] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, et al. Dissecting service mesh overheads. *arXiv preprint arXiv:2207.00592*, 2022.

[385] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker. Interface responsibility patterns: processing resources and operation responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–24, 2020.

[386] O. Zimmermann, C. Pautasso, D. Lübke, U. Zdun, and M. Stocker. Data-oriented interface responsibility patterns: Types of information holder resources. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–25, 2020.

[387] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun. Introduction to microservice api patterns (map). 2019.

[388] W. Zorgdrager, K. Psarakis, M. Fragkoulis, E. Visser, and A. Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. *arXiv preprint arXiv:2112.00710*, 2021.