Generating Executable Microservice Applications for Performance Benchmarking

Author version. Final version published by ACM.

Yannik Lubas
University of Würzburg
Würzburg, Germany
yannik.lubas@uni-wuerzburg.de

Martin Straesser
University of Würzburg
Würzburg, Germany
martin.straesser@uni-wuerzburg.de

André Bauer
Illinois Institute of Technology
Chicago, IL, USA
andre.bauer@iit.edu

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

Abstract

Microservice applications are the building blocks of modern cloud applications. As such, their performance aspects have been receiving increasing attention in the software engineering community. However, many microservice performance studies use only a small set of popular microservice test applications for experiments, questioning the applicability of their approaches in practice. Researchers currently lack the opportunity to collect large and diverse datasets containing performance metrics of microservices. This is because popular test applications only represent specific technology stacks and often come with custom benchmark tooling (e.g., load generation and monitoring). In this paper, we present Creo, a framework for generating microservice applications that (1) are fully executable, (2) have configurable properties and resource usage profiles, and (3) have built-in support for standardized monitoring, load generation, and deployment. Our approach enables researchers to run experiments with diverse microservice applications with minimal effort. We demonstrate the value of our approach in the context of two use cases. First, we show that using generated applications when training machine learning models for predicting performance degradation can improve the prediction accuracy. Second, we evaluate a recent approach for performance anomaly classification on a set of generated applications highlighting strengths and weaknesses not discussed in the original work.

CCS Concepts

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Cloud computing**.

ICPE '25, May 5–9, 2025, Toronto, ON, Canada

ACM ISBN 979-8-4007-1073-5/2025/05
https://doi.org/10.1145/3676151.3719361

Keywords

microservices, benchmarking, software performance, microservice generator

ACM Reference Format:
Yannik Lubas, Martin Straesser, André Bauer, and Samuel Kounev. 2025. Generating Executable Microservice Applications for Performance Benchmarking: Author version. Final version published by ACM.. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3676151.3719361

1 Introduction

Microservice applications have become common practice for the development of cloud applications in recent years. Apart from the many advantages that microservices bring to development and runtime, tasks in the area of application performance management have proven to pose significant challenges. Prominent examples from the literature include root cause localization [23], autoscaling [8], placement [2], and many more. The sheer mass of different services coupled with regular updates through modern DevOps practices makes it infeasible to use approaches specialized for one service or application. Instead, application-agnostic approaches are favored, which can deal with different microservices [27]. These application-agnostic approaches primarily rely on machine learning models that can learn the performance characteristics of different services and applications through supervised learning.

Our preliminary analysis of the literature in the field of performance engineering for microservices found a prevalent discussed threat to validity. Many related works train and evaluate their approaches on just one or two test applications. This limitation restricts the generalizability of research results and hampers their practical adoption. In addition to the limited scope of scientific work, our experience has shown that collecting training data and experimenting with various microservice applications is time-consuming. Many test applications require different tooling for performance experiments (e.g., monitoring frameworks, load generators, distributed tracing). Another reason is the limited diversity of open-source microservice applications [22] (e.g., in terms of the number of services, communication forms, and programming languages). Given that production microservice applications are usually not

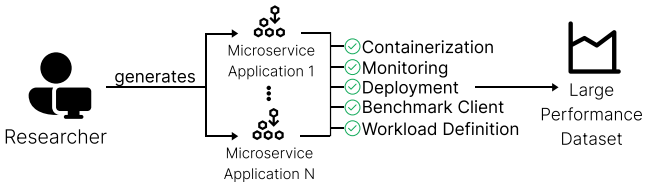

Figure 1: Use case scenario: Performance benchmarking

available for experimentation, researchers currently lack the ability to generate large, diverse performance datasets to evaluate novel approaches for application performance management.

To address this issue, we propose Creo, a framework for generating executable microservice applications for performance benchmarking. At its core, Creo assembles so-called handler functions written in different programming languages and with optional external dependencies (e.g., databases) into microservices. A microservice application is created by generating multiple microservices that interact via network communication. Researchers can generate a virtually infinite number of microservice applications by designing an arbitrarily large set of handler functions. In many use cases, researchers need to generate applications with specific properties instead of random ones. Handler functions have generic labels indicating their properties to support this requirement. In this work, we use performance labels (e.g., *CPU*, or *NETWORK_TRANSMIT* usage) specifying the resource usage profile. Researchers can specify the desired properties (e.g., high CPU usage) of the generated microservice applications in advance. In addition to generating the source code and container images, Creo also creates monitoring harnesses, workload definitions for load generators, benchmark clients, and deployment files for every generated application. Researchers can generate numerous microservice applications enabling the creation of large and diverse datasets with minimal effort, as illustrated in Figure 1. This paper explains the methodology of Creo and demonstrates its usage in two case studies. In the first case study, we show how a state-of-the-art machine learning-based approach for performance degradation prediction [13] can be improved by enriching its training data with measurements from generated applications. In our second case study, we evaluate a novel interference detection approach [4] on generated applications to evaluate its generalizability.

In summary, the contributions of this paper are three-fold:

- We present Creo, a framework for generating executable microservice applications with preconfigured properties and tooling for performance benchmarking.
- We show how training data can be enriched with data from generated applications, improving an approach for performance degradation prediction [13].
- We illustrate the versatility of our generated applications by using them in a case study for extending the evaluation of an interference detection approach [4].

The remainder of this paper is structured as follows: Section 2 covers extended background and motivation for this work. Section 3 presents the approach of Creo alongside selected examples. Sections 4 and 5 present our case studies, demonstrating our approach's applicability. In Section 6, we discuss the results of this work and address limitations. Section 7 summarizes related works, while Section 8 draws conclusions.

2 Background and Motivation

This section presents a motivational review of selected works on applications used for benchmarking and evaluation in microservice performance and resource management. Further, we introduce the two case studies covered in Sections 4 and 5 that illustrate the value of generated applications to obtain performance data.

2.1 Microservice Applications for Benchmarking and Evaluation Purposes

As part of our preliminary study and to motivate our work, we analyzed papers on microservice performance and resource management. Our keyword-based search spanned papers from 2018 to 2024 from 14 well-established conferences and journals in software engineering and cloud computing. Our replication package [18] contains a complete list of the venues we searched, the papers we found, and additional metadata for this review. We include papers that answer how many and which microservice applications are used for experiments in the paper. In addition, we review whether the authors mention limitations or future work concerning their evaluation or used benchmarking applications. The study aims to analyze typical microservice test applications and identify potential use cases for generated applications.

Within our search scope, we identified a total of 36 relevant papers. There was a clear trend in the number of microservice applications used. 22 out of 36 papers (61%) use only one microservice application for experiments. Two (8/36, 22%) or three (5/36, 14%) benchmarking applications were used less frequently. The paper by Somashekar et al. [24] used the most applications in our analysis. The authors evaluate their work on three applications from the DeathStarBench suite [6, 10] as well as TrainTicket [9, 33]. TrainTicket was the most popular test application in the papers we analyzed (15 times used). Other relevant test applications were SockShop [32] (10), OnlineBoutique [11] (4), and TeaStore [29, 30] (3). We describe selected test applications and their characteristics in more detail in Section 7.

Furthermore, we looked at the limitations of the reviewed papers and future work statements. 20 of the 36 papers (56%) explicitly question the generalizability of the results. The authors acknowledge the shortcoming of only testing their approach on one or a few test applications. Other authors state that their test applications differ from production systems regarding technology stack and number of services. Traini and Cortelessa [28] explicitly emphasize the lack of public datasets containing data from various microservice applications. 11 out of the 36 (31%) papers mention additional experiments with further test applications as a subject for future work.

The central insight from our literature analysis is that most papers only use one microservice application in their experiments. Many authors are aware of the limited generalizability of their results and plan to do experiments with other applications in future work. Experiments with different test applications are usually associated with a high effort for researchers, as many of the mentioned test applications have different toolchains for benchmarking (e.g.,

monitoring, load generation). Together with the limited resources available for preparing a scientific paper, this leads to experiments only being conducted on a few established test applications. This work aims to present an approach that allows the collection of large performance datasets of various diverse microservices with as little effort as possible. For this purpose, we present a generator for executable microservices with standardized load generation, monitoring, and configurable performance characteristics.

2.2 Exemplary Use Cases for Generated Applications

We now describe the two case studies selected from papers identified in the previous section that illustrate the value and applicability of our approach.

Use Case 1: Performance Degradation Prediction. Monitorless [13] uses platform-level metrics to identify whether a cloud application is saturated or not. In the case of saturation, a performance degradation indicated by key performance indicators (KPIs) occurs (e.g., increased response times). The core of the approach is a machine learning (ML) model trained on a set of representative applications (Solr, Cassandra, and Memcached). The training applications should be as diverse as possible to cover various resource usage profiles. A three-tier web application and TeaStore are used for evaluation. In this paper, we analyze whether extending the training dataset with measurements from generated applications can improve the accuracy of the ML model. This use case is an example of an application-agnostic approach that uses different applications for training and inference. Generated applications can enrich the training dataset here. Our approach also allows one to generate applications with specified resource profiles that mimic production applications. We present our case study in Section 4.

Use Case 2: Characterizing Performance Anomalies. Baluta et al. [4] consider performance anomalies of cloud applications, distinguishing between two potential root causes: workload changes and interference. Interference occurs when an application competes for hardware resources with other applications in a virtualized environment. The authors train an ML model that can detect performance anomalies and assign detected anomalies to one of the root causes. AcmeAir [1] and OnlineBoutique are employed as test applications. Our work extends the original evaluation using generated applications with different resource usage profiles (e.g., network-, CPU- and disk-intensive applications). We explore to what extent the paper's results can be generalized to our generated applications. We present our results in Section 5.

3 Approach

This section describes the approach of *Creo* in detail. First, we introduce our conceptual structure of microservice applications and state the framework's goals. Next, we describe the application generation steps and highlight the framework's usage through selected examples. Creo offers numerous optional configuration options that influence the generation process. In this paper, we focus only on the most relevant options; our GitHub repository[1] includes full parameter descriptions.

[1] https://github.com/DescartesResearch/Creo/

3.1 Overview

Figure 2 depicts our conceptual composition of a microservice application. A unique name identifies an *application* that comprises one or more *microservices*. A microservice is implemented in a particular programming language and usually uses a web server framework to process incoming requests that arrive on a specific port. Microservices consist of one or more *endpoints*. The accessible endpoints of a microservice define its application programming interface (API). This public API enables executing a typically narrow set of business functionality through network requests.

An endpoint is identified by its path identifier and network protocol. Frontend-facing endpoints typically use HTTP, while application-internal endpoints may communicate with different protocols like RPC. In the case of HTTP, the path identifier also includes the accepted HTTP method. An endpoint calls zero or more endpoints of *other* microservices, which we denote as *inter-service calls*. Inter-service calls enable the endpoint of one microservice to query information from or trigger actions of other microservices in the application. An inter-service call is uniquely identified by the combination of its origin (i.e., source endpoint and microservice) and its destination (i.e., target endpoint and microservice). Per definition, an inter-service call's source and target service are distinct. Communication inside a service generally uses faster in-process communication (e.g., by calling functions in other modules). The communication type may be synchronous or asynchronous.

Along with inter-service calls, every endpoint executes some business logic. We refer to this logic that processes the request as *handler function*. This business logic can be executed before, after, or (in case of asynchronous inter-service calls) concurrently to inter-service calls.

The *handler function's* source code is implemented in a specific programming language and has defined inputs and outputs that determine the endpoint's request and response payloads. A handler function's programming language must align with the programming language of the associated microservice. A set of labels describes particular properties of the handler function. We explain the role of labels in our approach in more detail in the next section.

In some cases, a handler function may also have external *dependencies*, such as databases or message queues. These dependencies are independently deployable components. The dependent handler function cannot fulfill its particular task without access to these dependencies. For instance, a handler function persisting a user entity requires a connection to a database. In the next section, we provide more detail about handler functions since they are the core building blocks of our approach.

3.2 Handler Functions

Real-world microservice applications commonly define their microservice boundaries by business use case. Consequently, microservices usually comprise handler functions that implement related functionality. However, unlike in real-world applications, the business logic implemented by a particular handler function is secondary to our approach, as we focus on generating microservice applications for performance benchmarking studies. These studies do not benefit from specific business use cases (e.g., booking vs. web store application) but rather from various application characteristics

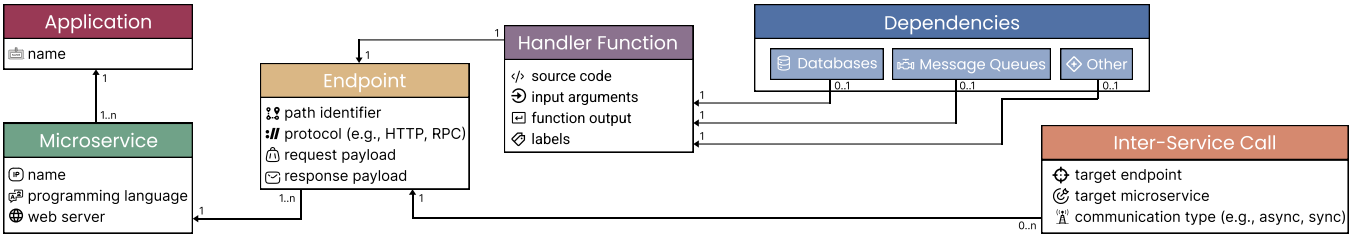

Figure 2: Conceptual composition of a microservice application

(e.g., CPU- vs. memory-intensive or faulty vs. non-faulty). To align with this observation, our framework generates applications with configurable characteristics instead of particular business use cases. Applications are assembled from a set of implemented handler functions used as building blocks. The set of available handler functions should ideally represent a wide range of relevant characteristics instead of focusing on covering any particular functionality.

We use labels for every function to describe its characteristics. Labels are structured as key-value pairs, where the key represents the label's name and the value is numeric. In our generation process, the labels quantify the priority of a handler function. In general, we assume an appropriate procedure for labeling that ensures the labels are comparable. Relevant labels for performance-related studies may include failure rate, energy consumption, or degree of parallelism. In our work, we use resource labels to quantify a function's resource usage characteristics, i.e., `NETWORK_TRANSMIT`, `NETWORK_RECEIVE`, `CPU`, `MEMORY`, `DISK_READ`, and `DISK_WRITE`. We conduct well-defined benchmark runs to determine the resource labels for handler functions. We stress the function with a constant request arrival rate and measure the resource usage. The resulting labels are the average values of the corresponding resource usage across repeated labeling runs. We provide more details about our labeling process in our GitHub repository.[1]

Listing 1 depicts exemplary resource labels for a handler function that generates a given number of random integer values between a provided minimum and maximum. In the listing, `CPU` relates to the CPU usage seconds, `MEMORY` to the used MB of memory, and the remaining labels to the MB/s usage of the respective resource under a measured constant load.

Listing 1: Exemplary resource labels of a handler function

```
---
CPU: 1.6707656131194961
DISK_READ: 0.0
DISK_WRITE: 0.0
MEMORY: 2.742680539841006
NETWORK_RECEIVE: 0.02229803287980295
NETWORK_TRANSMIT: 0.1742029981708346
---
```

As mentioned, a handler function takes a network request's (parsed) payload as its input. We need a specification of expected inputs for inter-service calls and load generators for every handler function. As we support multiple programming languages, we require a language-agnostic format. The OpenAPI Specification[2] defines *Schema Objects* for specifying the structure of data in a language-agnostic way. Listing 2 illustrates an example of a function input specification in YAML format under the `parameters` key inside the `signature` map. We extend the OpenAPI Specification of Schema Objects by an additional property `arg`. This property indicates the position of the particular parameter in the function's signature. In this example, the handler function expects three integer parameters with a value between 1 and 1000. The definition also includes additional information needed for generating valid source code, such as the function name (`function` key), whether the function returns a result (`returns` flag), or whether the function is asynchronous (`is_async` flag).

Listing 2: Exemplary input definition for a handler function

```
---
import_path: rng
description: "Generate random numbers."
is_async: false
returns: true
signature:
  function: generate_random_numbers
  parameters:
    - arg: 0
      type: integer
      format: int32
      minimum: 1
      exclusiveMinimum: false
      maximum: 1000
      exclusiveMaximum: false
    - arg: 1
      type: integer
      format: int64
      minimum: 1
      exclusiveMinimum: false
      maximum: 1000
      exclusiveMaximum: false
    - arg: 2
      type: integer
      format: int64
      minimum: 1
      exclusiveMinimum: false
      maximum: 1000
      exclusiveMaximum: false
---
```

Handler functions are the core of our approach. The more functions are available, the more diverse microservice applications users can generate. The number of possible handler functions is virtually infinite. We try to cover as many resource usage profiles as possible. For instance, we implemented common CRUD functions for persisting entities of different complexity in a database. Furthermore, we included various CPU-intensive functions often found in microbenchmarks, such as inverting a random matrix of a given

[2] https://spec.openapis.org/oas/v3.0.3#schema-object

size or generating random numbers. Finally, we also implemented handler functions that exhibit high disk utilization (i.e., a function archiving existing log files in a compressed or uncompressed `tar` archive or writing a given log message to disk). Within the scope of this work, we implemented 31 handler functions in two programming languages, namely Rust and Python. The complete list with a description for each handler function is accessible in our GitHub repository.[1]

3.3 Generating the Application Graph

This section describes the procedure for dynamically composing handler functions into microservices.

Our framework uses a bottom-up approach for generating applications, i.e., by composing endpoints to services. Creo internally represents a microservice application as a colored, directed, acyclic graph. Each vertex in the graph represents an endpoint, while each directed edge represents an inter-service call from the source vertex (i.e., endpoint) to the target vertex. The number of colors corresponds to the number of microservices. Same-colored vertices belong to the same microservice. Creo supports two conceptually different modes: autopilot (default) or manual. The manual mode allows complete control over the application endpoints, inter-service calls, services, and handler functions. This control comes at the cost of increased manual modeling effort. The autopilot mode requires a reduced configuration (see Listing 3). In this mode, Creo constructs this graph by randomly drawing the given number of edges from the set of all possible edges between the configured number of vertices. A coloring algorithm then provides a proper *k-coloring* for the graph, so there are no edges between endpoints of the same microservice. By default, our framework uses the *equitable coloring* algorithm [19]. The example configuration shown in Listing 3 specifies a graph with 35 vertices (`endpoints`) and 25 edges (`service_calls`).

Listing 3: Exemplary configuration

```
---
name: test-app
mode: auto_pilot
topology:
  services: 10
  endpoints: 35
  inter_service_calls: 25
# seed: myawesomeseed
workload:
  programming_languages: ["rust:30", "python:70"]
  service_types:
    # service type s_1
    - fraction: 50
      properties:
        - label: CPU
          bucket: HIGH
          fraction: 100
    # service type s_2
    - fraction: 50
      properties:
        - label: NETWORK_TRANSMIT
          bucket: LOW
          fraction: 100
---
```

At this point, the fully colored graph defines the complete application topology and communication structure. However, the computational aspects of the microservices are still missing. Hence, we assign each microservice (i.e., color) a particular *programming language* and *service type*. For instance, Listing 3 defines Rust and Python as available programming languages for the application. In this case, the configuration specifies a selection probability of 70% and 30% for Python and Rust, respectively.

A service type specifies the desired characteristics for a microservice using the available handler function labels. In Listing 3, the first service type defines the utilization of a specific resource (`CPU`) from a given bucket (`HIGH`) for a `fraction` of the microservice's endpoints. Buckets are the categorical abstractions of the numeric label values. During the generation, we dynamically map the label values to the three buckets *LOW*, *MEDIUM*, and *HIGH*. For this, we sort the handler functions by the particular label and partition this sorting into three equally sized chunks. Listing 3 specifies two service types s_1 and s_2. According to the configuration, the expected distribution of these two service types is 50% each. While service type s_1 represents a CPU-intensive service, service type s_2 corresponds to a service with low outgoing network traffic. In other words, the endpoints of a service of type s_2 send small response payloads. Even though our example does not depict this, a service type may specify multiple labels for selecting handler functions.

Selecting a handler function for every vertex in the graph results in a fully defined *application graph*. Specifically, the assignment of handler functions to the graph vertices defines the expected request payloads for each application endpoint. Thus, the handler function of the target vertex for each edge determines the required request data for that particular inter-service call.

3.4 Translating the Application Graph into Executable Code

In the next step, we use the *Handlebars template engine*[3] to dynamically generate the source code for each microservice based on the application graph. While the role of the handler function is to define the workload to be executed when a request arrives, the template takes care of assembling handler functions into an executable microservice. With the templates, the framework generates the source code for (1) parsing the function's input from the payload and query data from the network request, (2) invoking the assigned handler function with this input, (3) performing the inter-service calls, and (4) returning the handler function's result in the response payload. Each endpoint also generates random request payload data for its inter-service calls. As described previously, this request payload must match the signature definition (see Listing 2) of the handler function assigned to the target endpoint of the inter-service call. The template also takes care of importing required libraries and makes the handler function accessible through network requests. In general, at least one template is required for every supported programming language. For example, we use a template for Python that creates microservices based on the FastAPI[4] library, while our Rust template uses the Axum[5] library.

[3] https://docs.rs/handlebars/5.0.0/handlebars/index.html
[4] https://fastapi.tiangolo.com/
[5] https://docs.rs/axum/latest/axum/

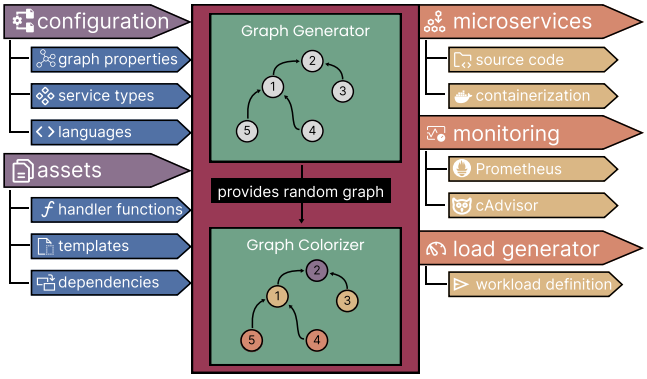

Figure 3: High-level overview of our framework including its inputs (left) and outputs (right)

3.5 Summary of the Input and Output Artifacts

In summary, we generate a hierarchy of source code and project files from a combination of configuration files and assets.

Figure 3 illustrates the input and output artifacts as well as Creo's internal representation of an application. The assets include *handler functions*, *templates*, and *dependencies*. *Handler functions* implement small, independent pieces of business logic. They do not contain any web server-specific code. The *templates* are the basis for the dynamic source code generation. *Dependencies* are auxiliary container images, such as databases, that the implementation of a *handler function* may require.

The configuration comprises *graph properties*, *service types*, and *languages* for the autopilot mode. The *graph properties* specify constraints for the graph generation (i.e., the number of vertices, edges, and colors). The *service types* define the resource usage characteristics of the application hierarchically (as shown in Listing 3). Lastly, *languages* determine the available programming languages and their selection probabilities. The manual mode requires the specification of the complete application graph.

The framework generates a graph with the given number of vertices (endpoints) and edges (inter-service calls). The proper vertex coloring then assigns each *endpoint* to a *microservice*. Various outputs are generated from this internal application representation.

A directory for every generated service contains the source code to run it. The output also includes a `Dockerfile` for building the service image and a Docker Compose file for starting the service container with the required dependencies, such as databases. Using the official `kompose`[6] software, the Docker Compose files can be translated to Kubernetes deployments. The preconfigured monitoring harness comprising Prometheus[7] and cAdvisor[8] facilitates collecting performance metrics of every service in the application. Lastly, the generator provides workload definitions for the HTTP Load Generator[9] enabling sending user requests, further automating the benchmarking process.

[6]https://kompose.io/
[7]https://prometheus.io/
[8]https://github.com/google/cadvisor
[9] https://github.com/joakimkistowski/HTTP-Load-Generator

Creo offers a command-line interface for several useful functionalities, such as deploying the generated application and load generator to distinct servers or starting specified experiments on the deployment servers from the researcher's local machine.

3.6 Extensibility of Creo

The framework's design aims to be as generic as possible, allowing for the change of almost every aspect of the generation process. The most relevant extension for most users is the addition of further handler functions. This requires minimal effort and is as simple as implementing a new function in an already supported language and defining the input specification similar to Listing 2. Including new labels (e.g., failure rate) is a simple yet powerful way to influence the application generation process. While adding a new web server framework or programming language naturally requires more effort, we provide thorough documentation of the process in our GitHub repository.[1] We also plan to include more languages soon. The generation process can be modified for particular needs by implementing new strategies for selecting graph edges, coloring the graph, or partitioning the handler functions into buckets. For instance, a clustering algorithm could assign handler functions to the three buckets instead of equally sizing the bins. In summary, our approach is very flexible by design and allows researchers to extend and modify the framework with varying levels of complexity and impact.

3.7 Validation of Creo

Naturally, a code generation approach requires thoroughly validating all components and artifacts. In addition, this work focuses on generating microservice applications with configurable resource usage characteristics. In our case, we split the validation into two parts: (1) a use case-independent validation focusing on the correctness and validity of the generation process, the generated applications, and the generated side artifacts, and (2) a use case-bound validation showing the value of our approach in a practical scenario. Due to space constraints, we cannot provide the full details and results of our use case-independent validation. Further information is included in Appendix A and our GitHub repository.[1] The following list outlines our use case-independent validation efforts:

- Unit tests for core components, such as adherence to graph properties or coloring algorithm
- Validation of the handler function selection algorithm
- Evaluation of the generated source codes, including parameter parsing, handler function invocation, and inter-service calls
- Evaluation of the generated side artifacts, including containerization, monitoring, and load generation
- Validation and effectiveness of the resource labels

In the following, we showcase the value of our approach in two case studies using generated applications with diverse performance characteristics.

Table 1: Workloads and resource limits of applications used in the baseline dataset

Application	CPU	MEM	Traffic
Cassandra	6	28 GB	A: 150-5000
Cassandra	6	28 GB	B: 100-3000
Cassandra	6	28 GB	D: 300-10000
Cassandra	1	28 GB	F: 200-2000
Memcached	8	32 GB	40000-200000
Memcached	8	6 GB	30000-95000
Memcached	1	32 GB	50000-200000
Memcached	1	3 GB	50000-180000
Solr	8	14 GB	sin1000
Solr	3	32 GB	sin1000
Solr	3	14 GB	sin1000
Solr	3	8 GB	sin1000

4 Use Case I: Performance Degradation Prediction

In our first use case study, we demonstrate the value of generated applications for enriching the training dataset of an ML-based performance degradation predictor. Monitorless [13] is an application-agnostic approach that uses platform-level metrics as features and different applications for training and inference. However, the approach's general applicability relies on diverse applications to obtain the training data, making it an attractive use case for our approach.

Monitorless's core idea is to detect saturation in applications leading to degraded key performance indicators (KPIs). During training, the resource consumption of different applications under various configurations and workloads is measured. These measurements ideally cover scenarios that are potential bottlenecks for all platform resources. Monitorless assumes that an application reaches its saturation point at a certain workload intensity. After this point, the relationship between the KPI (e.g., throughput) and the workload is non-linear, and the application is labeled as saturated.

Since the authors do not provide their implementation as an artifact, we implement the approach ourselves. First, we replicate the results of the original work to obtain reference results and ensure our setup is comparable. We conduct all measurements on a *e2-custom* VM in the Google Cloud running Ubuntu 24.04 LTS as the operating system. The VM has 32 vCPUs, 48 GB of memory, and 16 Gbps networking bandwidth. We use Prometheus and cAdvisor to collect platform-level metrics and use throughput measurements from the HTTP Load Generator[9] for labeling.

Table 1 shows our experiments with the three applications used in [13] to obtain training data. These data constitute our *baseline dataset*. The experiments are similar to the original work, with the intensities adjusted to the capabilities of our test environment. In the table, the traffic patterns *A*, *B*, *D*, and *F* for Cassandra correspond to the workloads defined by [5] that use a different mix of read, write, update, and insert operations. The *Traffic* column shows the minimum and maximum requests per second (RPS) of the linearly increasing workload for Cassandra and Memcached.

Table 2: Workloads and resource limits of the generated applications

Application	CPU	MEM	Traffic
CPU	32	48 GB	25-300
Memory	32	48 GB	25-300
Network Transmit	32	48 GB	25-400
Network Receive	32	48 GB	50-600

Table 3: Feature importances of the trained classifiers

Feature Category	Baseline	Enriched
CPU	50.96%	42.10%
MEMORY	20.52%	21.74%
NETWORK RECEIVE	32.05%	38.46%
NETWORK TRANSMIT	14.78%	18.85%
FILE SYSTEM	32.67%	27.55%

For Solr, *sin1000* describes a sinus workload pattern between 1 and 1000 RPS. Like the authors, we use TeaStore as the test application.

In addition, we enrich the baseline dataset with measurements from four generated applications. Generally, there are two strategies for generating applications to improve the predictor's performance. In the first case, we already know or can estimate the resource profile of the considered production application for which resource saturation should be predicted. Here, we can intentionally generate applications with similar performance characteristics to ensure that the predictor trains on representative data. The second strategy applies when the characteristics of the targeted production application are unknown. In this case, we rely on the diversity of our training data to maximize the generalizability of the trained ML model.

This paper considers the latter of the two cases, as we do not want to artificially optimize the results for our test application (i.e., TeaStore). Instead, we want to showcase Creo's ability to generate diverse applications. We enrich the training data so that 25% of the samples come from generated applications, while the remainder is the baseline dataset. We use resource labels to influence the generation process. We generate four applications with either *HIGH* CPU, memory, network transmit, and network receive usages to cover various resource bottlenecks. Similar to the previous experiments, we use linearly increasing workloads. Table 2 lists the experiments for the generated applications. The application's name corresponds to the resource set to *HIGH* during generation.

We adopt the original work's preprocessing pipeline and evaluation metric—the *lagged* F_1 score. While the platform metrics immediately increase for a saturated application, the KPI degradation is observed later due to the inherent delay of response time measurements. The *lagged* F_1 score accounts for this delay. As in the original work, our request timeout is three seconds; consequently, our used lag is also three seconds.

With the baseline dataset, the random forest predictor achieves a *lagged* F_1 Score of 69.4% on the TeaStore dataset, which is very close to the authors' reported score of 71.2%. Training the model

on the enriched training dataset results in an increased *lagged* F_1 Score of 74.5% on the identical test data. This improvement shows that our generated applications could enrich the existing training data in a valuable way. Further, we analyzed the feature importances reported by the random forests. Table 3 shows the sum of feature importances grouped by their corresponding resource(s). The preprocessing adds new features by combining features related to different resources. Hence, the total sum exceeds 100%. The data indicates that the classifier trained on the baseline dataset is biased towards CPU-related features, with an overall importance value of over 50%. In contrast, the classifier trained on the extended training data exhibits a more balanced importance distribution. Notably, the importance of network-related features is increased. These are underrepresented in the baseline dataset. Overall, this widens the trained classifier's applicability while improving its results on the TeaStore training dataset. In conclusion, this case study demonstrates how generated applications can valuablely enhance the training dataset for an ML model.

5 Use Case II: Interference Detection

As discussed in Section 2, it is laborious and time-consuming for researchers to evaluate their approaches on microservice applications with various resource characteristics. In our second use case study, we show that generated applications can extend the evaluation scope, allowing us to assess the general applicability of a proposed approach. Baluta et al. [4] propose a machine learning classifier for detecting changes in application performance for a *target application* due to resource contention.

The approach is split into three phases: baseline runs, interference probes, and model training. First, we conduct multiple repetitions of baseline benchmarks at various workload intensity steps with our *target application* and monitor its average response time. We construct the 95% confidence interval (CI) of the response time for each workload intensity step. These CIs confine the expected application behavior without interference and are used for labeling purposes, that is, to determine whether the application experiences resource contention. In the second step, we repeat the workload steps from the baseline phase once without interference and in combination with various controllable interference levels. If the measured response time is above the upper limit of the constructed CI for the corresponding workload intensity, the sample is labeled as interference.

Like Baluta et al. [4], we conduct the baseline and interference benchmarks for 120 seconds and increase the workload intensity in five steps while repeating the baseline measurements 40 times each. We use their first target application, AcmeAir [1], while stress-ng[10] acts as the controllable, interfering application. Again, the authors do not provide a publicly available implementation of their proposed approach. Hence, we implement the described methodology and replicate the experiments from the original work to obtain reference results for our environment. Our measurements consider seven levels of interference with stress-ng using between one and seven CPU cores. Prometheus and cAdvisor collect platform-level

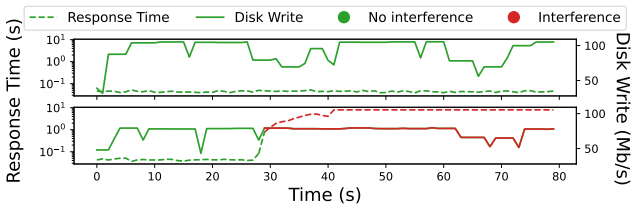

Figure 4: Response time and disk usage measurements of *Disk Write*

metrics like CPU and network usage. We use the HTTP Load Generator's response time measurements. We collect samples from $5 \cdot 8 = 40$ benchmarks for the five workload intensity steps.

However, as the scenarios in the original work only consider CPU contention, we want to evaluate the approach's applicability to different kinds of resource interference. We chose to evaluate three additional cases: outgoing and incoming network contention, as well as contention for disk writes. Thus, we generate three applications, namely *Network Transmit*, *Network Receive*, and *Disk Write*, for which we configure a single service type with the corresponding resource usage intensity set to *HIGH*. Our experiments use tools to limit the target application's available network and disk bandwidth to induce controllable interference. For network interface contention, we use the Linux kernel function `tc` to limit the outgoing or incoming bandwidth for *Network Transmit* and *Network Receive*, respectively. Similarly, we use the `blkio` configuration options of Docker to limit the disk write speed for *Disk Write*. Again, we consider seven interference levels for all three applications, resulting in 40 benchmarks per application. For every level of interference, we linearly lower the available bandwidth.

Figure 4 illustrates the approach with representative measurement results of *Disk Write*. The upper plot shows the disk write throughput and response time without interference for a workload of 75 RPS. We observe a throughput of about 100 Mb/s and a steady response time. In contrast, the lower plot shows the results for the same workload but with the write bandwidth lowered by 20% due to interference. We observe that the throughput hits this limit, and the response time increases significantly. The line colors illustrate the derived labels: interference (red) and no interference (green).

Like the original work, we perform a 80/20 stratified train-test split and use the H2O AutoML framework[11] for model training. Throughout the experiments in this section, we use a *e2-standard* VM in the Google Cloud with Ubuntu 24.04 LTS as the operating system for running our target and interfering applications. The VM has eight vCPUs and uses balanced persistent disks as its block storage.

Table 4 shows that the model trained on our measurements achieves a higher F_1 score (90.06%) for AcmeAir than the model reported in [4] (81.64%). Our model misses fewer cases of interference, resulting in a higher recall score, but its interference predictions include more false positives, resulting in a lower precision score. Based on the results in the table, the proposed approach can also

[10] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[11] https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html

Table 4: Evaluation metrics for all applications

Application	F_1 Score	Precision	Recall
AcmeAir (as in [4])	81.64%	100.0%	68.98%
AcmeAir	90.06%	92.55%	87.70%
Network Transmit	96.24%	95.09%	97.42%
Network Receive	82.71%	79.70%	85.94%
Disk Write	66.81%	66.67%	66.96%

detect network-related interference. The model for *Network Transmit* performs best with an F_1 Score of 96.24%. This is unsurprising, as increased response times directly impact network-related metrics (e.g., the transmitted network bytes decrease). In the *Network Receive* scenario, the predictor achieves similar scores to the model trained with AcmeAir. Lastly, our findings indicate that the proposed approach struggles to detect interference for disk write-heavy workloads. The corresponding model reaches a substantially lower F_1 Score (66.81%) compared to all other predictors. The precision score shows that one-third of the predicted interference cases are unrelated to actual interference.

In summary, this case study illustrates that generated applications can reveal strengths and weaknesses of an approach that might stay undiscovered when fewer, less diverse test applications are used. With Creo, researchers can create and benchmark further microservice applications for their approaches, each with its distinct resource usage profile.

6 Discussion

This section discusses this work's strengths, weaknesses, and threats to validity.

Strengths: Unlike most existing microservice application generators, our proposed approach provides a language-agnostic, extensible framework for generating *executable* microservices by composing functions with non-trivial business logic. In contrast, many other code generators produce code parts or skeletons only. As shown earlier, generated applications can be valuable in various contexts, even beyond performance benchmarking use cases (e.g., software visualization [12], or architecture recovery [3]). The uniqueness of our approach is based on three main characteristics: (1) its multi-language, characteristics-driven generation process, (2) its built-in extensibility, and (3) its user-friendly design with end-to-end support for performance benchmarking. Flexible labels are at the core of our approach, enabling many use cases. For instance, this work demonstrates the usage of resource labels in two use case studies.

While other generators require explicitly stating the resource consumption and other properties of the microservices as deterministic values (e.g., the execution time in seconds [21]), our configuration uses coarse-grained specifications of resource usage characteristics or other user-defined labels with three intensity levels. This is a strength of our approach, as the configuration becomes more intuitive, declarative, and less restrictive. This is desired since users of our approach like to generate many diverse applications. In addition, the manual mode of Creo enables fine-granular control of the composition of generated applications. The preconfigured benchmark harness and other generated side artifacts shown in Figure 1 enable experimenting with multiple microservice applications with minimal effort. Researchers familiar with the configuration can generate, deploy, and start benchmarks within minutes. The generation process is very scalable. For instance, we generated an application with 150 microservices, 450 endpoints, and 300 inter-service calls in just over three seconds. Our approach focuses on being extensible regarding the supported programming languages, web frameworks, and—most importantly—handler functions. Once an application is generated, researchers can easily customize monitoring settings, workload mixes, or even the application source code to meet specific requirements. All generated files follow best practices in terms of structure and format.

Weaknesses: A current downside of our approach is the limited number of supported programming languages and handler functions. In general, the diversity and applicability of the generated applications heavily rely on the set of implemented handler functions. Handler functions are the core building blocks of our approach, and their dynamic composition into microservices allows our approach to generate a diverse set of applications. The usefulness of our approach increases with more available handler functions and programming languages. Nevertheless, this is primarily an implementation effort and is not a general limitation of our approach. As our approach is open-source, we envision a user community that collectively extends the framework over time. We are actively working on adding Java and Go as available programming languages soon. Adding new handler functions of already supported programming languages is as easy as implementing a single function/method. Adding new programming languages or web frameworks requires new templates and supporting scripts to generate the side artifacts (e.g., Dockerfiles). This step requires solid knowledge of the target language and understanding of Creo's internals but is feasible with some experience.

While our approach for configuring the application properties via labels is very general, our current resource labeling process relies on measurements from the same environment (i.e., hardware configuration). We must ensure that the resource label experiments are run under the same conditions to keep resource labels comparable among functions implemented in the same programming language. This requirement forces users to re-label all functions of a particular programming language once they want to add new handler functions to the framework. This is neither resource- nor time-efficient. In the future, we plan to develop a feasible, hardware-independent labeling process. However, this weakness only applies to use cases relying on resource labels. Use cases with other relevant labels do not suffer due to the hardware-dependent performance labels.

Even though we aim to design Creo as flexible as possible, the inherent complexity of such a framework may negatively impact its adaptability to specific use cases. However, we provide documentation for all parts of the framework in addition to support via GitHub issues and discussions.

Threats to validity: Our case studies demonstrate that generated applications can be valuable in use cases involving applications

with different resource usage characteristics. Our experiment results have been conducted in a public cloud environment and are subject to variability [16, 17]. Further experiments in different environments might show different results. As the authors of the original works have not published their code, we had to implement their approaches ourselves, questioning the comparability to the original works. We addressed this concern by repeating the experiments described in the original works. Overall, we envision the Creo framework as a versatile tool for a wide range of use cases. The concrete benefits of using Creo have to be assessed in each scenario. Some use cases, such as the ones shown in this paper, profit from using generated applications, while others might not. However, our approach is designed to be highly adaptable, user-friendly, and scalable, promoting its potential to assist researchers in various fields.

7 Related Work

Microservice Benchmark Applications. Seshagiri et al. [22] published a survey identifying mismatches between microservice reference applications and the industrial perception of microservices. The reviewed microservice testbeds comprise DeathStarBench [6, 10]—with the applications Social Network, Movie Service, and Hotel Reservation—TrainTicket [9, 33], BookInfo [14, 15], μSuite [25, 26], and TeaStore [29, 30]. Other commonly used reference applications include AcmeAir [1], SockShop [32], and Online Boutique [11]. Among the reference applications, the number of services ranges from one to 41 microservices, while the used programming languages include C, C++, Go, Java, Node.js, Ruby, Python, and C#. Most reference applications are limited to a single and, at most, four programming languages. Most applications include Docker images for simplified deployment. However, they rarely include load-generation clients and ready-to-use harnesses for performance benchmarks. Consequently, the benchmarking setup is often labor-intensive and requires application-specific knowledge.

Our work differs from these reference applications in various aspects. First, the mentioned works focus on implementing business logic from real-world use cases (e.g., online shops). In contrast, our work creates applications with diverse user-defined properties of interest (e.g., resource usage characteristics), while the actual business logic is secondary. Further, the reference applications each feature one technology stack and topology graph. In contrast, our work allows the creation of microservice applications with diverse dependency graphs and programming languages.

Microservice Application Generation. HydraGen [21] is a microservice benchmark generator focusing on inter-service communication. A dummy microservice implemented in Python emulates customizable microservice applications based on configurable computational complexity and topology. Additionally, the configuration includes the traffic characteristics between paired microservices and the deployment characteristics relating to microservice placement and scalability.

MicroService Topology Generator (MSTG) [31] is a generator explicitly focusing on including configurable network architectures. The generated applications comprise replicas of a single microservice written in Go. If configured, a replica sequentially calls other services before returning random data of a given size. Also, routers with configurable network properties can be included in the generated topology.

μBench [7] is an open-source tool for generating dummy microservice applications for benchmarking cloud/edge computing platforms. The tool generates Python applications with configurable topology, resource usage characteristics, and resource limits. The generator includes configuration files for executing the generated applications in a Kubernetes cluster.

HydraGen vs. MSTG vs. μBench vs. Creo. Table 5 shows our qualitative comparison of HydraGen, MSTG, μBench, Creo, and a typical microservice benchmark application, e.g., TrainTicket. While the presented generators share some commonalities, Creo significantly differs from existing approaches. Some reference applications (e.g., DeathStarBench) have multi-language support. However, to the best of our knowledge, our approach is the only generator supporting multi-language microservice generation. HydraGen and MSTG primarily focus on network-related properties of microservice applications. In contrast, Creo does not come with any particular focus but rather the ability to extend and customize the generator to one's specific needs.

μBench mentions the support for adding new handler functions and arbitrary software through sidecar containers. However, these functions must be implemented in Python and their argument values must be set at generation time. Additionally, invoking sidecar containers always involves calling a Python function for the endpoint. Creo allows the inclusion of new handler functions and offers extensibility as a core feature of the framework. For instance, new programming languages (e.g., Java, Javascript, or C#) or new web frameworks (e.g., Flask) for existing languages may be added. Additionally, the generation process itself may be extended by implementing different strategies for graph generation, such as the Barabási-Albert algorithm used in μBench, using new strategies for assigning handler functions to graph vertices or exchanging the labeling process.

HydraGen and μBench use dummy functions as a means to stress resources (e.g., a busy wait that is executed for x number of seconds). Conversely, our approach uses handler functions executing typical web service workloads (e.g., CRUD operations).

8 Conclusion

In this paper, we present Creo, a framework for generating microservice applications that (1) are fully executable, (2) have configurable properties and resource usage profiles, and (3) have built-in support for standardized monitoring, load generation, and deployment. We explain the generation process in detail and highlight the configurability of our approach. Furthermore, we demonstrate the applicability of generated applications through two case studies. In the first case study, we show that enriching the training data for a state-of-the-art approach for performance degradation prediction improves the classifier's performance and widens its applicability. In the second case study, we extend the evaluation of a recent approach for performance anomaly classification using a set of generated applications. The original evaluation exclusively considered CPU contention. Our extended evaluation reveals that

Table 5: Qualitative comparison between a typical microservice benchmark application, HydraGen, MSTG, μBench, and Creo

	Benchmark Application	HydraGen	MSTG	μBench	Creo
Multi-language support	Yes	No	No	No	Yes
Resources	CPU, Memory Network(in/out), Disk	CPU, Network (in/out)	Network (out)	CPU, Memory Network (out), Disk	CPU, Memory Network (in/out), Disk
Resource Configuration	N/A	Numeric	Numeric	Numeric	Flexible
Inter-Service Call Mode	Sync, Async	Sync, Async	Sync	Sync, Async	Sync, Async
Dynamic Request Size	Runtime	Generation Time	No	No	Runtime
Dynamic Response Size	Runtime	Generation Time	Generation Time	Generation Time	Runtime
Handler Functions	Fixed, multi-language	Fixed, Python	None	Fixed, Python	Extensible, multi-language
Handler Function Type	Single business use case	Dummy	N/A	Dummy	Arbitrary business use case
# Microservices	Fixed	Configurable	Configurable	Configurable	Configurable
# Endpoints per Service	Fixed	Configurable	Configurable	1	Configurable
# Inter-service Calls	Fixed	Configurable	Configurable	Configurable	Configurable
External Dependencies	No	No	No	Yes	Yes

the proposed approach works well for outgoing and incoming network interference; however, write-heavy disk workloads seem to represent a weakness.

In summary, our approach enables researchers to conduct experiments with diverse microservice applications with minimal effort and holds great potential for future applications. We plan to extend the use cases of our framework and evaluate its applicability in novel scenarios, such as developing transfer learning approaches in the context of microservice applications.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 536362030. We thank the Google Cloud Research Credits program (GCP341998600) for supporting this work.

References

[1] AcmeAir. 2013. *AcmeAir.* AcmeAir. Retrieved June 4, 2024 from https://github.com/acmeair/acmeair

[2] Işıl Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdoğan. 2021. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software* 180 (Oct. 2021), 111014. https://doi.org/10.1016/j.jss.2021.111014

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2018. Towards Micro Service Architecture Recovery: An Empirical Study. In *2018 IEEE International Conference on Software Architecture (ICSA).* IEEE, New York, NY, 47–4709. https://doi.org/10.1109/icsa.2018.00014

[4] Alexandru Baluta, Yar Rouf, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. 2024. Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering* (London, United Kingdom) *(ICPE '24).* Association for Computing Machinery, New York, NY, USA, 286–297. https://doi.org/10.1145/3629526.3645046

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10).* Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Christina Delimitrou. 2019. *DeathStarBench.* Cornell University. Retrieved June 4, 2024 from https://github.com/delimitrou/DeathStarBench

[7] Andrea Detti, Ludovico Funari, and Luca Petrucci. 2023. μBench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 968–980. https://doi.org/10.1109/TPDS.2023.3236447

[8] Javad Dogani, Reza Namvar, and Farshad Khunjush. 2023. Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. *Computer Communications* 209 (2023), 120–150. https://doi.org/10.1016/j.comcom.2023.06.010

[9] FudanSELab. 2020. *TrainTicket.* Fudan University. Retrieved June 4, 2024 from https://github.com/FudanSELab/train-ticket

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. 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* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[11] GoogleCloudPlatform. 2018. *OnlineBoutique.* Google Cloud Platform. Retrieved June 4, 2024 from https://github.com/GoogleCloudPlatform/microservices-demo

[12] Mia E. Gortney, Patrick E. Harris, Tomas Cerny, Abdullah Al Maruf, Miroslav Bures, Davide Taibi, and Pavel Tisnovsky. 2022. Visualizing Microservice Architecture in the Dynamic Perspective: A Systematic Mapping Study. *IEEE Access* 10 (2022), 119999–120012. https://doi.org/10.1109/access.2022.3221130

[13] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19).* Association for Computing Machinery, New York, NY, USA, 149–162. https://doi.org/10.1145/3361525.3361543

[14] Istio. 2017. *BookInfo.* Istio. Retrieved June 4, 2024 from https://github.com/istio/istio/tree/master/samples/bookinfo

[15] Istio. 2019. *BookInfo.* Istio. Retrieved June 4, 2024 from https://istio.io/latest/docs/examples/bookinfo/

[16] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software Microbenchmarking in the Cloud. How Bad is It Really? *Empirical Software Engineering* 24, 4 (Aug. 2019), 2469–2508. https://doi.org/10.1007/s10664-019-09681-1

[17] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos–A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497

[18] Yannik Lubas, Martin Straesser, André Bauer, and Samuel Kounev. 2025. Generating Executable Microservice Applications for Performance Benchmarking (Supplementary Materials). https://doi.org/10.24433/CO.3331658.v2

[19] Walter Meyer. 1973. Equitable Coloring. *The American Mathematical Monthly* 80, 8 (Oct. 1973), 920–922. https://doi.org/10.1080/00029890.1973.11993408

[20] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (July 1900), 157–175. https://doi.org/10.1080/14786440009463897

[21] Mohammad Reza Saleh Sedghpour, Aleksandra Obeso Duque, Xuejun Cai, Björn Skubic, Erik Elmroth, Cristian Klein, and Johan Tordsson. 2023. Hydragen: A Microservice Benchmark Generator. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD).* IEEE, IEEE, New York, NY, 189–200. https://doi.org/10.1109/CLOUD60044.2023.00030

[22] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R Sambasivan. 2022. [SoK] Identifying Mismatches Between Microservice Testbeds and Industrial Perceptions of Microservices. *Journal of Systems Research* 2, 1 (March 2022), 27 pages. https://doi.org/10.5070/sr32157839

[23] Jacopo Soldani and Antonio Brogi. 2022. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Comput. Surv.* 55, 3, Article 59 (feb 2022), 39 pages. https://doi.org/10.1145/3501297

[24] Gagan Somashekar, Anurag Dutt, Rohith Vaddavalli, Sai Bhargav Varanasi, and Anshul Gandhi. 2022. B-MEG: Bottlenecked-Microservices Extraction Using

Graph Neural Networks. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 7–11. https://doi.org/10.1145/3491204.3527494

[25] Akshitha Sriraman and Thomas F Wenisch. 2018. . University of Michigan. Retrieved June 4, 2024 from https://github.com/wenischlab/MicroSuite

[26] Akshitha Sriraman and Thomas F Wenisch. 2018. μ Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, New York, NY, 1–12. https://doi.org/10.1109/IISWC.2018.8573515

[27] Martin Straesser, Stefan Geissler, Stanislav Lange, Lukas Kilian Schumann, Tobias Hossfeld, and Samuel Kounev. 2025. Trust your local scaler: A continuous, decentralized approach to autoscaling. *Performance Evaluation* 167 (2025), 102452. https://doi.org/10.1016/j.peva.2024.102452

[28] Luca Traini and Vittorio Cortellessa. 2023. DeLag: Using Multi-Objective Optimization to Enhance the Detection of Latency Degradation Patterns in Service-Based Systems. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3554–3580. https://doi.org/10.1109/tse.2023.3266041

[29] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. *TeaStore*. University of Würzburg. Retrieved June 4, 2024 from https://github.com/DescartesResearch/TeaStore

[30] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. 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* (Milwaukee, WI, USA) *(MASCOTS '18)*. IEEE, New York, NY, 223–236. https://doi.org/10.1109/MASCOTS.2018.00030

[31] Emilien Wansart, Maxime Goffart, Justin Iurman, and Benoit Donnet. 2024. MSTG: A Flexible and Scalable Microservices Infrastructure Generator. arXiv:2404.13665 [cs.NI] https://arxiv.org/abs/2404.13665

[32] WeaveWorks. 2016. *SockShop*. WeaveWorks. Retrieved June 4, 2024 from https://github.com/microservices-demo/microservices-demo

[33] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 323–324. https://doi.org/10.1145/3183440.3194991

A Validation Details and Results

A.1 Unit Tests for Core Components

We use unit tests for the core components of our framework to validate assumptions and increase confidence in the implementation's correctness. For both the autopilot and manual mode, we validate the following properties:

- The generation of the graph is successful.
- The generated graph is acyclic.
- The vertex count of the generated graph matches the configured number of endpoints.
- The edge count of the generated graph matches the configured number of inter-service calls.
- The color count of the coloring matches the configured number of microservices.
- The coloring is a valid coloring for the generated graph.
- (*Manual mode only*:) The generated graph only contains the configured edges.

This ensures that users of our framework receive the correct and configured results from the graph generation process.

A.2 Validation of the Handler Function Selection Algorithm

As part of the validation of our framework, we want to show the adherence of the handler function selection algorithm to the user-provided configuration. Using unit tests, we validate the following properties for each entity (e.g., programming language, label, service type, and definition) in the configuration:

- An entity with a fraction value of zero should never be selected
- An entity with a fraction value of 100 should always be selected
- We use the χ^2 distribution test [20] to test our null hypothesis that the selections of our algorithms follow the distribution given by the configuration

We illustrate the general structure of the χ^2 distribution tests with the following example for testing the selection of programming languages. The Pearson's χ^2 test statistic is given by:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{1}$$

where O_i is the number of observations of category i, E_i is the expected number of observations of category i, and n is the total number of categories. The expected number of observations of a category i is determined by:

$$E_i = N \cdot p_i \tag{2}$$

where N is the total number of observations, and p_i is the probability of category i given by the null hypothesis H_0.

In our example, the categories refer to available programming languages. The test comprises $n = 2$ categories, namely, Rust and Python. According to the test configuration, Rust has a 0.7 probability, while Python is selected with a 0.3 probability. Accordingly, we formulate our null hypothesis H_0 that we choose the languages according to their configured probabilities. We draw a total of $N = 100000$ number of observations. Using Equation (2), we can compute the expected number of observations for Rust ($0.7 \cdot 100000 = 70000$) and Python ($0.3 * 100000 = 30000$), respectively. We use the χ^2 test statistic given by Equation (1) and a significance level of $\alpha = 0.05$ to test our null hypothesis H_0. We evaluate the inverse cumulative distribution function (ICDF) of a χ^2 distribution with $f = n - 1$ degrees of freedom at $p = 1 - \alpha$. In our example, $f = 2 - 1 = 1$ and $p = 1 - 0.05 = 0.95$. If our test statistic χ^2 is smaller than the value of the ICDF at p, our null hypothesis holds. A larger test statistic indicates a significant difference (with $\alpha = 0.05$) from our expected distribution of observations, and we have to reject H_0 as false. Using this χ^2 distribution test, we ensure that all selections of our algorithm do not significantly deviate from the configured probabilities.

A.3 Evaluation of the Generated Artifacts

As part of the framework validation, we also have to ensure the correctness of the generated artifacts. We validate the generated source code for several sample applications for the following properties:

- The query and body payloads are correctly parsed.
- The selected handler function is correctly invoked with the payload data.
- The inter-service calls are correctly invoked.
- The inter-service calls can access the connected microservices via the network.

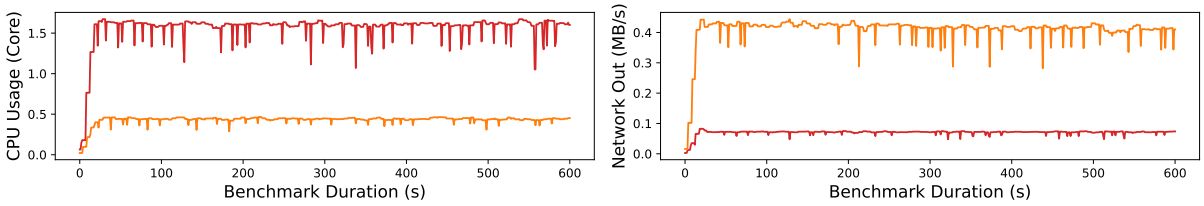

Figure 5: CPU usage of *service-1* and outgoing network traffic of *service-2* for generated applications *app-1* (red) and *app-2* (orange)

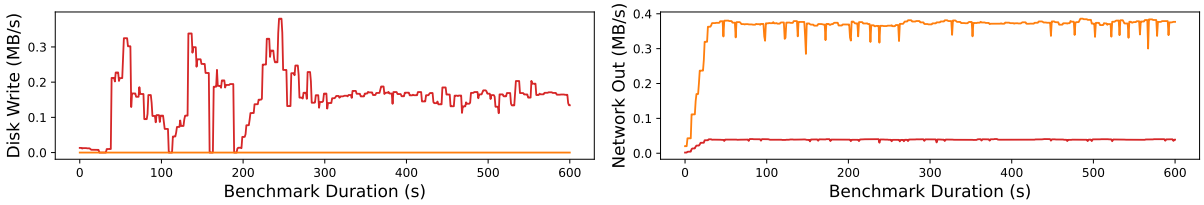

Figure 6: Disk (write) utilization of *service-1* and outgoing network traffic of *service-2* for generated applications *app-3* (red) and *app-4* (orange)

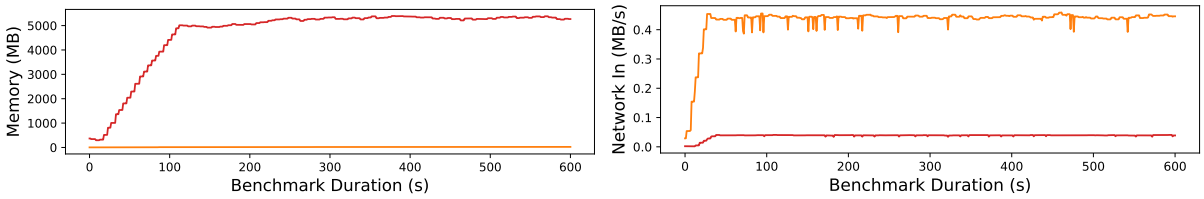

Figure 7: Memory utilization of *service-1* and incoming network traffic of *service-2* for generated applications *app-5* (red) and *app-6* (orange)

- A container image can be successfully built using the generated `Dockerfile`.
- The generated `compose.yml` file includes a correctly configured Prometheus container.
- The generated `compose.yml` file includes a correctly configured cAdvisor container.
- The container can be successfully started.
- The user-facing endpoints, i.e., the endpoints with no incoming inter-service calls, correctly respond to valid requests.

This validation, in combination with the previous validation steps, ensures the correct end-to-end generation of applications based on the user configuration. Consequently, we can be confident in the correctness of the generated applications for our use case studies.

A.4 Validation and Effectiveness of Resource Labels

This section details the results of validating the resource labels used to configure the performance characteristics of generated applications. Since our configuration does not use numeric values (e.g., milliseconds of a busy wait or number of random bytes returned), we cannot determine the expected, absolute usage values of a generated application. Hence, we need to compare the resource usage of generated applications relatively. For instance, an application configured with low CPU usage should use considerably less CPU during the same workload than an application with the same topology but configured with high CPU usage. We generate several pairs of validation applications with two microservices and four inter-service calls each. For example, we generate the first application (*app-1*) of one pair with the service type definitions of Listing 3. Using logs, we verify that one of the services corresponds to service type s_1 (*service-1*), while the other service corresponds to service type s_2 (*service-2*). Using the random seed of the first application, we generate the second application (*app-2*) of this pair with the

Table 6: Validation application pairs with the corresponding resource configuration for both services.

Application	Service-1	Service-2
app-1	CPU (HIGH)	NETWORK TRANSMIT (LOW)
app-2	CPU (LOW)	NETWORK TRANSMIT (HIGH)
app-3	DISK WRITE (HIGH)	NETWORK TRANSMIT (LOW)
app-4	DISK WRITE (LOW)	NETWORK TRANSMIT (HIGH)
app-5	MEMORY (HIGH)	NETWORK RECEIVE (LOW)
app-6	MEMORY (LOW)	NETWORK RECEIVE (HIGH)

same configuration, except for changing the *HIGH* bucket in s_1 to the *LOW* bucket and the *LOW* bucket of s_2 to *HIGH*. Consequently, we generated the same application topology but opposite resource characteristics.

We hypothesize that for a constant arrival rate, the CPU usage of *service-1* of *app-1* should be relatively higher than the utilization of *service-1* of *app-2*. Similarly, we expect a lower outgoing network traffic for *service-2* of *app-1* than for *service-2* of *app-2*. We conduct a ten-minute experiment at a workload intensity of 150 requests per second for both applications and compare the consumption of the targeted resources.

We repeat the above validation for three pairs to cover all resources, i.e., CPU, memory, network, and disk. Table 6 lists all generated validation application pairs. The two service columns describe the configured and expected resource usage of the respective services. We confirm that our hypothesis holds true for each pair to show that the configured resource usage profiles are reflected in the generated applications. Figures 5, 6, and 7 show the consumption of the targeted resources of the respective microservices. All results show that a microservice with a HIGH usage configuration for a resource uses more resources than a microservice with a LOW usage configuration.

Generally, a microservice application's performance/resource usage profile is the composite of all its microservices. A microservice's resource usage is primarily determined by its endpoints, which, in our case, invoke handler functions. The more accurate and expressive the resource labels of handler functions are, the better the resulting endpoint, microservice, and application behavior can be estimated.