# Comparison of Runtime Testing Tools for Microservices

*Abstract*—

**Recently there has been an increase in the number of software applications using the microservices architectural pattern. This is due to the benefits derived over the more traditional N-tier patterns that use monolithic designs for each of the tiers. The value of using the microservices architectural patten, particularly in the cloud, has been pioneered by companies such as Netflix and Google. These companies have created many protocols and tools to support the development of cloud-based applications. However, the testing of microservices applications continues to be challenging due to the added complexity of increased data communication between the collaborating services.**

**In this paper we provide a comparison of several open-source tools used to support the testing of microservices, describe polyglot Ridesharing microservices reference prototype, and present the results of a study that measures the overhead to test the Ridesharing application. We also describe our experiences in configuring and running the testing tools on the prototype.**

*Index Terms*—**Software testing, testing tools, microservices testing.**

## I. Introduction

Since 2014 there has been an increasing number of companies interested in adopting the Microservices Architecture pattern [1]. Due to its decentralized nature and its loosely coupled services approach is becoming a very popular architecture for developing large applications. However, this pattern needs a cloud infrastructure in order to establish communication between microservices, databases, third-party services or other external components. The network connections used in this infrastructure add complexity to testing the application and that's why it's necessary to test it at runtime in order to monitor the behavior of the different components in a production environment. This paper compares different open source tools that allow the execution and monitoring of these type of tests.

Furthermore, as in any other software development process, software testing is an important part of the overall process that needs to be considered in order to avoid costly losses in the future due to unexpected behaviors. Although, there are some guidelines to test communication between services with contract and component testing, the nature of the dynamically self-adapting microservices requires testing at runtime [Anonymous, 2014], specially the deployment of new service instances and the communication between them to evaluate the response time and the failure tolerance [2].

Some of the most representative technology companies worldwide like Netflix [3], have developed their own open source tools to support the operation and testing of microservices, e.g., Eureka, to provide discovery service, Chaos Monkey to test the resiliency of the system, among many others. Another interesting tool is Hoverfly [4], it provides support to simulate failure injection to the communications to/from an API. Allowing to write automated tests to simulate communication with other microservices. Ambassador [5] is an API gateway created by Datawire that built on Lyft's Envoy [6] proxy and communication bus, allowing microservices to easily register their public API endpoints. Envoy provides statistics about traffic and allows to monitor messages. Telepresence connects a local environment to a running Kubernetes [7] or OpenShift [8] cluster, replacing temporarily an actual running service in production with a proxy that enables the communication with the local environment. Allowing to debug or execute a service in a realistic environment.

This paper includes the following contributions:

- Provides a comparative survey of tools to support microservices testing,
- Describes the design of a polyglot Ridesharing microservices reference prototype for investigating runtime testing, and
- Discusses the results of a study that measures the overhead of runtime testing on a microservices based application.

In order to accomplish these objectives the paper is distributed as follows: on section II we cover some background information to introduce software architecture and software testing. On section III we compare different tools used to test microservices. In sections IV and V, we explain the microservices prototype and the case of study, showing the results and the discussion. Finally, we on sections VI and VII we show some related work and the conclusion of the present study.

## II. Background

In this section we provide background on monolithic and microservices software architectures and software testing.

### A. Software Architecture

Garlan and Perry [9] define software architecture as: *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time*. This definition refers to software architecture as high level structures of a software system, the discipline of creating such structures, and their documentation. These structures are needed to reason about the software system. Each structure comprises software elements, relations among them, and properties of both elements and relations.

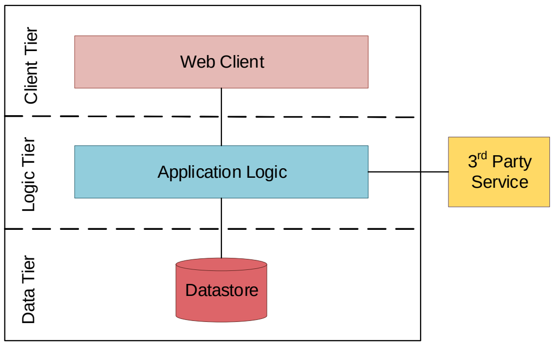

Fig. 1. 3-tier architecture with application logic monolithic component.

The work presented in this paper is centered around the microservices architectural pattern, however we initially describe the 3-tiered architectural pattern to highlight some of the differences between a traditional architecture pattern with a monolithic component and the microservices architecture pattern.

By monolithic component we mean a single component that is deployed with all the modules and libraries needed to support its execution. Dehghani [10] states that a monolithic system is composed of tightly integrated layers that are deployed together and have brittle interdependencies. Lewis and Fowler [1] define microservices as *an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.*

One of the most common architectural patterns is the layered architecture [11]. The layered architecture divides the software into layers, where each layer represents a grouping of modules that offers a cohesive set of services [12]. An example of a layered architecture is the 3-tier architecture which consist of: a client layer that handles all user interactions through the appropriate interfaces; a business logic layer that contains the business rules to process requests sent from the client layer; and the data layer that stores the persistent data for the application. Figure 1 shows the layered structure of the 3-tier architecture.

The benefits of using a layered architecture pattern are as follows. Each layer can be developed as a separate module thereby allowing independent teams to design, implement and test modules separately. It is relatively easy to deploy layered applications, since the entire application must be ready for the production environment before the release. The main drawback of the layered software architecture is that changes to the any part of the functionality in a module requires the entire module being updated and re-deployed. For example in Figure 1, if a change is required for some business rules in the application logic module (logic tier) then the entire module needs to be updated, tested and re-deployed.

The core concept in the microservices architecture is the notion of separately deployed units also known as *service components* or simply *services* [11]. Figure 2 shows an application using the microservices architecture with the application logic

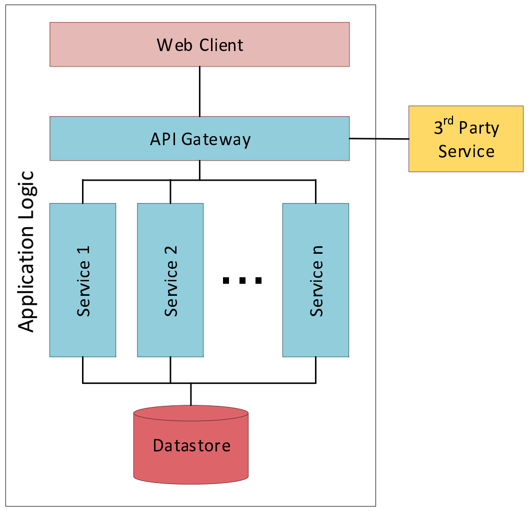

Fig. 2. Microservices architecture with application logic service components.

separated in to several services and the API gateway (also a service). It is worth noting that each service is independently deployable which is an advantage over the architecture that uses monolithic components. Each of these services will have their own environment and the most common method of communication is through a network interface.

Similar to the layered software architecture, each service can be assigned to an independent team to be designed, implemented and tested separately. However, each service can be easily implemented in a different programming language, and can be tested using specialized tools tailored to a specific type of testing. Some of the drawbacks of using a microservices architecture include the inherent need for tools that help with deploying and monitoring the status of the different microservices at runtime. Another challenge occurs during testing, that is, the testing process not only must consider testing the functionality of a service, but also the communication between services. To give an example during testing, each microservice can be working correctly, however there may be a communication problem between two services which would result in some unexpected behavior.

## B. Software Testing

The main objective of software testing is to verify the system against the expected behavior, more formally: *Software testing is the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the expected behavior'* [13]. Software testing techniques are usually classified into the following categories: black-box, white-box and gray-box. Black-box testing - tests are generated from formally and informally specified requirements and without the aid of the code under test, i.e., the code under test is treated as a black-box . White-box testing - tests are generated using the features of the code being tested and helps in determining the quality of the test set. Gray-box testing is a hybrid of black-box and white-box testing [14].

In this paper we focus on three levels of software testing which are unit, integration and system testing [13]. Within unit testing we also look at component and contract testing. We will refer to these five testing approaches as testing strategies in the paper. Each of these strategies have different objectives during the verification and validation of the software application.

*Unit testing* ensures that the unit function correctly in isolation. The unit under test is usually a method or a class and testing occurs with access to the code and the support of debugging tools [13]. *Component testing* verifies isolated components of the system without the interactions of the external components, which is similar to unit testing. The focus of component test is to verify the services provided by the component to other entities in the application [14]. Unit and component testing use a combination of black-box, white-box or gray-box testing techniques.

A contract is defined as a formal agreement expressing the rights and obligations between a client and the component providing the service. Therefore *contract testing* involves verifying the services provided by a component given the pre-conditions stated in the contract [15]. Contract testing use mainly black-box testing techniques unlike unit and component testing.

The goal of *integration testing* is to discover any problems that might arise when the previously tested components are integrated. Prior to performing integration testing, the components to be integrated are assumed to successfully tested in isolation. Note that testing components in isolation does not guarantee that the components work as expected when combined [14]. The goal of *system testing* is to ensure that all the desired behavior is contained in the system and works as specified by the requirements. This behavior includes both functional and non-functional requirements [14].

In addition to the testing strategies previously mention there is also the notion of testing objectives [13]. These objectives can be broadly classified into *functional testing* that focuses on validating the correctness of the functional requirements of the system and *non-functional testing* the non-functional requirements of the system. *End-to-End testing* is black-box functional system testing technique. *Exploratory testing* is similar to end-to-end testing but uses an ad-hoc approach to selecting test inputs. Non-functional testing includes *performance testing* (capacity and response time) and *stress testing* (system load). *Regression testing* is the retesting of a system or component to show that the software behavior is unchanged after the software is updated. *Functional testing*

## III. Testing Tools for Microservices

The increasing popularity of using microservices architectures in software applications, specifically large cloud-based applications, has resulted in several testing tools that were developed or have been tailored to test these applications. In this section, we identify the testing tools that may be used to test micrörservices using different testing strategies (unit, component, contract, integration and system), and compare a subset of these tools for each strategy.

### *A. Classification of Testing Tools*

There are several tools to support the testing of microservices, including tools to support different levels of testing (unit, integration and system), different testing objectives, and tools for scaffolding e.g., test harnesses. Table I shows a cross-section of the tools use to support the testing of microservices. In the subsequent paragraphs we describe the key aspects of the tools shown in the table. Table I consist of 8 columns, these columns from left to right are:

- ***No*** - number assigned to the tool.
- ***Name*** - name of the tool.
- ***Interface Method*** - identifies the interaction between tools and system or unit under test. *WebDriver*, supports direct calls to the web browser e.g., Appium [16]. *HTTP request*, is a communication protocol for distributed systems, e.g., Gatling [18]. *Two-way proxy*, uses two proxies to for communication between a local environment and an external system in production. e.g., Telepresence [28].
- ***Implementation*** - how the tester uses the tool to perform testing. *IDE*, integrated development environment, e.g., JMeter [20]. *Library*, set of passive functions that can be called from an application, e.g., Docker Compose Rule [17]. *Framework*, is an active application that invokes call to other applications, e.g., JUnit [21]. *Side-car + proxy*, an independent component that is attached to an applications via a proxy, e.g., Gremlin [19]. *Web browser extension*, a plugin that can be installed to provide added functionality to a browser. *CLI*, command line interface, Watir [30].
- ***Platform*** - groups of technologies used to support the execution of other applications. This column in the table list maily well-known platforms. *SPP*, Scala Platform Process - runtime system for Scala applications. *OTP*, Open Telecom Platform - collection of middleware, libraries and tools.
- ***Testing Case Language(s)*** - implementation language(s) for writing test cases. *WebDriver compatible languages*, e.g., Java, Objective-C, Perl with the Selenium WebDriver API and language-specific client libraries, among others. The other languages referenced in this column are common programming languages.
- ***Testing Objectives/Support*** - identifies the testing objectives (see Section II-B) of the tool or the infrastructure provided by the tool, e.g., Appium [16] facilitates regression and exploratory testing, while Spring Boot Starter Test [26] is a test harness.
- ***Testing Strategies*** - identifies the levels of testing including unit, component, contract, integration and system (see Section II-B).

### *B. Testing Microservices*

Unit testing of microservices focuses on testing the internal functionality of the service. The testing process is usually automated and have a large number of defined tests cases [31]. Unit testing of microservices is done using xUnit frameworks

TABLE I
TOOLS USED TO SUPPORT THE TESTING OF MICROSERVICES.

| *No.* | *Name* | *Interface Method* | *Implementation* | *Platform* | *Test Cases Language(s)* | *Testing Objectives/ Support** | *Testing Strategies* |
|---|---|---|---|---|---|---|---|
| 1 | Appium [16] | WebDriver | IDE | iOS, Android | Any WebDriver compatible language | Regression, Exploratory | System |
| 2 | Docker Compose Rule [17] | HTTP Requests | Library | JVM | Java | Regression | Integration, System |
| 3 | Gatling [18] | HTTP Requests | Framework | JVM, SPP (Scala) | Java, Scala | Performance, Stress | Component, System |
| 4 | Gremlin [19] | HTTP Requests | Side-car + Proxy | Linux, Python Runtime Environment (PRE) | Linux Scripting Language, Python | End-to-End | Component, Integration, System |
| 5 | Hoverfly [4] | HTTP Requests | Side-car + Proxy | JVM, PRE, MacOS, Windows, Linux | Java, Python, Scripting Language | End-to-End | Component, Integration |
| 6 | JMeter [20] | HTTP Requests | IDE | JVM | Java | Functional, Regression | Component, Integration |
| 7 | JUnit [21] | Method call | Framework | JVM | Java | Functional, Regression | Unit |
| 8 | Minikube [22] | CLI | Container's Orchestrator | Linux, Windows, MacOS | Scripting Language | Test Harness* | Component, Integration |
| 9 | NUnit [23] | Method call | Framework | .Net Framework | .Net Languages | Functional, Regression | Unit |
| 10 | Pact [24] | HTTP Requests | Framework | Cross-section of runtime systems | Any supported language | Regression, Component | Contract |
| 11 | Selenium IDE [25] | HTTP Requests | Web browser extension | Chrome, Firefox | Selenium Script | Regression, Exploratory | System |
| 12 | Spring Boot Starter Test [26] | Method Call | Framework | JVM | Java | Test Harness* | Unit |
| 13 | Spring Cloud Contract [27] | HTTP Requests | Framework | JVM | Groovy, YAML | Regression, End-to-End | Contract |
| 14 | Telepresence [28] | Two-way proxy | CLI | Windows, Linux | Any | Test Harness* | Component |
| 15 | Tsung [29] | HTTP Requests | Framework | OTP (Erlang) | XML | Stress | Component, System |
| 16 | Watir [30] | WebDriver | CLI | Ruby VM | Ruby | Regression | System |

such as JUnit or NUnit, see Table I rows 7 and 8. Fowler [1] states that in the context of microservices the components are the services themselves, and should be tested as independent systems by mocking or stubbing the interaction with external dependent components. By removing these dependencies during testing, the service does not rely on the network communication which may modify the outcome of the test if the network is not behaving as expected. Tools to support component testing include Gatling [18] and Gremlin [19] (see Table I rows 3 and 4), among others.

When using microservices the interfaces to be defined in the contract are implemented communication protocols usually a public API or a messaging system. Sundar [31] states that contract testing is integral to microservices testing and identifies two types of contract testing, integration contract testing and consumer-driven contract testing. Using the producer-consumer service model, integration testing verifies the contract of producer's interface by creating mocks or stubs of the producer thereby replication the service to be consumed. Consumer-driven contract testing is done on the producer's side by verifying the different consumer contracts that need to be serviced. Tools to support contract testing include Pact [24] and Spring Cloud Contract [27], see Table I rows 9 and 12. Note that there is a difference between contract and component testing. The former doesn't test the behavior of the system but only its outputs against the inputs provided [1].

Fowler [1] states that when performing integration testing of system containing microservices, each service can be considered as a component and the services can be grouped together using the traditional approach to integration testing [13]. More specifically, a test or staging environment is used to integrate the individually tested microservices [31]. For example, using Docker Compose Rule [17] the tester can setup the environment with the services to be integrated and run test cases on this environment. Other tools that perform integration testing include Gremlin [19], Hoverfly [4], JMeter [20], see Table I rows 2, 4, 5, and 6.

System testing for microservices application is usually done by executing functionality through the UI or public API of the system. This type of testing includes end-to-end testing of user-define requirements and involves total coverage of the various microservices. Tools used for system testing include Appium [16], Docker Compose Rule [17], Gatling [18], Gremlin [19], and Selenium IDE [25], among others, see Table I. Sundar [31] states that end-to-end testing of a microservices architecture can be difficult and expensive to debug.

## IV. MICROSERVICES TESTBED

In this section we describe the polyglot testbed microservices system called *Rideshare* that implements a cluster of microservices for a ridesharing application that allows passengers to request a ride from a list of available drivers going from a pickup address to a destination address. The description of the application includes a high-level architecture of the system and a component diagram of the system. You can download this testbed from github repository (url omitted for review).

## *A. Architecture*

The requirements for the Rideshare application include use cases for two different actors, including a passenger and driver. The passenger requirements include: login, logout, registration, request a new trip, request an estimated cost of the trip, cancel a trip, request a list of available drivers, update the destination address, receives updates on the progress of the trip, and finish the trip. The driver requirements include: login, logout, registration, cancel trip and finish trip.

The Rideshare application uses the microservices architectural pattern consisting of a front-end, backing services and domain services. The high-level architecture of the Rideshare application is shown in Figure 3. The front-end is a web user interface provides user interfaces for both the passenger and the driver. These interfaces include screens for the requirements stated in the previous paragraph. The back-end services are the generic services to support running a microservices application and include the following:

- *Authentication* - validates user credentials.
- *Edge* - is a proxy that provides communication between the front-end and the domain services.
- *Discovery* - provides a directory and lookup functionality of the active microservices.
- *Configuration Services* - provides a centralized repository of files to support configuration of the domain services

The domain services are the service inherent to providing the functionality for the given application. These services include:

- *Notification*, is a stateless communication service that send asynchronous information expected by the front-end, such as ride updates, price estimates and payments.
- *Google Maps Adapter*, is a stateless application that forwards the request of information regarding the route for the ride, retrieving information about estimated time to reach the destination, directions and distance.
- *Trip Calculator*, takes the information retrieve by the Google Maps Adapter and calculate the price of the ride to be charged to the passenger user, saving the information temporarily on a data-store and sending it to the front-end through the Notification service.
- *Passenger*, keeps information of the passenger, e.g., name, phone, payment information and so on.
- *Driver*, keeps information on the driver, e.g., name, phone, vehicle, bank account and so on.
- *Billing*, bills customers for trips.
- *Payment*, request payment from the third party creditor.
- *Trip*, updates the status of the trip including location based on the passenger's and driver's locations.

Note that the Driver, Billing, Payment and Trip services shown in Figure 3 each consists of command and query components to show that the service is interacting differently with the database.

## *B. Implementation*

The Front-end of the Rideshare application is implemented using AngularJS [32]. The backing services were developed with Java Spring Boot framework [26]. The Edge service uses the Zuul library from Netflix [3] to perform the proxy communication as previously mentioned, in addition to the proxy tasks the Edge service also provides load balancing and fault tolerance. The Discovery service implements Eureka [3], which is a REST (Representational State Transfer) based service that provides support for locating other active services in the network. The Configuration service provides remote configuration capabilities to Spring Boot applications and finally the Authentication service is an OAuth2 implementation [33] that authorizes applications on behalf of an authenticated user.

The domain services are implemented as follows: Notification uses RabbitMQ [34] as a message broker that facilitates asynchronous communication between all the services. Google Maps Adapter, connects to the Google Maps Platform [35] to retrieve details of locations. The Trip Calculator is implemented in Golang [36] and uses a MongoDB database. The other services implement Command and Query Responsibility Segregation (CQRS), that is, every service is split into two smaller services one that takes care of the command executions using an event bus implemented with RabbitMQ, and another one for queries.

To illustrate the interactions between services we show a component diagram of the services involved in the *Request Trip Estimate* use case, as shown in Figure 4 highlighted by the lower rectangle labeled “System Under Test”. The initial request comes from the interaction of a passenger with the Front-End on the top left of the figure. The Front-End sends the request to Calculation through Edge API served to a port communication served by the container. The Edge service forwards the request to the Calculation API, then the Calculation service creates a message to be delivered to Google Maps Adapter through RabbitMQ.

When the message is received by Google Maps Adapter (shown at the bottom of the figure) it sends a request to the Google Maps Platform. If the Google Maps Adapter receives a successful response, the Calculation service uses this information to compute the trip estimate and stores it temporarily in MongoDB. The Notification service pushes the trip estimate to the Front-End. Since Google Maps Platform is a third party service, the server from Google may never respond, the Google Maps Adapter times out and sends an error message to the Calculation service. In which case the trip estimate is based on a standard price for the trip. The communication ends when the price is delivered to the Front-End.

The components shown in the top rectangle of Figure 4 labeled “Testing Framework” will be described in Section V when we discuss how the microservices are tested using several of the tools presented in Table I.

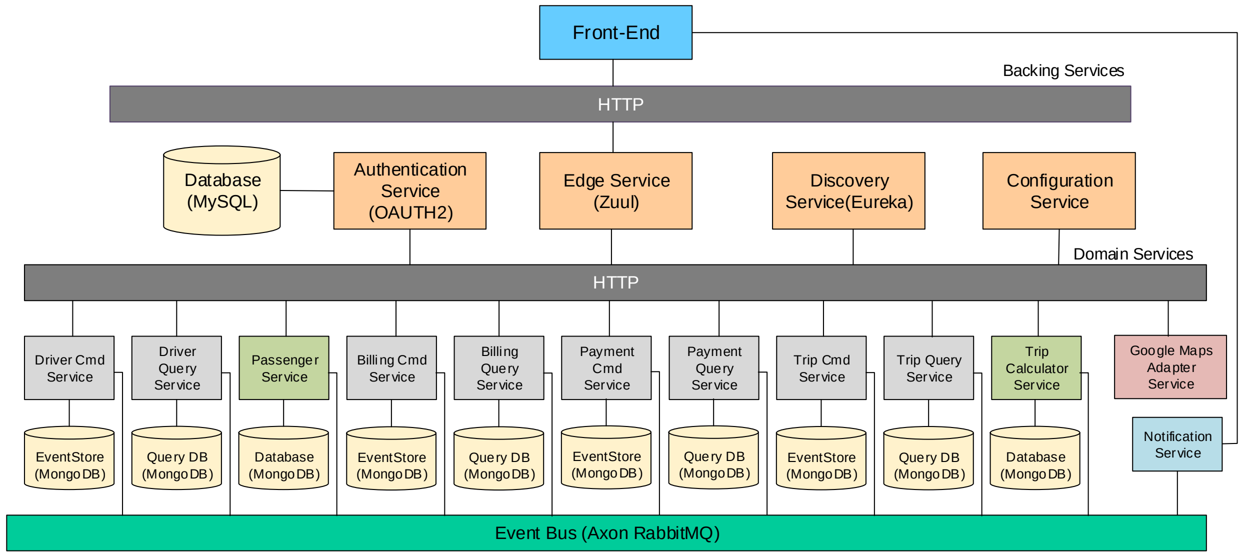

Fig. 3. Ridesharing testbed application.

## V. Case Study

In this section we present a case study that test the microservices for our Rideshare application presented in Section IV. The objective of the case study is to provide insights into the overhead required to setup and run the testing tools on the Rideshare application. In addition, we also provide static metrics on the tools and snippets of code to setup and test the application.

### A. Method

The tools used to test the Rideshare application are Gremlin [19], Hoverfly [4], Minikube [22] and Telepresence [28]. The tools are used to perform component and integration testing on a set of services based on several use cases. These use cases include: *create trip*, *request trip estimate*, *Gmaps adapter request*, *update destination*, and *request trip information*.

The platform used to execute the test cases on the Rideshare application are as follow. The hardware consisted of: Intel Core i7 2.2 Ghz, 32 GB of RAM and 500 GB of Hard drive space. The software installed on the platform include: Windows 10, Oracle VirtualBox version 5.2, Docker Tool Box, Gremlin version 0.1, Hoverfly version 0.17.7, Minikube version 0.32.0 and Kubectl 0.32.0.

To illustrate how the tools are used we show the setup and a sample test case using Gremlin. The setup required to run the test is shown at the top of Figure 4 and labeled *Testing Framework*. An instance of the testing framework may be as follows: GremlinSDK (Testing SDK), Fluentd (DataCollector) and ElasticSearch (Database).

To use Gremlin first run GremlinSDK with ElasticSearch (a full-text search and analytics engine) and specify three files that contain the topology, the fault injection instructions, and the expected output. The topology file *topology.json*, specifies a structured communication schema that reflects the services' communication dependencies, giving the names of each service and the services it depends on. For example, Listing 1 shows a list of services, e.g., `edge_service`, `calculation_service`, among others. Every service needs to implement a proxy that supports failure injection, precisely: abort, delay, and mangle. The Gremlin project provides a lightweight proxy written in Golang [36] that can be attached to each microservice in order to process HTTP requests.

Listing 1. Gremlin network topology example (*topology.json*).

```
{"services" : [
   {"name":"edge_service","service_proxies":
        ["127.0.0.1:9876"]},
   {"name":"calculation_service","service_proxies":
        ["127.0.0.1:9877"]},
   {"name":"gmaps_adapter"},
   {"name":"notification_service"}
  ],
 "dependencies" : {
   "edge_service":["calculation_service"],
   "calculation_service":["gmaps_adapter"],
   "gmaps_adapter":["notification_service"]
  }
}
```

The expected behavior in the service under test needs to be monitored and is specified in the file *checklist.json*, see Listing 2. The `log_server`, fist line of Listing 2, stores the activities specified in the `checks` list, e.g., the maximum latency allowed from the edge service to the calculation service is 100 milliseconds (`"max_latency" : "100ms"`). To define the list of fault injection actions applied to the inter-service communication, a scenario is define in a *gremlins.json* file [19]. The scenario shown in Listing 3 delays all HTTP requests 50 milliseconds from the edge service to the calculation service containing the header pattern `testUser-timeout-*`.

Running the *runrecipejson.py* loads the previous files shown in Listings 1, 2 and 3. The Gremlin SDK is now ready to filter the incoming HTTP requests with the header *X-Gremlin-ID*

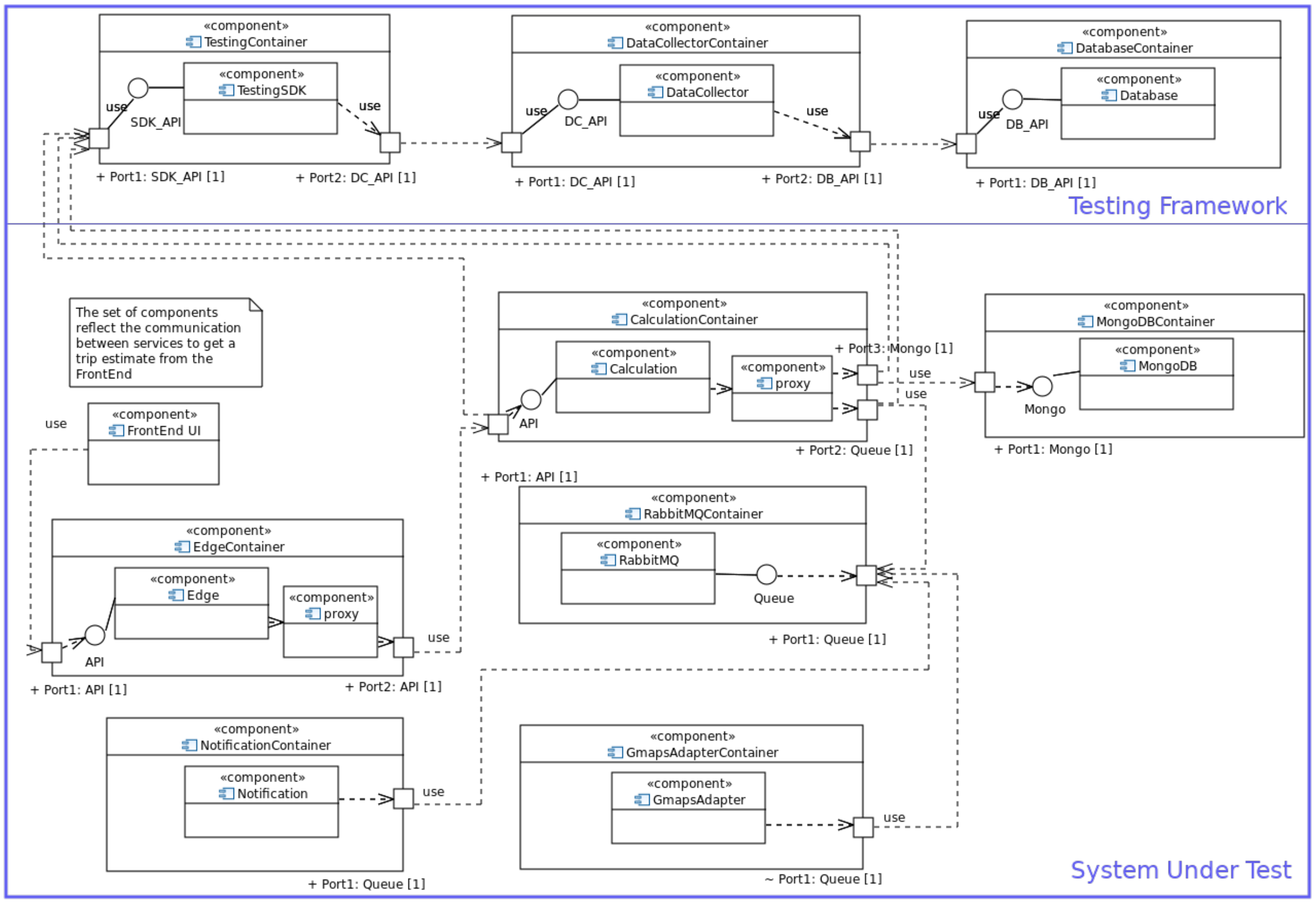

Fig. 4. Component diagram to request a trip estimate from the front-end showing how the testing SDK interacts with the system components. The proxies shown in the `EdgeContainer` and `CalculationContainer` are part of the testing framework.

with the value specified in the *gremlins.json* file in the field *headerpattern*. The system can now run test cases written with any HTTP client like Postman [37].

Listing 2. Gremlin checklist example (*checklist.json*).

```
{"log_server":"192.168.99.100:29200",
    "checks":[
      {"name" : "bounded_response_time",
        "source" : "edge_service",
        "dest" : "calculation_service",
        "max_latency" : "100ms"
      }
    ]
}
```

Listing 3. Gremlin recipe example (*gremlins.json*).

```
{"gremlins":[
    {"scenario":"delay_requests",
     "source":"edge_service",
     "dest":"calculation_service",
     "headerpattern":"testUser-timeout-*",
     "delaytime":"50ms"
    }
  ]
}
```

Unlike Gremlin the other tools in our study are relatively easy to setup and execute. Telepresence [28] is used to debug a service in an application by executing that service on a local machine while connecting the service to the production environment. For example, the `CalculationContainer`, shown in the middle of Figure 4, would be replaced by a empty container that routes all requests to the local machine that is executing the actual calculation service. By running the service on the local machine Telepresence supports the debugging of the service. In order for Telepresence to work Kubernetes is required to establish a connection to the production environment running the Rideshare application.

Minikube [22] encapsulates services inside pods, thereby providing the services with some level of resiliency by monitoring the health of each service. That is, Minikube continuously send a message to determine if the service is running. For example, the system under test shown in Figure 4 is deployed in Minikube by encapsulated each service to be tested in a pod, and exposing external ports to allow communication between the services. Assuming the `Calculation` component in the `CalculationContainer` is to be tested, it is encapsulated within a pod and the pod ports forwards

TABLE II
STATIC METRICS FOR THE APPLICATION UNDER TEST AND THE TESTING TOOLS.

| | LOC | No. Files | Bugs | Vulnerab. | Code Smells | Dup. (%) |
|---|---|---|---|---|---|---|
| *System under test:* | | | | | | |
| Rideshare | 8,129 | 394 | 28 | 16 | 892 | 9.9 |
| *Supporting Platform:* | | | | | | |
| Docker CE | 263,984 | 10,091 | 1 | 676 | 4,555 | 6.0 |
| *Testing tools:* | | | | | | |
| Gremlin | 1,538 | 11 | 6 | 4 | 70 | 34.9 |
| Hoverfly | 39,332 | 1,337 | 0 | 9 | 1,354 | 21.4 |
| Minikube | 25,948 | 2,150 | 5 | 253 | 283 | 5.2 |
| Telepresence | 10,699 | 30 | 21 | 55 | 38 | 8.8 |

TABLE III
EXECUTION TIMES TO RUN 5 SYSTEM TEST CASES EACH ITERATED 5 TIMES.

| Tool | Total Time (sec.) | Mean (sec.) | Std Dev. | Max. (sec.) | Min. (sec.) |
|---|---|---|---|---|---|
| Docker CE | 5.30 | 0.11 | 0.23 | 0.88 | <0.01 |
| Gremlin | 6.36 | 0.13 | 0.28 | 1.19 | <0.01 |
| Hoverfly | 5.57 | 0.11 | 0.27 | 1.15 | <0.01 |
| Minikube | 14.85 | 0.30 | 0.43 | 2.17 | <0.01 |
| Telepresence | 1,003.22 | 20.06 | 40.92 | 123.13 | 0.05 |

communication to the container's ports (`Port1` shown to the left of the `CalculationContainer`). After launching the pod Minikube monitors the health of the `Calculation` component by sending requests to the container's port. If there is no answer then Minikube destroys the container and creates a new one based on the configuration file.

Hoverfly [4] may be used in several different modes including capture, simulate, spy, synthesize modify and diff. In this work we focus on on capture and simulation modes. In capture mode Hoverfly is used to record data contained in the HTTP requests between services, including the time to complete a transaction. This data is can be used to simulate the communication between services during testing. For example, first setup Hoveryfly as a proxy in capture mode, then the scenarios are executed thorough the Hoverfly proxy. Note that Hoverfly would act as a proxy prior to reaching the `EdgeContainer`, thereby logging all the data communication going to/from the `EdgeContainer`. This captured data can then be used to simulate the behavior of Rideshare application during system testing.

## B. Results

The data collected during the study includes both static and dynmaic metrics for the Rideshare application, supporting platform, and the tools executing the test cases. The static metrics for the application, platform and the tools were collected using SonarQube [38] which is a tool for continuous inspection to show the health of an application. The metrics provided by SonarQube are: *LOC* - Lines of Code; *Bugs* - are either demonstrably wrong code, or code that is more likely not giving the intended behavior; *Vulnerabilities* - security concerns, e.g., SQL injections; *Code Smells* - the code does (probably) what it should, but it will be difficult to maintain; and *Duplication* - counts replicated pieces of code.

Table II shows the static metrics for the Ridershare application, the supporting platform, and the tools used while testing the application. The table is divided into 3 sections, metrics for the Rideshare application, metrics for Docker CE [39], and metrics for the tools. The Rideshare application is shown in row 1 and has 8,129 lines of code, 394 files, 28 bugs (as defined by SonarQube [38]), 16 vulnerabilities, 892 code smells, and 9.9% of the code is duplicated. Docker CE [39] has 263,984 lines of code, 10,091 files, 1 bug, 676 vulnerabilities, 4,555 code smells, and 6.0% of the code is duplicated. Hoverfly [4] is the largest tool with 39,332 lines of code, 1,337 files, 0 bugs, 9 vulnerabilities, 1,354 code smells, and 21.4% of the code is duplicated.

Table III shows the execution times to run 5 system test cases each for 10 iterations. We decided to run each test case 10 times to get a better estimate of the execution times. The columns in the table from left to right are the total time, the arithmetic mean, standard deviation, maximum time and the minimum time for all the iterations. Since the Rideshare application runs on Docker the execution times in the first row of the table can be considered the baseline for running the Rideshare application. Telepresence takes the longest time to run all iterations of the system test cases with a total of 1,003.22 seconds. The percentage increase of execution time of Telepresence over Docker CE is 18,842% while Hoverfly has the least percentage increase over Docker CE, 5.1%. Gremlin has an increase of 20.11% and Minikube 180.35% over Docker CE.

One aspect of the data in Table III that stands out is the relatively high standard deviation with respect to the execution times. This is supported by the difference between the maximum and minimum execution times for the different system test cases. The greatest difference between the maximum and minimum execution times is for Telepresence [28].

## C. Discussion

The data shown in Tables II and III provides some insight into the variability of the tools available to test microservices. The tools used in the study are a small sample of the list shown in Table I, and represent test harness tools (Minikube and Telepresence) and end-to-end testing tools (Gremlin and Hoverfly). Based on the static metrics reported by SonarQube [38], it appears some of the testing tools can themselves be improved. For example, Hoverfly has approximately 3.4% of its code base that might be difficult to maintain compared to Minikube which only has 1.1%. On the other hand Minikube has approximately 1.0% vulnerabilities in its code and Hoverfly has 0.02%.

Some of the executions times shown in Table III were unexpected, particularity the high executions times observed for Minikube and Telepresence both of which are test harnesses. It is inferred that these high executions times are related to the communication setup with the services to be tested. This thesis is supported by the fact that Hoverfly has execution times close to Docker CE, and requires manual changes to the system test cases in order to use the proxy service provided by Hoverfly.

One of the most challenging aspects to conducting the experiments using the tools in Table II is setting up the tools to run with the prototype. The tool configuration files need to be written manually and match the environments in the containers of the services. Other challenges include: the resources needed to run both the tools and the prototype ,e.g., RAM; unreliable network connectivity; and other supporting entities needed to run the tools for different operating systems. The unreliable network connectivity refers to the communication ports that are needed for the docker containers on the host machine that can sometimes be reassigned to other applications.

## VI. Related Work

In this section we present the work related to tools used to test microservices. Heorhiadi et al. [19] describes Gremlin as a framework for systematically testing the failure-handling capabilities of microservices. It is considered to be a general purpose tool that performs test on a fully deployed system or set of components. Additionally, it requires a HTTP header filter to work, this is usually achieved by a proxy attached to every service to redirect the HTTP messages to the Gremlin SDK. The work by Heorhiadi et al. motivated the contents of this paper resulting in the comparison of a cross-section of microservices testing tools similar to Gremlin.

Sundar [31] describes different test scenarios and testing strategies for microservices. These scenarios and strategies include: testing between microservices internal to an application; testing between internal microservices and a third-party service; and testing microservices to be exposed to the public domain. The descriptions of the scenarios and test strategies are presented at a high level, and for each strategy the authors describes test at various levels including unit testing, contract testing, integration contract testing, integration testing and end-to-end testing.

Savchenko et al. [40] describes an approach to testing the Mjolnirr microservice platform, which is prototype developed by the authors. The work starts by describing the microservices architectural pattern and comparing it to a monolithic system architecture. This introduction lays the foundation for the different feature types and testing levels that may be applied to a microservices architecture. Our work also initially compares microservices and monolithic architectures, however we then provide a comparison of the tolls that may be used to test microservices which is not done by Savchenko et al. [40].

Unlike the previous works described in this section, Panda et al. [41] argue that the recent advances in formal methods can pave the way to build mechanisms to ensure the correctness of systems that use microservices. The authors propose a system *ucheck* that enforces invariants in microservices based applications. The ucheck system can be used to define invariants on the remote procedure calls (RPC) thereby indicating erroneous behavior that may occur in a microservices based system, e.g., insertion calls to the database before a corresponding authentication call. ucheck requires no coordination which results in minimal overhead to the running system. This work shows that researchers are seeking out solutions to improve the correctness of microservices applications using formal techniques.

## VII. Conclusion

The increasing use of microservices in many software applications has resulted in the need for testing tools and techniques more suited to testing these applications. To address this problem, new testing techniques and tools are being developed that focus on the added complexity of increased data communication required between several services. In this paper we compared several tools that can be used to test microservices at different levels including end-to-end testing and regression testing, among others. Some of these tools also provide infrastructure support for executing test cases, e.g., test harness.

We also described an open-source polyglot Ridesharing microservices reference prototype that can be used as a testbed to evaluate new testing techniques and tools. Using the Ridesharing application test cases were executed using four different tools and the execution times were reported. Our future work include developing a testing framework that can run system tests with minimal communication overhead. We also plan to extend the study presented in this paper to include additional tools and perform a more in-depth performance analysis using the Ridesharing prototype.

## References

[1] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," http://web.archive.org/web/20180404230511/https://martinfowler.com/articles/microservices.html, accessed: 2018-04-05.

[2] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka, "The morabit approach to runtime component testing," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 2, Sept 2006, pp. 171–176.

[3] Netflix Development Team, "Netflix," https://www.netflix.com/, 2019, [Online; accessed 18-Jan-2019].

[4] SpectoLabs, "Hoverfly," https://hoverfly.io/, 2019, [Online; accessed 18-Jan-2019].

[5] D. Inc., "Ambassador," https://www.getambassador.io/, 2019, [Online; accessed 10-Feb-2019].

[6] Lyft Development Team, "Envoy," https://www.envoyproxy.io/, 2019, [Online; accessed 18-Jan-2019].

[7] Google, "Kubernetes," https://kubernetes.io/, 2019, [Online; accessed 10-Jan-2019].

[8] ——, "Red Hat Developer Community," https://openshift.io/, 2019, [Online; accessed 10-Jan-2019].

[9] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 269–274, Apr. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=205313.205314

[10] Z. Dehghani, "How to break a monolith into microservices," https://martinfowler.com/articles/break-monolith-into-microservices.html, April 2018, [Online; accessed 21-Jan-2019].

[11] M. Richards, *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[12] F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley Professional, 2015.

[13] IEEE Computer Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[14] A. P. Mathur, *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[15] C. Atkinson and H.-G. Groß, "Built-in contract testing in model-driven, component-based development," in *ICSR Workshop on Component-Based Development Processes*, 2002.

[16] J. Foundation, "Appium," http://appium.io/, 2019, [Online; accessed 25-Jan-2019].

[17] P. Technologies, "Docker Compose Rule Library," https://github.com/palantir/docker-compose-rule, 2018, [Online; accessed 15-Oct-2018].

[18] G. Corp, "Appium," https://gatling.io/, 2019, [Online; accessed 25-Jan-2019].

[19] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *Proceedings - 2016 IEEE 36th International Conference on Distributed Computing Systems, ICDCS 2016*, vol. 2016-August. Institute of Electrical and Electronics Engineers Inc., 8 2016, pp. 57–66.

[20] Apache Software Foundation, "Appium," http://jmeter.apache.org/, 2019, [Online; accessed 25-Jan-2019].

[21] The JUnit Team, "JUnit," https://junit.org/, 2019, [Online; accessed 25-Jan-2019].

[22] Google, "Minikube," https://github.com/kubernetes/minikube, 2019, [Online; accessed 10-Jan-2019].

[23] The NUnit Team, "NUnit," http://nunit.org/, 2019, [Online; accessed 25-Jan-2019].

[24] Pact Foundation, "Pact," https://docs.pact.io/, 2019, [Online; accessed 25-Jan-2019].

[25] Software Freedom Conservancy (SFC), "Selenium IDE," https://www.seleniumhq.org/selenium-ide/, 2019, [Online; accessed 25-Jan-2019].

[26] Pivotal Software, Inc, "Spring Boot," https://spring.io/projects/spring-boot, 2019, [Online; accessed 25-Jan-2019].

[27] ——, "Spring Cloud Contract," http://spring.io/projects/spring-cloud-contract, 2019, [Online; accessed 25-Jan-2019].

[28] D. Inc., "Telepresence," https://www.telepresence.io, 2019, [Online; accessed 10-Jan-2019].

[29] Nicolas Niclausse, "Tsung," http://tsung.erlang-projects.org/, 2019, [Online; accessed 25-Jan-2019].

[30] Watir Development Team, "Watir," http://watir.com/, 2019, [Online; accessed 25-Jan-2019].

[31] A. Sundar, "An insight into microservices testing strategies," https://www.infosys.com/it-services/validation-solutions/white-papers/documents/microservices-testing-strategies.pdf, [Online; accessed January 2019].

[32] Google, "Angualr Js," https://angularjs.org/, 2019, [Online; accessed 10-Jan-2019].

[33] O. Community, "OAuth 2.0," https://oauth.net/2/, 2019, [Online; accessed 10-Jan-2019].

[34] Pivotal Software, Inc, "RabbitMQ," https://www.rabbitmq.com/, 2019, [Online; accessed 26-Jan-2018].

[35] Google, "Google Maps Platform," https://cloud.google.com/maps-platform/, 2019, [Online; accessed 10-Jan-2019].

[36] Google, "The Go Programming Language," https://golang.org/, 2019, [Online; accessed 10-Jan-2019].

[37] Postman Inc., "Postman," https://www.getpostman.com/, 2019, [Online; accessed 10-Feb-2019].

[38] Google, "SonarQube," https://docs.sonarqube.org/, 2019, [Online; accessed 18-Jan-2019].

[39] D. D. Team, "Docker Community Edition," https://docs.docker.com/, 2018, [Online; accessed 12-Feb-2019].

[40] D. I. Savchenko, G. I. Radchenko, and O. Taipale, "Microservices validation: Mjolnirr platform case study," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 235–240.

[41] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, pp. 30–36. [Online]. Available: http://doi.acm.org/10.1145/3102980.3102986