2016 IEEE 36th International Conference on Distributed Computing Systems

Gremlin: Systematic Resilience Testing of Microservices

Victor Heorhiadi
UNC Chapel Hill
victor@cs.unc.edu

Shriram Rajagopalan
IBM T. J. Watson Research
shriram@us.ibm.com

Hani Jamjoom
IBM T. J. Watson Research
jamjoom@us.ibm.com

Michael K. Reiter
UNC Chapel Hill
reiter@cs.unc.edu

Vyas Sekar
Carnegie Mellon University
vsekar@andrew.cmu.edu

***Abstract*—Modern Internet applications are being disaggregated into a microservice-based architecture, with services being updated and deployed hundreds of times a day. The accelerated software life cycle and heterogeneity of language runtimes in a single application necessitates a new approach for testing the resiliency of these applications in production infrastructures. We present Gremlin, a framework for systematically testing the failure-handling capabilities of microservices. Gremlin is based on the observation that microservices are loosely coupled and thus rely on standard message-exchange patterns over the network. Gremlin allows the operator to easily design tests and executes them by manipulating inter-service messages at the network layer. We show how to use Gremlin to express common failure scenarios and how developers of an enterprise application were able to discover previously unknown bugs in their failure-handling code without modifying the application.**

1. Introduction

Many modern Internet applications are moving toward a *microservice* architecture [15], to support rapid change and respond to user feedback in a matter of hours. In this architecture, the application is a collection of web services, each serving a single purpose, i.e., a microservice. Each microservice is developed, deployed and managed independently; new features and updates are delivered continuously [10], hundreds of times a day [20]–[22], making the applications extremely dynamic. Microservice applications are typically *polyglot*: developers write individual microservices in the programming language of their choice, and the communication between services happens using remote API calls.

As cloud-native applications, microservices are designed to withstand infrastructure failures and outages, yet struggle to remain available when deployed. Table 1 summarizes some recent failures experienced by microservice-based applications. The postmortem reports point to missing or faulty failure-recovery logic, indicating that unit and integration tests are insufficient to catch such bugs. The application deployment needs to be subjected to *resiliency testing*—testing the application's ability to recover from failures commonly encountered in the cloud. Specifically, we argue that resiliency testing needs to be *systematic* and feedback-driven: allow operators (developers or testers) to orchestrate a specific failure scenario and obtain quick feedback about how and why the application failed to recover as expected. This feedback makes systematic testing more valuable than randomized fault injection by better enabling developers to quickly locate and fix faulty failure-handling logic, redeploy, and test again.

Company	Downtime	Postmortem findings
Parse.ly, 2015 [25]	13 hours	Cascading failure due to message bus overload
CircleCI, 2015 [19]	17 hours	Cascading failure due to database overload
BBC, 2014 [18]	48 hours	Cascading failure due to database overload
Spotify, 2013 [26]	Several hours	Cascading failure due to degradation of a core internal service
Twilio, 2013 [28]	10 hours	Database failure caused billing service to repeatedly bill customers

TABLE 1: A subset of recent outages experienced by popular, highly available Internet services. Postmortem reports revealed missing or faulty failure-handling logic.

Microservices and their development model, however, pose new challenges for systematic resiliency testing:

- Microservices' polyglot nature requires the testing tool to be agnostic to each service's language and runtime.
- Rapidly evolving code requires tests that are fast and focus on the failure-recovery logic, not business logic.

Existing works on resilience testing of general distributed systems and service-oriented architectures (SOAs) are unsuitable for microservice applications, since they do not address the challenges mentioned above (see Section 8 for details). In contrast, our work is designed to provide systematic, application-agnostic testing on live services.

To achieve this goal, we make the following observations about microservice applications:

- Interactions between microservices happen solely over the network; and
- Microservices use standard application protocols (e.g., HTTP) and communication patterns (e.g., request-response, publish-subscribe).

Our key insight is that failures can be staged by manipulating the network interactions between microservices; the application's ability (or lack of thereof) to recover from failures can be evaluated by observing the network interactions between microservices during the failure.

Based on this insight, we present *Gremlin*, a systematic resiliency testing framework for microservices. Gremlin's design is inspired by software-defined networks (SDN): the operator interacts with a centralized control plane, which in turn configures the data plane. The operator provides Gremlin with a *recipe*—Python-based code describing a high-level outage scenario, along with a set of assertions on how microservices should react during such an outage. The control plane translates the recipe into a fixed set of fault-

1063-6927/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDCS.2016.11

injection rules to be applied to the network messages exchanged between microservices. Gremlin's data plane consists of network proxies that intercept, log, and manipulate messages exchanged between microservices. The control plane configures the network proxies for fault injection based on the rules generated for a recipe. After emulating the failure, the control plane analyzes the observation logs from the network proxies to validate the assertions specified in the recipe. Gremlin recipes can be executed and checked in a matter of seconds, thereby providing quick feedback to the operator. This low-latency feedback enables the operator to create correlated failure scenarios by conditionally chaining different types of failures and assertion checks.

Our case study shows that Gremlin requires a minimal learning curve: developers at IBM found unhandled corner-case failure scenarios in a production enterprise application, without modifying the application code. Furthermore, Gremlin correctly identified the lack of failure handling in an actively used library designed specifically for abstracting away failure-handling code. Controlled experiments indicate that Gremlin is fast, introduces low overhead, and is suitable for resiliency testing of operational microservice applications.

Fault-injection ideas from Gremlin and their software-defined architecture have been integrated into IBM's cloud offerings for microservice applications. Integration of other parts, such as validation of assertions, is planned for the future. The source code for Gremlin is publicly available in GitHub at `https://github.com/ResilienceTesting`.

In summary, our contributions are as follows:

- A systematic resiliency-testing framework for creating and executing recipes that capture a rich variety of high-level failure scenarios and assertions in microservice-based applications.
- A framework that can be integrated easily into production or production-like environments (e.g., shadow deployments) without modifications to application code.
- Recipes capable of tolerating the rapid evolution of microservice code by taking advantage of the standardized interaction patterns between services.

2. Background and Motivation

Large-scale Internet applications such as Netflix, Facebook, Amazon store, etc., have demonstrated that in order to achieve scalability, robustness and agility, it is beneficial to split a monolithic web application into a collection of fine-grained web services, called microservices [15]. Figure 1 illustrates the architecture of a typical microservice-based web application deployed in a cloud platform, such as Amazon AWS, IBM Bluemix, Microsoft Azure, etc. Each microservice is a simple REST [6] based web service that interacts with other services using HTTP. Modern applications leverage both managed services offered by the hosting cloud platform (e.g., relational databases, key-value stores) and third party services (e.g., Facebook, Twitter).

When compared to traditional service-oriented architectures, microservices are very loosely coupled with one another—they can be updated and deployed independently

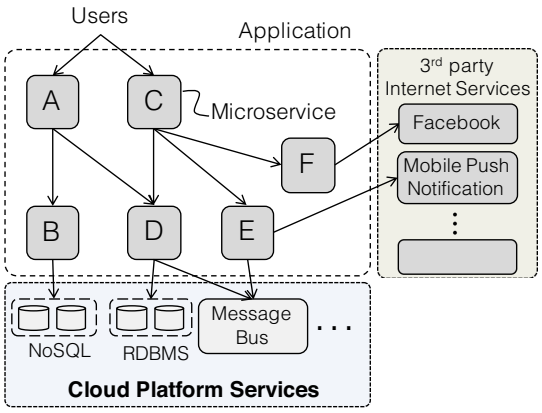

Figure 1: Typical architecture of a microservice-based application. The application leverages services provided by the hosting cloud platform, (e.g., managed databases, message queues, data analytics), and integrates with Internet services, such as social networking, mobile backends, geolocation, etc.

of other microservices as long as the APIs they expose are backward compatible. To achieve loose coupling, microservices use standard application protocols such as HTTP to facilitate easy integration with other microservices. Organizationally, each microservice is owned and operated by an independent team of developers. The ability to immediately integrate updates into the production deployment [5] has led to a continuous software delivery model [10], where development teams incrementally deliver features while incorporating user feedback.

2.1. Designing for Failure: Current Best Practices

To remain available in the face of infrastructure outages in the cloud, *a microservice must guard itself from failures of its dependencies*. Best design practices advocate incorporating resiliency design patterns such as timeouts, bounded retries, circuit breakers, and bulkheads [9], [16].

- *Timeouts* ensure that an API call to a microservice completes in bounded time, to maintain responsiveness and release resources associated with the API call in a timely fashion.
- *Bounded retries* handle transient failures in the system, by retrying the API calls with the expectation that the fault is temporary. The API calls are retried a bounded number of times and are usually accompanied with an exponential backoff strategy to avoid overloading the callee microservice.
- *Circuit breakers* prevent failures from cascading across the microservice chain. When repeated calls to a microservice fail, the circuit breaker transitions to *open* mode and the caller service returns a cached (or default) response to its upstream microservice. After a fixed time period, the caller attempts to re-establish connectivity with the failed downstream service. If successful, the circuit is closed again, resuming normal operation. The definition of success is implementation dependent (e.g., response times within a threshold, absence of errors in a time period, etc.)

- *Bulkheads* provide fault isolation within a microservice. If a shared thread pool is used to make API calls to multiple microservices, thread pool resources can be quickly exhausted when one of the downstream services degrades. Resource exhaustion renders the service incapable of processing new requests. The bulkhead pattern mitigates this issue by assigning an independent thread pool for each type of dependent microservice being called.

Our study of the postmortem reports of recent outages (recall Table 1) shows that while developers may have implemented failure-recovery measures in their application logic, they remain unaware whether their microservice can tolerate failures, until the failure actually occurs. To our knowledge, there are no tools that provide systematic feedback indicating whether failure-recovery measures work as expected in a microservice application.

2.2. Challenges in Resiliency Testing of Microservices

A microservice-based application is fundamentally a distributed application. However, it differs from distributed file systems, databases, co-ordination services, etc. The latter group of applications have complex distributed state machines with a large number of possible state transitions. Existing tools for resiliency testing cater to the needs of these traditional low-level distributed applications [4], [7], [8], [11]. We find these tools to be unsuitable for use in web/mobile focused microservice applications, due to the following challenges:

C1. **Runtime heterogeneity.** An application may be composed of microservices written in different programming languages. Microservices may also be rewritten at any time using a different programming language, as long as they expose the same set of APIs to other services. Consequently, approaches that rely on language specific capabilities (e.g., dynamic code injection in Java) for fault injection and verification [7], [11] are infeasible in such heterogeneous environments, since few runtimes provide these capabilities.

C2. **High code churn.** Microservices are autonomously managed by independent teams. New versions of microservices are deployed 10-100 times a day, independently of other services. Exhaustive checkers [13] cannot keep up with this time scale.

C3. **Automatic validation.** The key to the agility of a microservice-based architecture is automation. Resiliency-testing tools such as Netflix's Chaos Monkey [34] inject unpredictable faults automatically. However, manual validation that the microservices survived the failure as expected is still required. When services fail to recover, it could result in lengthy troubleshooting sessions.

A useful resiliency-testing tool must be automated, systematic, and agnostic to the application's runtime platform. Furthermore, it is crucial that both the fault injection and behavior validation are automated.

3. Gremlin Overview

To tackle the challenges described earlier, we propose a systematic resiliency-testing framework called Gremlin. The key observations behind Gremlin's approach are as follows:

O1. **Touch the network, not the app.** Irrespective of the runtime heterogeneity (C1), all communication in the application happens entirely over the network. Multiple microservices work in coalition to generate the response to an end user's request. There are two important ramifications of this increased reliance on the network. First, *common types of failures can be easily emulated by manipulating the network interactions*. For example, appearance of an overloaded service can be created by delaying requests between two microservices. Second, *the failure recovery of a microservice can be observed from the network*. For example, by observing the network interactions, we can infer whether a microservice handles transient network outages by retrying its API calls to the destination microservice. We leverage this network observability property to automatically validate (C3) the recovery behavior of collection of microservices.

O2. **Volatile code with standard interactions.** Despite the rapid rate at which the microservice application evolves in a daily fashion (C2), the interaction between different microservices can be characterized using a few simple, standard patterns such as request-response, publish-subscribe, etc. The semantics of these application-layer transport protocols and the interaction patterns are well understood. Therefore, it is possible to elicit a failure-related reaction from any microservice, irrespective of its application logic or runtime, by manipulating these interactions directly. For example, an overloaded server can be emulated by intercepting the client's HTTP request and responding to it with the HTTP status code `503 Service unavailable`.

Gremlin's design leverages these observations to provide a resiliency-testing tool that is purely network-oriented and agnostic to the application code and runtime.

3.1. Fault Model

In a microservice-based application, response to a user request is a composition of responses from different microservices that communicate over the network. We confine our fault model to failures that are observable from the network by other microservices. Gremlin supports emulation of *fail-stop/crash failures, performance/omission failures, and crash-recovery failures* [1]—the most common types of failures encountered in modern-day cloud deployments. We do not formally prove coverage of these failure types. We also do not test the resilience of the application against malicious attacks.

From the perspective of a microservice making an API call, failures in a remote microservice or the network manifests in the form of delayed responses, error responses (e.g., HTTP 404, HTTP 503), invalid responses, connection timeouts and failure to establish the connection. The failure

incidents described in Table 1 (in Section 1) can be emulated by the failure modes currently supported by Gremlin, even though our system does not cover emulating OS-level errors, such as failed system calls.

3.2. Using Gremlin

Before we delve into the design of Gremlin, we provide the reader with a sample of Gremlin's capabilities. In Gremlin, the human operator (e.g., developer or tester) writes a Gremlin *recipe*: a test description, written in Python, which consists of the outage scenario to be created and assertions to be checked. Assertions specify expected behavior of microservices during the outage. An operator can orchestrate elaborate failure scenarios and validate complex application behaviors in short and easy-to-understand recipes.

Consider a simple application consisting of two HTTP-based microservices, namely ServiceA and ServiceB, where ServiceA makes API calls to ServiceB. An operator might wish to test the resiliency of ServiceA against any degradation of ServiceB, with the expectation that ServiceA would retry failed API calls no more than five times. With Gremlin, she can conduct this resiliency test using the following recipe (boilerplate code omitted for brevity):

Example 1: Overload test

```
1 Overload(ServiceB)
2 HasBoundedRetries(ServiceA, ServiceB, 5)
```

In line 1, Gremlin emulates the overloaded state of ServiceB, without actually impacting ServiceB. When traffic is injected into the application, ServiceA would experience delayed responses from ServiceB, or receive an HTTP error code (`503 Service unavailable`). The operator's expectation (when ServiceA encounters such behavior, it should restrict the number of retries to five attempts) is expressed using the assertion in line 2.

4. Design

A high-level view of Gremlin architecture is shown in Figure 2. Gremlin takes an approach similar to a software-defined network (SDN) for emulating failures. Broadly, the framework is divided into *data plane* and a *control plane*. We describe each layer in greater detail below.

4.1. The Data Plane

The data plane consists of network proxies, called *Gremlin agents*. Microservices are configured to communicate with each other via these agents. In addition to proxying the API calls, Gremlin agents can manipulate the arguments, return values, and timing of the calls, thus acting as fault injectors. As shown in Table 2, the data plane supports three primitive fault injection actions: `Abort`, `Delay`, and `Modify`. Using these primitives, we can construct complex failure scenarios that emulate real-world outages. Like SDN switches, Gremlin agents expose a well-defined interface to the control plane. The control plane uses the interface to send *rules* to the agents, instructing them to inspect the messages and perform fault-injection actions if a message matches a given criteria.

Identifying message boundaries. The question of how to delineate message boundaries arises, given that the Gremlin agents are agnostic to the application. We leverage earlier observation (O2) of common application-layer protocols, which can be easily decoded by the Gremlin agent. The semantics of fault-injection primitives (`Abort`, `Delay`, and `Modify`) also depend on the application protocol being used. For example, with HTTP, the `Abort` primitive would return HTTP error codes such as `503 Service unavailable`. Other protocols can also be supported, given that the implementation is augmented accordingly.

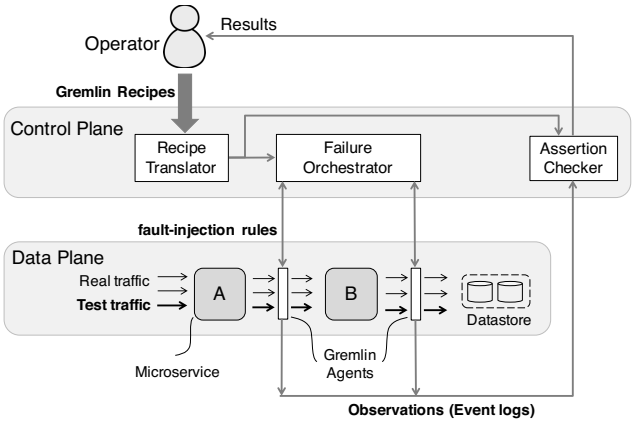

Figure 2: High-level overview of the Gremlin framework. The Recipe Translator breaks down high-level failure scenarios into fault-injection rules and assertions using the logical application graph. The Failure Orchestrator programs the Gremlin agents in the physical deployment to inject faults on matching request flows. The Assertion Checker executes assertions in the translated recipe against event logs provided by Gremlin agents.

Interface	Mandatory Parameters	Description
`Abort`	`Src`, `Dst`, `Error`, `Pattern`	Abort messages from `Src` to `Dst`, where messages match pattern `Pattern`. Return an application-level `Error` code to `Src`
`Delay`	`Src`, `Dst`, `Interval`, `Pattern`	Delay forwarding of messages from `Src` to `Dst`, that match pattern `Pattern`, by specified `Interval`
`Modify`	`Src`, `Dst`, `ReplaceBytes`, `Pattern`	Rewrite messages from `Src` to `Dst`, that match pattern `Pattern` and replace matched bytes with `ReplaceBytes`

TABLE 2: Interface exposed by the data plane (Gremlin agents) to the control plane. Messages in this context are application layer payloads (Layer 7), without TCP/IP headers. Non-mandatory parameters (with default values) not shown.

Injecting faults on specific request flows. To aid monitoring and troubleshooting, a common practice in microservice applications is to generate a globally unique request ID per user request and propagate that ID to downstream services by embedding it in the message headers. The flow of a user's request across different microservices can be traced using this unique request ID [3], [17], [29]. Gremlin agents can limit fault injection and logging to specific flows by matching request IDs.

Logging observations. During a test, Gremlin agents log the API calls made by the microservices and report them to the control plane. Each Gremlin agent records the following

information about an API call:

- Message timestamp and request ID
- Parts of the message (e.g, status codes, request URI)
- Fault actions applied to the message, if any

This information is used by the control plane to check the assertions in a Gremlin recipe.

4.2. The Control Plane

The control plane has three components: a Recipe Translator, a Failure Orchestrator and an Assertion Checker. The individual components are described in greater detail below.

Recipe Translator. The Recipe Translator exposes a Python interface to the operator, which enables her to compose high-level failure scenarios and assertions from pre-existing recipes or directly from low-level primitives for fault injection (shown in Table 2) and assertions (shown in Table 3). The operator is also expected to provide a logical application graph: a directed graph describing the caller/callee relationship between different microservices. Internally, the translator breaks down the recipe into a set of fault-injection rules to be executed on the application's logical graph—a dependency graph between various microservices. Recall Example 1 from Section 3.2 and its `Overload` failure. In this case, `Overload` is internally decomposed into `Abort` and `Delay` actions, parameterized and passed to the Failure Orchestrator.

Failure Orchestrator. The Failure Orchestrator sends fault-injection actions to the Gremlin data plane agents through an out-of-band control channel. Since an application might have multiple instances of any given service, the Failure Orchestrator locates and configures all physical instances of the Gremlin agents. In our example, the physical deployment is shown in Figure 3, with both ServiceA and ServiceB having two instances. When applying the fault-injection rules, the Failure Orchestrator affects communication between *every pair* of instances of ServiceA and ServiceB, by configuring Gremlin agents located at `10.1.1.1` and `10.1.1.2`.

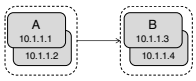

Figure 3: Multiple instances

Assertion Checker. This component is responsible for validating the assertions provided in the recipe. It does so by querying a centralized data store that contains event logs collected from the data plane (i.e., Gremlin agents) and performing a variety of processing steps. To aid the operator in querying the event logs, Gremlin provides abstractions (listed in Table 3) for fetching and analyzing the data. The queries (such as `GetRequests` and `GetReplies`) return a filtered list of observations from the Gremlin agents, sorted by time, referred to as `RList`. `NumRequests` and `ReplyLatency` operate on that list to compute basic statistics on the requests (or replies), without requiring knowledge of the log-record structure.

Using the queries, we develop a layer of *base assertions* that allows building of more complex checks, and also act as convenience methods. Unlike queries, base assertions return a boolean value so they can be chained. For example:

```
def AtMostRequests(RList, Tdelta, withRule, Num):
  return NumRequests(RList, Tdelta, withRule) <= Num
```

Here, `AtMostRequests` can subsequently be used in combination with other base assertions (as we show below).

Base assertions have two important features. First, they allow us to combine observations of service behavior with Gremlin's fault injection during analysis. Consider our earlier example with ServiceA and ServiceB. Gremlin was emulating an `Overload` of ServiceB. Setting the `withRule` parameter to `True` causes the `ReplyLatency` query to calculate *delayed* reply timings from ServiceB, as ServiceA would have observed them (due to Gremlin agents' actions). This is necessary to construct accurate preconditions when validating behavior of ServiceA. On the other hand, if the operator is also interested in the *actual* behavior of ServiceB without Gremlin's interference (e.g., during multiple injected faults), specifying `withRule=False` will return such untampered observations.

Second, base assertions can be chained using a special `Combine` operator to evaluate a sequence of operations. Consider the following combination:

```
Combine(RList, (CheckStatus, 503, 5, True),
        (AtMostRequests, '1min', False , 0))
```

This assertion can validate the initial behavior of a circuit-breaker design pattern—upon seeing five API call failures, the caller service should backoff for a minute, before issuing more API calls to the same callee. Note that `Combine` automatically "discards" requests that have triggered the first assertion before passing `RList` to the second.

We have also built a number of pattern checks into the assertion checker, using the base assertions. In our example, `HasBoundedRetries` can be implemented as follows:

```
1 def HasBoundedRetries(Src, Dst, MaxTries):
2   RList = GetRequests(Src, Dst)
3   Combine(RList, (CheckStatus, 503, 5, True),
4           (AtMostRequests, '1min', False, MaxTries))
```

Here, if five replies with error codes are observed by `Src`, then `Src` should send at most `MaxTries` more requests to `Dst` within the next minute.

Chained failures. The operator can take advantage of Python and its constructs to create complex test scenarios by interacting with the control plane. Once again, consider Example 1, which could be expanded into a multi-step test as follows:

```
1 Overload(ServiceB)
2 if not HasBoundedRetries(ServiceA, ServiceB, 5):
3   raise 'No bounded retries'
4 else:
5   Crash(ServiceB)
6   HasCircuitBreaker(ServiceA, ServiceB, ...)
```

First, this introduces an `Overload` failure of ServiceB. If ServiceA implements a bounded-retry pattern, the operator can proceed to emulate a different type of failure, namely a `Crash`, to determine if ServiceA has a circuit breaker.

	Call	Description
Queries	`GetRequests` (`Src`, `Dst`, `ID`)	Return all observed requests between `Src` and `Dst` services filtered by the request `ID` pattern.
	`GetReplies` (`Src`, `Dst`, `ID`)	Return all observed replies between `Src` and `Dst` services filtered by the request `ID` pattern.
	`NumRequests` (`RList`, `Tdelta`, `withRule`)	Compute the number of requests in the given request (or reply list). Optional `Tdelta` limits the time window for which the computation is performed. `withRule` is a boolean parameter specifying whether Gremlin actions (such as `Abort` or `Delay`) should be taken into account.
	`ReplyLatency` (`RList`, `withRule`)	Compute the time for replies to arrive for each reply in `RList`. `withRule` is a boolean (see above).
Base assertions	`AtMostRequests` (`RList`, `Tdelta`, `withRule`, `Num`)	Check that at most `Num` requests have been sent within time window `Tdelta`. `withRule` is a boolean (see above).
	`CheckStatus` (`RList`, `Status`, `NumMatch`, `withRule`)	For given request list `RList`, check that at least `NumMatch` requests have returned status `Status`. `withRule` is a boolean (see above).
	`RequestRate` (`RList`)	Compute and return the rate of requests (req/sec) in the given list `RList`.
	`Combine` (`RList`, (`Assertion`, *args*)...)	Combine multiple base assertions, wherein each assertion "consumes" portion of requests when it is `True`, in the style of a state machine.
Pattern checks	`HasTimeouts` (`Src`, `MaxLatency`)	Check that `Src` replies to it's upstream services within `MaxLatency`.
	`HasBoundedRetries` (`Src`, `Dst`, `MaxTries`)	Check that `Src` implements a bounded-retry pattern when making API calls to `Dst`.
	`HasCircuitBreaker` (`Src`, `Dst`, `Threshold`, `Tdelta`, `SuccessThreshold`)	Check that `Src` implements a circuit breaker when making API calls to `Dst`. `Threshold` failed requests triggers absence of calls for `Tdelta` time. `SuccessThreshold` requests should close the circuit breaker.
	`HasBulkHead` (`Src`, `SlowDst`, `Rate`)	Check that a service has a bulkhead—ensures that service request rate is at least `Rate` to dependents other than `SlowDst`.

TABLE 3: Subset of the interface exposed by the Assertion Checker. Base assertions can be composed using the `Combine` operation to build more complex assertions. The Assertion Checker also provides checks to validate presence of recommended resiliency patterns (see Section 2.1).

5. Example recipes

In this section, we demonstrate how to create reusable recipes and emulate outages similar to those that affected microservice applications in the past.

We show a few examples of service failures that are built on top of the previously described `Abort`, `Delay` and `Modify` primitives. For instance, consider a disconnect primitive, which returns a HTTP error code when Service1 sends a request to Service2:

```
def Disconnect(Service1, Service2):
  Abort(Service1, Service2, Error=503,
        Pattern='test-*', On='request', Probability=1)
```

Internally, this instructs the Gremlin agent of Service1 to abort all (`Probability`=1) test requests (based on ID pattern) and return the 503 error code. A network partition is implemented using a series of `Abort` operations with a TCP-level reset along the cut of an application graph (not shown for brevity). A `Crash` failure of a service can be created by aborting requests from all dependent services to the service in question (for brevity, we assume existence of functions such as `dependents` and `services` that return dependents of a service and the list of all services, respectively):

```
def Crash(Service1):
  for s in dependents(Service1):
    Abort(s, Service1, Error=-1, Pattern='test-*',
          On='request', Probability=1)
```

The `Error`=-1 instructs the agents to terminate the connection at the TCP level, and return no application error codes to service `Src`, thus emulating an abrupt crash. *Transient* crashes can be simulated by reducing the probability, while software hangs are simulated using long delays (e.g., 1 hour):

```
def Hang(Service1):
  for s in dependents(Service1):
    Delay(s, Service1, Interval='1h',
          Pattern='test-*', On='request')
```

An overload of a service can be simulated using a combination of `Delay` and `Abort`:

```
def Overload(Service1):
  for s in dependents(Service1):
    Abort(s, Service1, Error=503, Pattern='test-*',
          On='request',  Probability=.25)
    Delay(s, Service1, Interval='100ms',
          Pattern='test-*', On='request',
          Probability=.75)
```

Here, Gremlin delays 75% of requests between Service1 and its neighboring services by 100 milliseconds and aborts 25% of requests with an error code. The `Modify` operation can be used for input validation. For instance, if Service1 returns a `key=value` pair and a success status (i.e., 200), Gremlin can modify the `key` to trigger unexpected behavior in services that depend on Service1:

```
def FakeSuccess():
  for s in dependents(Service1):
    Modify(s, Service1, Pattern='key',
           ReplaceBytes='badkey')
```

Next we show Gremlin's applicability to emulating real-world outages, by modeling a subset of them that occurred in the last 3 years (recall Table 1 in Section 1). Where applicable, we describe the assertions that could have caught the unexpected behavior. For clarity of explanation, we make some simplifying assumptions about the application graph.

Cascading failures caused by middleware. In October 2013, Stackdriver experienced an outage [27], when its Cassandra cluster crashed. Data published by various services

into a message bus was being forwarded to the Cassandra cluster. When the cluster failed, the failure percolated to the message bus, filling the queues and blocking the publishers, causing the entire application to fail. Example recipe:

```
Crash('cassandra')
for s in dependents('messagebus'):
  if not HasTimeouts(s, '1s')
    and not HasCircuitBreaker(s, 'messagebus', ...)):
    raise 'Will block on message bus'
```

Data store overload. In July 2014, BBC Online experienced a very long outage of several of its popular online services including the BBC iPlayer [18]. When the database backend was overloaded, it started to throttle requests from various services. Services that had not cached the database responses locally began timing out and eventually failed completely. Example recipe:

```
Overload('database')
for s in dependents('database'):
  if not HasCircuitBreaker(s, 'database', ...)
    raise 'Will overload database'
```

A very similar overload scenario has caused a Joyent outage [23] in July 2015, when an overloaded PostgreSQL database caused multiple delayed and canceled requests. We note that the same recipe can be reused for different applications by changing the application graph.

6. Implementation

Data plane. To implement the Gremlin agent, we leverage a *service proxy* [15]. The service proxy acts as a Layer-7 router, handling outbound calls from a microservice. It is well-suited for Gremlin, as the proxy already has natural access to the messages passing through the application. There are several ways of implementing the service-proxy capabilities. The first is called a *sidecar approach* used by companies such as AirBnB [24] and Yelp. In this model, the service proxy runs as a standalone process in the same Docker [30] container or virtual machine as the microservice. Communication between the microservice and the service proxy occurs over the loopback network interface, reliably and with low overhead. The second approach, followed by companies like Netflix and Spotify, is to provide language-specific libraries [33]. In either case, the service proxy remains immune to the evolution of the application logic. The control plane can interact with any service-proxy element that supports the primitives described in Table 2.

Our Gremlin agent, written in Go, is a reference service proxy based on the sidecar approach and supports fault-injection interfaces described earlier. It can be configured via a REST API by the control plane and can be run alongside unmodified microservices that use the sidecar-style service proxy. In this model, microservices specify their external dependencies to the service proxy in the form of a configuration file containing a list of mappings in the form of `localhost:<port> - (list of <remotehost>[:<remoteport>])` corresponding to each dependent microservice. Such mappings can be statically specified, or be fetched dynamically from a service registry.

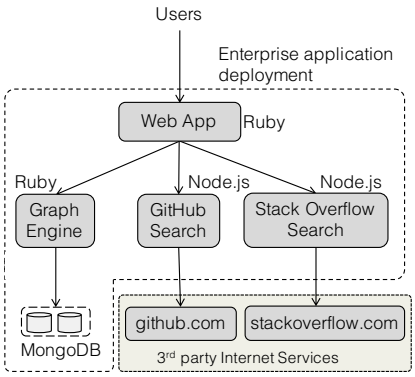

Figure 4: Architecture of the enterprise application in our case study. The user interacts with the Web App, which in turn depends on several backend services, both internal and external to IBM.

Control plane. We have implemented the Gremlin Recipe Translator, the Failure Orchestrator and the Assertion Checker as reusable Python libraries. Gremlin recipes are written as Python programs that leverage our libraries. We chose standard logging pipelines, such as logstash [32], to collect logs from the Gremlin agents and store them in Elasticsearch [32]. The Assertion Checker's `GetRequests` and `GetReplies` are queries to Elasticsearch to obtain records required for recipes' assertions. Further computation on the log records is specific to each assertion.

Test input generation. Our current implementation does not address the task of injecting test load that will traverse through the desired chain of microservices. We assume that the developer is aware of specific types of user requests that would traverse the microservices she is trying to test. Alternatively, standard load-generation tools can be used to easily inject arbitrary load into the system. To validate behavior of user-facing services, we assume that test load can be injected via a Gremlin agent, thus allowing us to log the behavior of edge services.

7. Evaluation

In this section we show that Gremlin is effective in testing existing real-world applications with no source-code modifications, and is capable of triggering previously unknown bugs. In addition, we benchmark different aspects of Gremlin to demonstrate its suitability for testing microservice-based applications.

7.1. Case Studies

Enterprise Application. We used Gremlin to test a proprietary application developed at IBM and deployed in the IBM Container Cloud platform. The architecture of the application is shown in Figure 4. At a high level, the application enables developers to search for web services with specific capabilities, such as mobile payments, location services, etc. The user-facing Web App shows the web services matching the search criteria (e.g., PayPal API, Google Maps API), characteristics shared across the other similar services, and development activities surrounding them, such as open-source projects from `github.com` and questions and an-

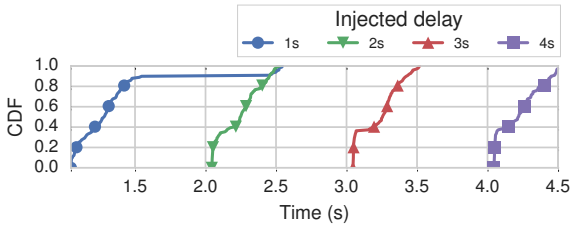

Figure 5: CDFs of response times from WordPress, based on injected delay between WordPress and Elasticsearch. Quickest response times were dictated by the delay, indicating absence of a timeout pattern.

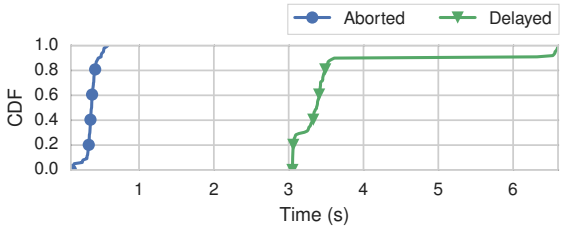

Figure 6: CDFs of response times from WordPress: first for aborted, then immediately delayed requests. None of the delayed requests returned without delay, indicating absence of a tripped circuit breaker after 100 consecutive failed requests.

swers from forums like `stackoverflow.com`. The application consists of two Ruby-based microservices and two Node.js microservices. Additionally, the application makes API calls to `github.com` and `stackoverflow.com`.

The development teams of various microservices agreed to use Gremlin to write recipes that tested for various failure scenarios. The developers were able to quickly design Gremlin recipes for various outage scenarios, using Gremlin's built-in failure patterns and assertions. Our case study with the enterprise application developers resulted in the following outcomes: *1)* We confirmed that developers require a minimal learning curve to start using Gremlin in their production applications; *2)* One team found bugs in their microservice's failure-handling logic *prior to running tests*, simply by virtue of writing a recipe; and *3)* Developers discovered a previously unknown bug in the timeout failure handler, after running a Gremlin recipe. This indicates that systematic testing and Gremlin in particular are of value to microservice developers.

Elaborating on the last outcome, developers of the Web App microservice relied heavily on the `Unirest` [35] library for abstracting boilerplate failure-handling logic. Emulating network instability between the Web App and backend microservices led to the discovery that the `Unirest` library's implementation of the *timeout* resiliency pattern did not gracefully handle corner cases involving TCP connection timeout; instead the errors percolated to other parts of the microservice. We consider this to be a serious issue, given that the `Unirest` library is actively used (with over 38,000 downloads) for abstracting away common failure-handling behavior in Ruby applications.

WordPress. We used Gremlin to conduct a test of a WordPress plugin called ElasticPress [31]. The plugin enhances

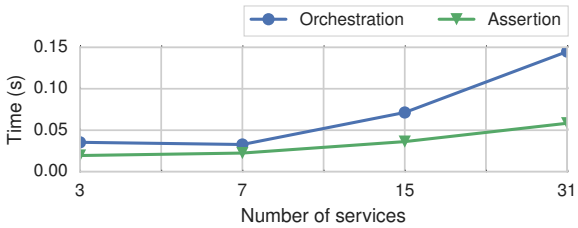

Figure 7: Time to orchestrate an outage and run assertions as a function of number of services in the application.

search capabilities of WordPress by using Elasticsearch for indexing and querying published data, therefore enabling additional features (e.g., fuzzy search).

Our deployment consisted of three unmodified services: WordPress (with ElasticPress enabled), Elasticsearch, and MySQL database (required for Wordpress). We injected basic `Delay` and `Abort` failures between WordPress and Elasticsearch to examine the plugin's behavior. We found that ElasticPress handled failure gracefully and fell back to the default (MySQL-powered) search method when Elasticsearch instance was unreachable or returned an error.

However, the plugin had no circuit-breaker or timeout support. In addition to failing the respective Gremlin assertions, the lack of these resiliency patterns impacted the response times of WordPress. Figure 5 shows CDFs of response times from WordPress, for different delays injected between WordPress and Elasticsearch. Response times were always offset by the injected delay, indicating that the plugin implemented no timeout patterns. We also crafted an `Overload` test of an Elasticsearch instance, where Gremlin aborted 100 consecutive requests from WordPress to Elasticsearch, then immediately delayed the next 100 by three seconds. If a correct implementation of a circuit breaker were present, a portion of the requests would have returned immediately. Figure 6, however, shows that all delayed requests completed only after three seconds.

7.2. Benchmarks

Setup. Since many microservice applications use Docker for deployment, we packaged a naive Python-based application along with the Gremlin agent into a Docker container. We then deployed the containers in different configurations by constructing binary trees of various depths and using them as the application graph.

Orchestration and assertions. We measured the time to setup an outage and run assertions on the collected data. We setup an outage for different application graphs scenario that impacts *all* services (for consistency, we use the `Delay` fault). We then injected 100 test requests into the system, followed by execution of an assertion for *every* service in the system. Figure 7 shows the time to execute a test as a function of the number of services in the application. Time is broken up into two components: failure orchestration, and assertions. This shows that the orchestration and assertion parts of the test induced low overhead. Even counting the time to inject 100 requests, the test was completed in under

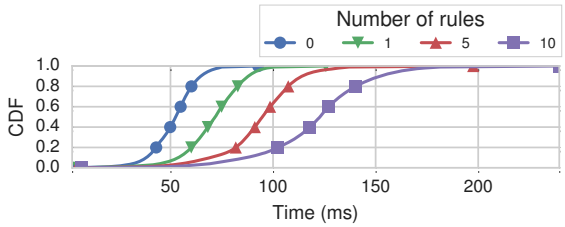

Figure 8: Worst case overhead of rule matching, wherein a request was compared against all available rules, but matched none.

one second, easily allowing the operator to run multiple tests and support chained failure/assertion pairs.

Proxy benchmarks. The rule matching process of the Gremlin agent introduces some overhead to the application because the proxy must process requests in-line with the data path. We evaluated the proxy overhead in a controlled manner by using the Apache Benchmark tool. We measured the time to complete a series of HTTP requests to a server through the service proxy with different number of rules installed. Figure 8 shows the CDF for completing 10000 requests in the worst case scenario: request IDs were compared against all rules without a match, prior to being forwarded. Since the proxy is not the main contribution of this work, the measurements do not reflect the optimizations that could be implemented to reduce this overhead, such as using more efficient regular expression engines, or structured (e.g., prefix-based or numeric) request IDs.

8. Related Work

There is a rich set of literature on testing the resiliency of distributed systems. We compare and contrast our work with other software and hardware-based techniques for failure injection and behavior validation.

8.1. Fault Injection in Service Oriented Architectures

Gremlin shares similarities with WS-FIT [14], a tool for dependability testing of SOAP RPC-based web services. Both frameworks inject faults by manipulating messages at the application layer between web services. However, WS-FIT lacks the ability to analyze the behavior of the application after the test is completed. Genesis2 [12] and PUPPET [2] focused on model-based testbed generation for conducting fault-injection experiments. The testbed consists of stubs representing application services and infrastructure components. In contrast, Gremlin operates on live services, enabling developers to test real services whose behavior may deviate arbitrarily from their stub counterparts.

Chaos Monkey is a randomized fault-injection tool from Netflix, being used at large scale in production [34]. It is capable of staging unforeseen faults that were not captured by systematic testing, and has the ability to kill an entire availability zone or a region of the application. However, the tool lacks support for automatically analyzing application behavior, which is necessary to quickly zero in on implementation bugs. Moreover, faults injected by Chaos Monkey cannot be constrained to a subset of requests or services.

8.2. Resiliency Testing using Code Injection

Setsudo [11] is a perturbation-based testing framework for Java-based distributed systems. It provides a policy language for specifying high-level failures, and minimal facilities for verifying application behavior. Failure injection happens on the I/O path between components of the distributed system. Similarly, In FATE and DESTINI [7], the authors describe a system to dynamically inject low-level I/O related failures into Java-based distributed systems and execute assertions. Our work differs from these systems in two key aspects: *1)* Gremlin's approach is suitable for polyglot microservice deployments, unlike aforementioned systems, which are constrained to Java-based systems in order to take advantage of AspectJ for injecting faulty code. *2)* Gremlin does not require knowledge of microservice internals nor modifications to the source code.

8.3. Low-level Fault-Injection

Doctor [8] provides a framework to inject CPU, memory and network faults, and a way to analyze the generated error logs. It relies on dedicated hardware for testing. Orchestra [4] injects protocol-level faults in a distributed system built on top of the *x*-kernel, along with the ability to script various types of faults at a high level. While Gremlin does not provide the ability to inject system-level faults pertaining to CPU, memory, or disk I/O, it is capable of simulating a wide variety of faults in modern distributed applications in an OS-agnostic manner, without the need for dedicated hardware.

9. Discussion & Future Work

State cleanup. Unlike unit testing, there is no straightforward way to re-initialize an operational application, to ensure that each test is performed on a fresh copy of the application. Even when faults are injected only on synthetic test requests, implementation bugs could cause the microservice to crash, affecting real users. Other side effects can include data-store values that Gremlin itself cannot clear. We do not see this limitation as a barrier to adoption as other frameworks for integration testing of applications in production are being widely used without significant issues, despite sharing similar limitations. One possible solution is the use of *canaries*—copies of a microservice dedicated to handling test requests.

Automating recipe generation. A key area for further exploration is automatic generation and execution of Gremlin recipes. Given semantic annotations to the application graph, it might be possible to automatically identify microservices and resiliency patterns in need of testing, then construct and run appropriate recipes. We leave detailed exploration of these ideas to future work.

10. Conclusion

While designing for failures is critical in microservice architectures, we argue that it is equally important to test the recovery capabilities of the application. The heterogeneous development environment necessitates a runtime-agnostic testing framework that can withstand the rapid pace of

application development. To this end, we presented Gremlin, a purely network-oriented, systematic resiliency-testing framework inspired by SDN. We described the design and use cases of Gremlin, and demonstrated its suitability to microservice applications. Our case studies show that Gremlin has a minimal learning curve and helps uncover bugs in failure-recovery code. Our benchmarks show it has low overhead, making Gremlin a valuable tool for resiliency testing of microservice applications.

Acknowledgments

The authors thank Mandana Vaziri, Chris Young, Saurabh Sinha, and Tamar Eilam from IBM Research, and Alexey Lapitsky from Spotify, for their feedback and comments on the drafts of this paper. This work was partially supported by the NSF Graduate Research Fellowship.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg, "Failure Detection and Consensus in the Crash-Recovery Model," *Distributed computing*, vol. 13, no. 2, 2000.

[2] A. Bertolino, G. De Angelis, and A. Polini, "A QoS Test-Bed Generator for Web Services," in *Proc. of International Conference on Web Engineering*, 2007.

[3] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2014.

[4] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection," in *Proc. of IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1996.

[5] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, 1st ed. Addison-Wesley Professional, June 2007.

[6] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.

[7] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2011.

[8] S. Han, K. G. Shin, and H. A. Rosenberg, "Doctor: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems," in *Proc. of International Computer Performance and Dependability Symposium*, 1995.

[9] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft, 2014.

[10] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, July 2010.

[11] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "SETSUDO: Perturbation-based Testing Framework for Scalable Distributed Systems," in *TRIOS: Conference on Timely Results in Operating Systems*, 2013.

[12] L. Juszczyk and S. Dustdar, "Programmable Fault Injection Testbeds for Complex SOA," in *Service-Oriented Computing*, 2010, pp. 411–425.

[13] Lin, Haoxiang and Yang, Mao and Long, Fan and Zhang, Lintao and Zhou, Lidong, "MODIST: Transparent Model Checking of Unmodified Distributed Systems," in *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009.

[14] N. Looker, M. Munro, and J. Xu, "WS-FIT: A Tool for Dependability Analysis of Web Services," in *Proc. of IEEE Computer Software and Applications Conference*, 2004.

[15] S. Neuman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, February 2015.

[16] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*. The Pragmatic Programmers, 2007.

[17] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, A Large-Scale Distributed Systems Tracing Infrastructure," *Google research*, 2010.

[18] "BBC Online Outage on Saturday 19th July 2014," http://www.bbc.co.uk/blogs/internet/entries/a37b0470-47d4-3991-82bb-a7d5b8803771, July 2014, [ONLINE].

[19] "CIRCLECI. DB performance issue," http://status.circleci.com/incidents/hr0mm9xmm3x6, July 2015, [ONLINE].

[20] "GRUBHUB. Enabling Continuous (Food) Delivery at GrubHub," *DockerCon (2015)*.

[21] "HUBSPOT. How We Deploy 300 Times a Day," http://product.hubspot.com/blog/how-we-deploy-300-times-a-day, November 2013, [ONLINE].

[22] "ORBITZ. Enabling Microservices at Orbitz," *DockerCon (2015)*.

[23] "Postmortem for july 27 outage of the manta service," https://www.joyent.com/blog/manta-postmortem-7-27-2015, July 2015, [ONLINE].

[24] "AIRBNB. SmartStack: Service Discovery in the Cloud," http://nerds.airbnb.com/smartstack-service-discovery-cloud/, October 2013, [ONLINE].

[25] "PARSE.LY. Kafkapocalypse: a postmortem on our service outage," http://blog.parsely.com/post/1738/kafkapocalypse/, March 2015, [ONLINE].

[26] "SPOTIFY. Incident Management at Spotify," https://labs.spotify.com/2013/06/04/incident-management-at-spotify/, June 2013, [ONLINE].

[27] "STACKDRIVER. Report on October 23 Stackdriver Outage," http://www.stackdriver.com/post-mortem-october-23-stackdriver-outage/, October 2013, [ONLINE].

[28] "TWILIO. Billing Incident Post-Mortem: Breakdown, Analysis and Root Cause," https://www.twilio.com/blog/2013/07/billing-incident-post-mortem-breakdown-analysis-and-root-cause.html, July 2013, [ONLINE].

[29] "TWITTER. Distributed Systems Tracing with Zipkin," https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin, June 2012, [ONLINE].

[30] "Docker Containers - Build, Ship, and Run Any App, Anywhere," https://www.docker.com/, [ONLINE].

[31] "ElasticPress Plugin: Integrate Elasticsearch with WordPress," https://wordpress.org/plugins/elasticpress/, [ONLINE].

[32] "An Introduction to the ELK Stack," https://www.elastic.co/webinars/introduction-elk-stack, [ONLINE].

[33] "HYSTRIX. Latency and Fault Tolerance for Distributed Systems." https://github.com/Netflix/Hystrix/, [ONLINE].

[34] "Netflix - Chaos Monkey Released Into The Wild," http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, July 2012, [ONLINE].

[35] "Unirest for Ruby - Simplified, lightweight HTTP request library," http://unirest.io/ruby.html, [ONLINE].