Resilient Microservices without the Chaos

Thesis Proposal: Christopher S. Meiklejohn

Revised: 2022-07-31

This proposal makes fundamental contributions to the Distributed Systems and Software Engineering communities by investigating the causes of partial failure in microservice applications, proposing new algorithms for both identifying and managing those failures, and engineering solutions that implement those algorithms, thus providing advancing the state of the art in both fault prevention and fault tolerance.

1 The Widening Gap Between Research and Practice

The widespread adoption of both microservice architectures and cloud computing services have forced developers to deal with a new type of application complexity: *partial failure*, where one or more of the services that their application depends on to provide service to customers may be unavailable or malfunctioning. As a result of this additional complexity, the developers of these applications increasingly need to turn to both *fault prevention* and *fault tolerance* techniques in order to ensure the resilience of their application when failure inevitably strikes.

Despite the extensive academic research on both fault prevention and fault tolerance in distributed systems, the overwhelming majority of this literature is focused on the implementation of stateful distributed protocols, where distribution is used primarily to provide either both scalability or fault tolerance. Microservice applications are not implementations of stateful distributed protocols, however: while they may benefit from improved scalability and fault tolerance, these applications choose distribution primarily for improving developer productivity. Therefore, microservice applications look quite different from more traditional distributed data systems: they favor a large number of highly modularized distributed components, both stateful and (effectively) stateless, implemented in different programming languages, where no single specification of application behavior exists. These architectural distinctions make the successful application of existing fault prevention and fault tolerance techniques, designed with distributed data systems in mind, difficult to apply in practice to microservice applications.

This ever widening gap between research and practice is problematic, especially as microservice applications built using cloud computing services become the most common type of distributed system written and deployed today. In this proposal, I detail a comprehensive research agenda focused on (A) the design and implementation of new techniques for both fault prevention and fault tolerance in microservice applications; and (B) the evaluation of these techniques on both a newly constructed microservice application corpus and in an industrial setting where a microservice application powers a large food delivery platform.

1.1 Fault Prevention and Fault Tolerance

In order to understand where existing fault prevention and fault tolerance research is insufficient for microservice applications, we look at two different classes of applications: distributed data systems (DDS) and microservice applications (MSA) and compare their architectural characteristics.

We depict examples of one DDS, Amazon's DynamoDB, a large-scale key-value database, and one MSA, the microservice application behind Audible, an audiobook streaming service owned by Amazon, in Figure 1.

Where applicable, we use the standard definitions of *fault*, *failure*, and *error*, as well as both *active* and *latent* classes of application faults, as defined by the IEEE Computer Society's Technical Committee on Dependable Computing and Fault-Tolerance [42].

1.1.1 Distributed Data Systems

Distributed data systems (DDS) (*e.g.*, Amazon DynamoDB) are stateful systems that use distribution to increase system scalability. By increasing system scalability in this manner, the probability of at least one node of the distributed system being down at a given time increases. Therefore, to ensure availability, these systems implement replication: where one node can stand in for the failure of another node in the same replica set. These systems implement one or more well-studied replication protocols that are designed to tolerate a set number of specific failures — for example, quorum replication, which can withstand any number of failures where a majority of the nodes in a replica set remain online and operational. When failures outside of this failure bound, the system ideally returns an error to the user saying that their request is not possible at that time. These replication protocols can be specified, as they often are, and then mechanically verified using model checking or theorem proving.

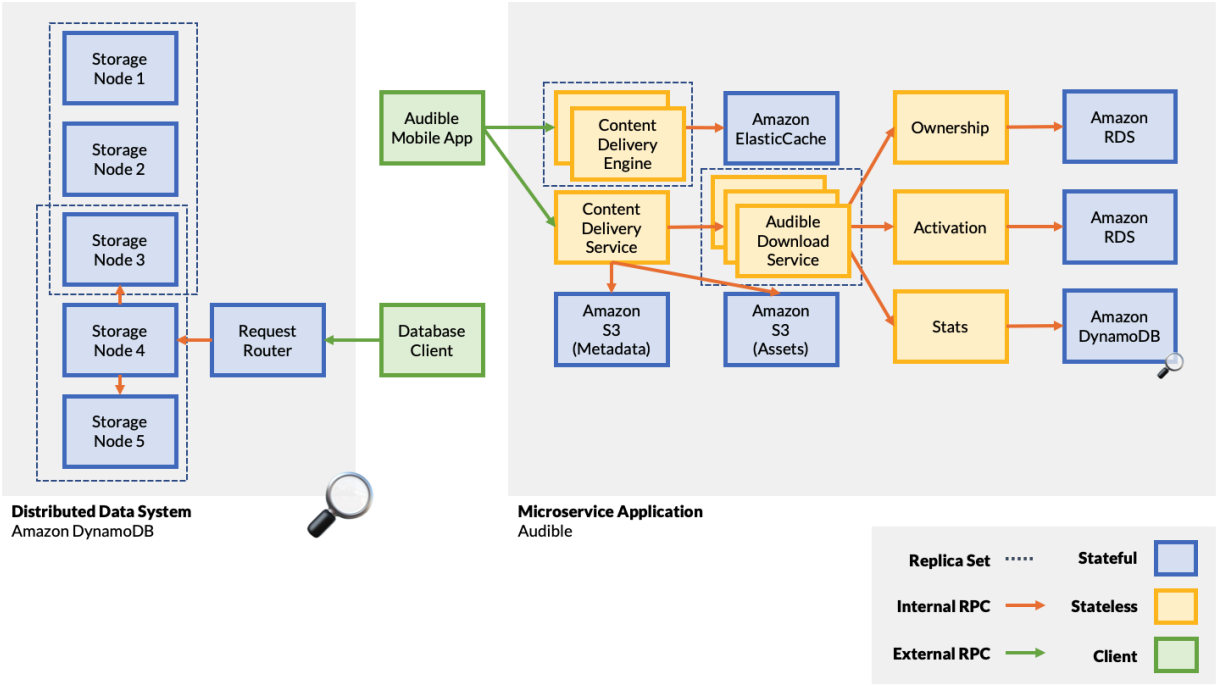


Figure 1: Architectural differences between distributed data systems (DDS) and microservice applications (MSA), noting that microservice applications compose distributed data systems by combining several different systems, using stateless services, into a single user application.

Fault tolerance is core to DDSs and is at the forefront during their implementation. The faults that are of primary concern are that of replica crashes and message omission due to network partitioning, where the network splits into one or more segments due to an underlying failure and messages sent between replicas are lost. Therefore, to increase availability of the system when faults do occur, two techniques have become commonplace in these systems: first, message retries to ensure reliable message delivery and second, timeouts to detect failures of remote nodes. As the use of these techniques are historically both ad-hoc and error prone, fault prevention techniques, such as fault injection, are used to verify that fault tolerance works as desired: for example, by ensuring that timeouts are not under-specified, which may result in false negatives, or by ensuring that replayed messages are properly deduplicated and do not result in processing of the same message twice. Inherent in the way that these systems are tested is the assumption that all components share a common fault model: they all fail in the same manner.

1.1.2 Microservice Applications

In contrast to DDSs, microservice applications (MSA) (*e.g.*, Audible) can be viewed as applications that compose stateful DDSs with stateless[1] services that contain the business logic of the application. In many cases, these DDSs are closed source and often provided as pay-as-you-go cloud computing services: for example, Amazon's DynamoDB, which was discussed in the previous section.

In MSAs, the stateless services that contain the core business logic of the application are both developed and deployed incrementally by independent teams within the organization. This is done to provide optimal developer productivity: a necessity as applications get too large for a single developer to understand, deploy, and operate the entire system themselves. They are tested independently using unit and integration tests and then, once integrated into the system, tested using functional end-to-end testing. In contrast to DDSs, there are no specifications of behavior and in many cases, the desired behavior of the application, under fault, may be unknown.

MSAs have to address the same faults as DDSs as they are both distributed applications. However, often these faults are handled differently in application code and are exacerbated by the use of an MSA, where decentralized, incremental development and deployment are core to the MSA design choice. This simplifies and complicates both fault prevention and fault tolerance, as described below.

First, when it comes to message omission faults and crash failures, unlike DDSs, application code rarely addresses the first occurrence of these faults. More often that not, the fault tolerance code that directly addresses the initial

[1]It should be obvious that no service is truly *stateless*: here, the term is used to draw a connection with the traditional 3-tier architecture for web applications where stateless front-end and back-end services retrieve data from storage to process each end-user operation and no state is explicitly managed by those services themselves [48].

occurrence of these faults is placed inside of infrastructure code or supporting frameworks used by the application. For example:

- **Message omission.**
  Message omission is less of an direct concern for application developers, as the modern Remote Procedure Call (RPC) frameworks used by applications for intra-service communication (*e.g.*, GRPC) typically re-transmit messages until they are acknowledged. Therefore, application developers need only concern themselves with the resulting application logic that is invoked when an RPC is received. Typically, application developers rely on well-known techniques for exactly-once processing[2] which are often already necessary to deal with customer-induced retries from both UI double-clicks and mobile application issued retries, possible when switching between cellular and wireless networks.

- **Crash failures.**
  Failure detection is also less of a direct concern for application developers, as most modern cluster orchestration systems (*e.g.*, Kubernetes) provide auto-scaling mechanisms for stateless services that detect these failures and create new instances when crash failure occurs: load balancing is used to distribute requests across this pool of instances. Therefore, the combination of auto-scaling with automatic retries means that requests are eventually retried against a non-malfunctioning, available replica of a stateless service [36].

Since this fault tolerance code is directly placed in either the infrastructure code or supporting frameworks and reused across all services in the MSA, its frequently tested in isolation as part of the unit or integration suite of these libraries or systems.

When message omission faults and crash failures are not tolerated by these mechanisms, the application can observe failures in the form of runtime exceptions, raised by the RPC framework or client library, or incorrect (*i.e.*, Byzantine) responses, returned by the RPC framework or client library. The latter can often occur when the client library or RPC framework contains either an active fault or latent fault that is activated by an unhandled fault. Similarly, when the error experienced by the application is unhandled (*i.e.*, a missing catch block, an example of a latent application fault, that when activated, results in a failure), it may result in the service exposing an error (*i.e.*, propagated) to services that take it as an explicit dependency.

Finally, application developers must also concern themselves with both active and latent application faults, specific to the business logic, in the stateless services that they develop. These bugs are exacerbated through the use of a MSA. We present three examples of faults that are exacerbated by the choice of a MSA in Figure 2.

First, (1) services in an MSA may contain **latent application faults.** For example, a service may fail to handle an error that is returned by a dependent service, when the dependent service experiences a fault. These may range from custom HTTP or GRPC status codes used to indicate an exceptional situation (*e.g.*, Precondition Failed, Deadline Exceeded, Resource Not Found) to runtime (*i.e.*, unchecked) exception types used to indicate the unavailability of a remote service. Ensuring that application code has proper error handlers for these faults is challenging in a MSA, where decentralized development may alter the APIs that are called by the developers of services without their knowledge by changing the existing or introducing new error types.

Second, (2) services in an MSA may contain **active application faults.** For example, a service may depend on another service that is redeployed with a bug that causes one or more RPC methods to fail when invoked. In this case, the entire service may still be online with many of its RPC methods working correctly; however, only one method is returning failures. These errors are commonplace in MSAs and exacerbated by the use of incremental, decentralized (continuous) deployment where one or more services of an application are deployed as often as many times a day.

In order to address (1) and (2), developers of MSAs often reach for commonly used fault tolerance techniques for MSA: fallbacks, circuit breaking, and load shedding.

- **Fallbacks.**
  With *fallbacks*, when a remote service returns an error, the application code contacts a different service to try to retrieve similar information. The canonical example here is Netflix where, when movie recommendations tailored to the user are not able to be retrieved, a set of movie recommendations based on the global user base is returned instead.

- **Circuit breaking.**
  With *circuit breaking*, a malfunctioning service is detected by accumulating counters for failures within a given window, where, when counters exceed a particular threshold, the RPC to the malfunctioning service is short circuited by immediately returning an error, ideally giving the remote service time to recover.

- **Load shedding.**
  With *load shedding*, an overloaded service avoids processing RPCs by immediately returning an error to the invoker in order to give itself time to recover before taking on additional work.

[2]For example, by embedding unique identifies in message that can be used for message de-duplication, if those messages result in side-effects that are not idempotent nor deterministic.

1. **Application code contains *latent* application fault.**
   Application may fail to handle an error response from a dependent service because it does not know to expect such a response (*e.g.*, HTTP status codes, GRPC runtime exceptions.)

   For example, Service A depends on Service B. Service B is redeployed with a change where a failed precondition GRPC error, a standard GRPC error code, is returned to the caller to indicate that a certain RPC did not provide enough information in the request for it to be successfully completed. Service A does not expect this new error response and therefore either throws an error, or returns an incorrect response, when this response is received.

   ✗ *Exacerbated by* the nature of decentralized development in MSAs, where the API specifications of dependent services may change without callers knowing.

2. **Application code contains *active* application fault.**
   Dependent service is redeployed with a bug that affects one or more, but not all, RPC methods causing the service to partially fail when that method is invoked by a service that depends on it.

   For example, Service A depends on Service B. Service B is redeployed with an active bug where a List operation is called to get the first element from a list without checking if the list is empty first. This results in Service B throwing a runtime exception that is uncaught and exposed to Service A as a HTTP internal server error. Service A does not expect an internal server error response from Service B.

   ✗ *Exacerbated by* the nature of incremental development and deployment in MSAs, where dependent services are deployed as necessary and incrementally, without redeployment of the entire application.

3. **Application's fault tolerance code contains *active* or *latent* faults.**
   Fault tolerance mechanism in application code fails to operate correctly. These faults may be be either *active* (*e.g.*, circuit breakers do not open, load shedding does not shed load) or *latent* (*e.g.*, fallbacks fail, timeouts incorrect when fallbacks used.)

   For example, Service A depends on Service B. Service B depends on Service C. Service B is inadvertently redeployed with a bug that disables the circuit breaker between Service B and Service C. When Service C begins responding slowly, Service B continues to call Service C causing Service B to overload with outstanding requests to Service C. Service B begins returning errors to Service A when it runs out of resources.

   ✗ *Exacerbated by* the lack of specifications or tests that describe desired behavior when faults are present.

Figure 2: Three types of faults exacerbated by microservice architectures and their development methodology that place the entire application at risk of cascading failure.

By combining these three techniques, developers encode alternative logic for when services are malfunctioning, avoid repeated invocation of failing services by short circuiting those requests with an error, and immediately reduce pressure on services that may be overloaded. This is done to reduce the risk of a cascading failure, where a failure in one or more dependent services induces subsequent failures in the calling services that often leads to application or service outages.

However, (3) often implementations or use of these **fault tolerance techniques may contain latent or active faults.** For example, circuit breakers may be missing at RPC sites or misconfigured; similarly, load shedding may also be missing or misconfigured. Circuit breakers and load shedding also suffer from scope issues: when used too coarsely, circuit breaking may short circuit correctly functioning RPCs of the application in an effort to contain a failure of one specific RPC. Fallbacks also introduce their own set of problems: fallbacks themselves can experience faults and fail. Finally, when timeouts are used for fault tolerance, the use of fallbacks can introduce additional delays causing timeouts to fire too early. Often, developers do not test fault tolerance code. When they do, they are left to answer the question of what the application should do when faults do occur: an increasingly difficult question to answer due to the nature of decentralized development inherent to MSAs.

1.2 Research Agenda

In this proposal, I detail a comprehensive research agenda that addresses the three types of faults common to, and exacerbated by, MSAs, presented in Figure 2.

In order to address (1) application code that contains latent application faults, I propose a novel testing strategy called Service-level Fault Injection Testing (SFIT) (§5). SFIT automatically identifies the possible errors exposed by dependent services and systematically tests each service for failures of its dependent services using the existing functional test suite of a MSA. By prompting the developer of an MSA when an assertion failure occurs with the precise faults that cause the assertion to fail, developers are forced to think through what should happen when a fault occurs, use an appropriate fault tolerance technique to address the fault, and encode the desired behavior of the application and the fault tolerance technique directly into the functional test suite. In order to evaluate the design of SFIT, I implement a prototype of SFIT called Filibuster and construct an open-source application corpus of MSAs, each containing one or more fault tolerance bugs, from a study of grey literature (§4) and demonstrate that Filibuster can identify all bugs. (§10.2)

To address the problems of testing MSAs at scale, where there may be hundreds of services in a single application, I first propose a novel indexing algorithm for RPCs called Distributed Execution Indexing (DEI) (§6) that allows for the precise targeting of an RPC in a MSA regardless of the use of function indirection, branching control flow or scheduling nondeterminism as a result of concurrent execution. DEI enables a fault space reduction technique in SFIT, called Dynamic Reduction (SFIT-DR) (§7), that removes redundant test executions. This reduction technique exploits a property of microservice application graphs called *service encapsulation*, whereby faults combinations that are only transitively visible by services that take that service as a dependency do not require explicit testing. Using the application corpus, I demonstrate that SFIT-DR can achieve upwards of 70% reduction on MSAs that exhibit the property of service encapsulation (§10.2). To justify the design decisions made in DEI, I perform an empirical evaluation using a real-world industrial MSA (§10.3.1).

In order to address (2) application code that contains active application faults, I study the existing fault tolerance techniques used by MSAs (*e.g.*, circuit breakers and load shedding) in order to identify their deficiencies. (§8) From this study, I propose new designs of fault tolerance techniques that address these deficiencies. I plan to perform an empirical evaluation of these new designs by extending the application corpus with supporting examples, derived from examples from a real-world industrial MSA, to demonstrate the new designs applicability to the problem of fault tolerance when application code contains active application faults. (§10.3.2)

In order to address (3) the problem of fault tolerance implementation and usage that contains active or latent faults, I plan to extend SFIT to support the style of fault injection required to properly test these fault tolerance mechanisms (*e.g.*, circuit breakers, load shedding.) (§9) This style of fault injection is both novel, as it does not appear in existing literature, and is of paramount importance to ensuring that the advanced fault tolerance techniques that are used in MSAs operates correctly when faults to occur. I plan to perform an empirical evaluation of this new testing strategy by extending the application corpus with supporting examples, derived from examples from a real-world industrial MSA, to demonstrate its applicability at identifying application bugs in fault tolerance code. (§10.3.2)

In order to demonstrate that the technical contributions made in this proposal are relevant to practice and address the ever widening gap between industrial practices and academic research, I plan to perform at least one of two additional empirical studies. (§10.3.3) First, I plan on performing a qualitative study of incident reports, taken from a large, real-world industrial MSA, to demonstrate that the technical contributions address pressing, contemporary concerns of the developers of MSAs. Second, I plan on reporting on the integration and usage of the technical contributions made in this proposal from a large, real-world industrial MSA to demonstrate applicability and use.

Finally, I present a proposed timeline for this thesis proposal. (§11)

2 Background and Related Work

Fault prevention is the process of identifying application vulnerabilities, often using fault injection, before and after the deployment of the application to production. One example of fault prevention in an MSA is when a latent application fault, which is activated by the unavailability of a service dependency, is identified as part of functional testing. Fault tolerance compliments fault prevention by ensuring that a latent application fault, when not prevented and subsequently activated, does not prevent the application from functioning correctly. One example of the use of fault tolerance in an MSA is when an undetected latent application fault is activated by the unavailability of a service dependency, the invoking service dynamically re-configures itself to avoid invoking the unavailable dependency until it becomes available again.

Fault prevention and fault tolerance have been identified as the two key categories of means for building a dependable system that can deliver a service in the presence of faults [42]. Despite this, too often the error handling code necessary for tolerating faults and preventing them from impacting the system adversely is not written or not tested by developers due to the required overhead, necessary effort, and inherent complexity. This remains true despite research [86] on open source DDSs that shows that many production failures could have been prevented with

simple testing of error handling code. This would seem to indicate that an automated approach to fault prevention, where both the application's business logic and its fault tolerance measures are tested, is the critical first step in building dependent (*i.e.*, resilient) MSAs.

In this section, I review both the industrial practices and academic research related to fault prevention and fault tolerance in MSAs. To understand how the techniques used by MSAs differ from the traditional techniques used by DDSs, I include a description of the relevant techniques for DDSs. I both highlight the deficiencies in the state-of-the-art and identify a clear trend where the industrial practices and nascent academic research is converging: it is at this convergence where I target this research proposal.

2.1 Industrial Practices

I first examine the industrial fault prevention and fault tolerance techniques that are used by the developers of MSAs today. I start with the presentation of industrial techniques, as practitioners are faced with faults daily and are highly motivated to identify practices that have real impact towards improving the resilience of their application.

2.1.1 Fault Prevention

Industrial practices for fault prevention in MSAs has historically relied on fault injection, *in production*, in order to determine a system's tolerance to a given fault. The collection of tools, techniques, and process that support this is colloquially known under the umbrella term of resilience engineering: a term that has its formal roots in the safety science [73] community and refers to the processes used by organizations and communities to adapt and respond to unanticipated failures.

Game Days, one of the earliest resilience engineering techniques used by practitioners and the spiritual successor of most of the approaches taken by practitioners today, have been used by Amazon, Google, and Stripe [75, 66] to identify resilience issues in both their applications and infrastructure. Game Days acknowledge that failure is inevitable at the scale that these companies operate at and therefore, they opt to preemptively trigger failures and explore the organizational response to those failures: for example, Google discovered through a Game Day exercise that their monitoring and alerting infrastructure existed only in the data center where they simulated a power outage. [75]

Chaos engineering is another resilience engineering technique, originally pioneered by Netflix when first moving to the cloud [9]. The first iteration of chaos engineering, Netflix's Chaos Monkey [29], randomly terminated instances in the live production cloud to ensure that Netflix's applications were resilient to instance failure: common, in the early iterations of Amazon's EC2 cloud environment. Next, Netflix's Simian Army [31, 1], a collection of tools for performing different types of fault injection, allowed developers to simulate increased latency, and failures of both EC2 availability zones and EC2 regions. Since then, chaos engineering has evolved into a discipline [76] practiced by many different companies, where its supported by a variety of different open source tools (*e.g.*, ChaosToolkit [21], ChaosMesh [20], ChaosBlade [19], Litmus [37], LinkedOut [4]), books [76], community meetups[3], and commercial software-as-a-service (SaaS) offerings (*e.g.*, Gremlin [13].)

As a discipline, chaos engineering closely resembles the scientific method: a hypothesis is formed about what the application will do when faults are injected, faults are injected in either on the entirety of, subset of, or mirror of production traffic, and the hypothesis falsified, if possible. Therefore, key behind the chaos engineering approach is application observation. In the case of Netflix, the key performance metric that is observed during chaos engineering experiments is a metric that counts the number of movie streams started per second, which varies little day to day making it easy to detect deviations from the norm when running a chaos experiment. This directly contrasts more traditional testing approaches where a test oracle is used that contains assertions about the application's desired behavior.

Netflix has continued to innovate in chaos engineering. Their Failure Injection Testing (FIT) [2] framework, for example, is integrated into the RPC framework that all of their services use for intra-service communication, allowing them to inject faults at any RPC site in their MSA. Their Chaos Automation Platform (CHAP) [47] enables automated failure testing with a minimal blast radius by automatically spinning up replicas of services where faults will be injected on a small percentage of their production traffic: in the event of a noticeable deviation from the norm in their key performance metric, the experiment is automatically terminated. Monocle [47] pushes this even further by examining the RPC configuration code that is associated with each of their services and automatically generates chaos experiments that are then automatically ran with CHAP. It is important to note that Monocle was recently disabled [6] due to the large number of experimental configurations is generated and the required overhead in running those experiments at scale. Finally, Gremlin, the chaos engineering SaaS company formed by former Netflix chaos engineers [13], also briefly promoted a product called "application-level fault injection" (ALFI), where a library-level fault injection approach was used to provide more granular fault injection with an even smaller blast radius and errors specific to the library in use. As of 2018, this product is no longer offered.

[3] `https://chaos.community`, now defunct.

Regarding the industrial practices for fault prevention in MSAs, there are a number of interesting observations that can be made. First, the adoption of chaos engineering techniques in practice seems to be related to two key aspects of chaos engineering: low-level fault injection and application observation. Low-level fault injection (*e.g.*, disrupting the network, terminating instances) is extremely low overhead for developers: for example, Gremlin [13] uses a daemon installed on the virtual machine instances of each service and requires no modifications to application code to perform fault injection. Application observation, via key performance metrics, is also low overhead when compared to the heavyweight specifications describing the behavior of the application in enough detail for mechanical verification or test oracles that contain assertions of application behavior under failure. These two key aspects seem to indicate that developers can easily "try out" chaos engineering before moving to more advanced techniques for fault prevention, which has presumably helped increase the wide adoption that chaos engineering has seen in recent years.

Second, the evolution of chaos engineering tools seems to indicate a desire for functionality that is traditionally seen in academic approaches: automation, granular fault injection, and library-level fault injection. For example, Monocle [47] automatically generates chaos experiments from software configuration: this resembles a traditional exhaustive or systematic search commonly found in approaches built on some form of model checking. ALFI, the abandoned approach from Gremlin [13], sought to provide fault injection in the libraries that services used for issuing RPCs and communicating with DDSs to allow granular fault injection with a minimal blast radius and library specific errors: this resembles traditional academic library-level fault injection approaches that aim to give developers confidence in proper API use and error handling of those libraries. However, both of these approaches have failed in their own way. For example, Monocle relies on experimentation in production using ChAP to automatically create and destroy clusters with a subset of production traffic for experimentation. This is an expensive task that could be reduced by either (A) experimentation in a staging or development environment; or (B) through the use of test case reduction techniques, commonly seen in academic approaches that employ model checking. ALFI [13], initially designed for returning library-specific errors using fault injection with a minimal blast radius required that developers manually instrument the libraries in use by the MSA. This is also an expensive task that could be reduced by either (A) experimentation in a staging or development environment; or (B) through the use of some sort of automatic instrumentation.

When considering (A), running chaos experiments in the staging or development environment is not as straightforward as it sounds. While the tools work in the same manner regardless of environment, the reliance on a key performance metric as the test oracle no longer works: the local development environment will not see any requests outside of what is issued by the developer; similarly, the staging environment may not as well. Therefore, in order to bring this style of experimentation into the local development environment, one must first solve the problem of the missing test oracle.

2.1.2 Fault Tolerance

When it comes to fault tolerance in MSAs, developers typically rely on a set of techniques specifically designed for the MSA context, in addition to the standard retries and timeouts often used by the developers of DDSs. These techniques are fallbacks, circuit breakers, and load shedding. Rather obviously, circuit breakers and load shedding are named after their counterparts in the field of electrical power management and delivery: both techniques used to prevent overload of a system in the event of one or more faults.

Fallbacks are used when a remote service is malfunctioning or unavailable in order to find alternative or replacement information from a different, properly functioning service. The canonical example here is from the streaming service, Netflix, where when movie recommendations tailored to the user are not able to be retrieved, a set of movie recommendations based on the global user base is returned instead. Fallbacks allow the system to keep functioning in the event of a fault, potentially with a degraded user experience.

Circuit breakers are also used when a remote service is malfunctioning or unavailable in order to relieve pressure on the remote service and to avoid waiting for a resource that will not respond in a timely manner. To achieve this, circuit breakers accumulate counters that reflect the number of successful and unsuccessful responses within a given window. When the counters exceed a particular threshold, the RPC to the malfunctioning or unavailable service is "short circuited" by returning an error immediately to the caller to indicate the circuit is open. Periodically, in an effort to close the circuit once the remote service begins functioning properly, a RPC is allowed to happen. Eventually, once the remote service fully recovers, the circuit moves back into the closed state.

Load shedding is a technique used when a remote service is overloaded and cannot respond in a timely manner to reduce pressure on that service. Where circuit breakers are located at the invocation site of an RPC, load shedding is located on the invokee side of an RPC. Load shedding compliments circuit breaking, as circuits may fail to fire quickly enough — or, the remote service may have multiple invokers whos combined load exceeds the service's capacity — to keep services functioning correctly. To achieve this, load shedding typically tracks the number of outstanding, concurrent requests, and, once a threshold is exceeded, requests are immediately "short circuited" by returning an error to the invoker (or, dropped silently.) These requests typically then cause the invokers circuit

breaker's counters to increment, if a circuit breaker is in place.

These three techniques are not a panacea of fault tolerance, however. For example, the use of fallbacks must be carefully considered, as in the event of a fault, the fallback may also be unavailable or malfunctioning. With circuit breaking, thresholds may be misconfigured or, when the circuit breaker is used too coarsely, might simultaneously disable correctly functioning components of the application while trying to contain a fault. With load shedding, the same type of faults can occur. Therefore, any fault prevention approach needs to also consider faults within the fault tolerance measures employed by the application.

For most MSAs, a combination of these three techniques, along with timeouts and retries, are used to prevent against the dreaded cascading failure, where a fault in one service left unhandled or improperly handled, propagates to the services that depend on it through the application's RPC graph, inducing further faults until the entire application fails.

2.2 Academic Research

In this section, we look at both the fault prevention and fault tolerance techniques for MSAs that have been the subject of academic study. In order to place this in the proper historical context, we also present related work on fault prevention in DDSs.

2.2.1 Fault Prevention

Academic research on fault prevention and fault tolerance in DDSs has historically built upon the rich history of model checking, either relying on specifications of either the external behavior of the system under test or the internal system state [41, 61, 64, 84, 85, 78, 62, 54]. However, successful application of these techniques to MSAs has been quite limited [40]. Outside of the challenges of mechanizing an approach that supports deterministic fault injection across multiple services implemented in multiple languages, one of the main challenges has been the lack of a test oracle that specifies behavior under failure. In industrial MSAs specifically, specifications that are rich enough to support model checking are rarely, if ever, written; similarly, the decentralized nature of MSA development prevents the use of global state invariants placed across all stateful services in an MSA. However, the developers of industrial MSAs are writing functional tests, and therefore it would seem that any successful approach should built on, and extend to cover behavior under fault, the test oracles that are already being written.

One key observation about MSAs is that intra-service communication is typically performed using client libraries (*e.g.*, AWS DynamoDB client) or RPC frameworks that are packaged as libraries (*e.g.*, HTTP via Java's Netty library, GRPC via Google's GRPC library.) Therefore, the existing research on library-level fault injection [49, 45, 65], which purports that low-level faults will manifest themselves as library-level errors in an application, may be a useful starting point for automated fault prevention techniques for MSAs. In fact, there is evidence of this: ChaosMachine [88], for example, uses a library-level fault injection approach to exercise and test an MSAs exception handlers. Library-level fault injection strikes a good middle-ground where library-specific errors can be simulated, automatically and exhaustively, without requiring the use of low-level fault injection to trigger them organically.

In contrast to library-level fault injection, recent academic approaches have proposed targeting the network-layer for fault injection. For example, Gremlin[4] [**?**] proposes the use of sidecar proxies at each node, where all RPC communication is routed through them before reaching the destination, for fault injection; this removes the requirement of local code modifications to support fault injection. Even further, ucheck [72] proposes the use of software defined networking (SDN) infrastructure for fault injection, removing the requirement for any code modification or additional infrastructure on each node. However, this style of low-level fault injection can prove problematic when trying to inject certain faults: for instance, triggering a GRPC failed precondition error, as opposed to a somewhat more straightforward GRPC service unavailable error. Effectively, the movement away from library-level fault injection, presumably done because of the costs of instrumenting each library the application uses to issue RPCs has made it both more challenging to inject certain types of faults and more costly in terms of computing resources and required infrastructure.

Despite advances in the fault injection methodology, the problem of the missing test oracle still remains unsolved. ucheck [72], which relies on fault injection in the SDN layer, still requires that application developers write state invariants that can be used for verification: not feasible for MSAs. In contrast, Gremlin [**?**], which also operates at the network-layer but instead uses sidecar proxies for fault injection, has no visibility into system state and only provides an assertion languages over the request patterns between different services: this requires developers, in addition to writing end-to-end functional tests also encode the manner in which communication occurs. Finally, ChaosMachine [88] eliminates the need for state invariants by allowing developers to specify, using annotations in application code, whether or not an injected fault will: be resilient (*i.e.*, no change on system state), observable (*i.e.*, by the user), debuggable (*i.e.*, creates log messages), or silent (*i.e.*, no derivation in state nor additional logging.) This proposal advances the academic work in the direction of commonly used industrial techniques (*c.f.*, chaos

[4] An academic research prototype [57], not to be confused with the SaaS company. [13]

engineering), however still remains somewhat disconnected from typical functional testing. A promising natural progression of this research is to explore specification of the test oracle as observable deviations from the application behavior when faults are not present.

The unfortunate casualty of this migration away from specifications is test case reduction. In DDSs specifically, test case reduction has typically relied on properties of the system under test: for example, symmetry reduction [64], where certain test cases can be avoided under the assumption that different replicas of the same service will behave identically. However, the lack of access to realistic MSAs [58, 83, 67], and the deficiencies in existing open-source corpora, which have not been operated at scale [83] nor contain realistic bugs specific to the choice of a MSA [89], have limited the ability of researchers to discover similar techniques for MSAs. Therefore, with increased access to descriptions or implementations of realistic MSAs and actual bugs that have been experienced in MSAs, discovery of test case reduction techniques will be possible.

2.2.2 Fault Tolerance

Research on MSA specific fault tolerance is quite limited, presumably due to the lack of access for academic researchers to realistic applications [58].

Several qualitative studies [39, 80, 63, 59, 70, 82, 74, 52, 43, 44] have identified circuit breakers as a core pattern used to improve reliability or availability (*i.e.*, resilience) in microservice applications. Most notably, Surendro and Sunindyo's systematic mapping study [79] identified a lack of existing research on circuit breakers when compared to other topics on microservice applications. They propose several areas for further academic investigation: specifically, more flexible and intelligent circuit breakers. In this proposal, I specifically target this deficiency in the existing literature.

Outside of qualitative studies, research on the implementation and configuration of circuit breakers is also quite limited. For example, previous work has explored the transparent application of circuit breakers in an academic programming language [69] and either the dynamic tuning of [77], or optimal configuration via model checking of [68], a circuit breaker's configuration parameters.

With respect to fault prevention as a mechanism for verifying the fault tolerance measures used by MSAs, there is also limited research. For example, Palliwar *et al.* [71] proposed distributed circuit breaking, where a gossip protocol is used to disseminate circuit breaker state across nodes in a cluster for faster detection of failures in a microservice application. However, often the concern of reaction speed of a circuit breaker is superceeded by concerns around the scope of the circuit breaker; in this proposal, we target this issue specifically.

Heorhiadi *et al.* [57] who proposed the fault prevention tool Gremlin, considered circuit breaker testing as part of the design of Gremlin and provided a mechanism for asserting that they operated correctly. To achieve this, Gremlin allows the developer to specify the errors and the thresholds under which a circuit breaker should fire. However, as discussed previously, the Gremlin assertion language only supports assertions about the RPCs issued in an application; therefore, even though circuit breaker testing is supported, it remains disconnected from application behavior under fault. In this proposal, I specifically look at integrating circuit breaker, and other fault tolerance testing, directly into an application's existing functional test suite.

Finally, researchers have motivated the design of data plane fault injection tools using circuit breakers; however, none of these designs contained anything specific for circuit breaker testing.

When it comes to load shedding as a fault tolerance technique in MSAs, I could find no relevant existing academic research.

2.3 Takeaways

When it comes to fault prevention in MSAs, there is a very clear trend when you examine both the existing academic research and industrial practices: they are converging.

Industrial practices seem to be evolving in the direction of more traditional academic research through the use of, or desire to use, (A) library-level fault injection, with (B) automation for generation of tests or chaos experiments, using (C) granular fault injection to minimize the blast radius. With (A), developers are realizing that in order to build more resilient systems, they need to be concerned with library-specific faults, in addition to low-level faults (*e.g.*, service unavailability, network partitioning); however, this approach remains difficult in practice as it requires instrumentation of each library used by an MSA. With (B), developers are beginning to use the application's configuration to identify remote calls and automatically create test scenarios for them; however, running these experiments in production, combined with the lack of test case reduction, makes running exhaustive exploration infeasible. Finally, with (C), developers are looking at ways for blast radius minimization, as many chaos experiments are still performed in the production environment; however, with a proper test oracle, it would be possible to run these experiments in a local development environment where blast radius minimization is not necessary.

Academic research seems to also be evolving in the direction of the contemporary industrial practices. For example, academic research has sought to solve the problem of heterogeneous implementation language usage in MSAs

through the use of fault injection at the network-level: however, this approach fails to account for faults that cannot be generated through low-level fault injection (*e.g.*, Precondition Failed, Not Found.) Similarly, recent academic research has acknowledged the problem of the missing test oracle, proposing solutions that build on application observation, as is common with chaos engineering. Despite this, the existing proposals still use assertions that rely on the technical aspects of the implementation: for example, messages sent between services [57] or whether or not the state will change and be observable by the end user when a fault occurs [88]. Rather unfortunately, manual experiment (*i.e.*, test) specification is still required with many of these techniques; as a result, little research into MSA specific test case reduction exists.

It is clear that academic research is being influenced by industrial practices, but without access to implementations of industrial MSAs, academics are left guessing at practical solutions to the most pressing of industry concerns. This extends itself to fault tolerance techniques for MSAs as well; while qualitative research that relies on developer interviews, questionnaires, and surveys identified circuit breakers as a main technique used for increasing the resilience of MSAs, studies performed on open-source MSAs failed to identify circuit breakers, load shedding, or fallbacks as resilience techniques used in MSAs. This seems to indicate, that while academic research and industry practices are converging on a similar design based on observation of each others techniques, academics, without access to industrial code bases, are left to infer the techniques and solutions that they feel will be useful to practitioners.

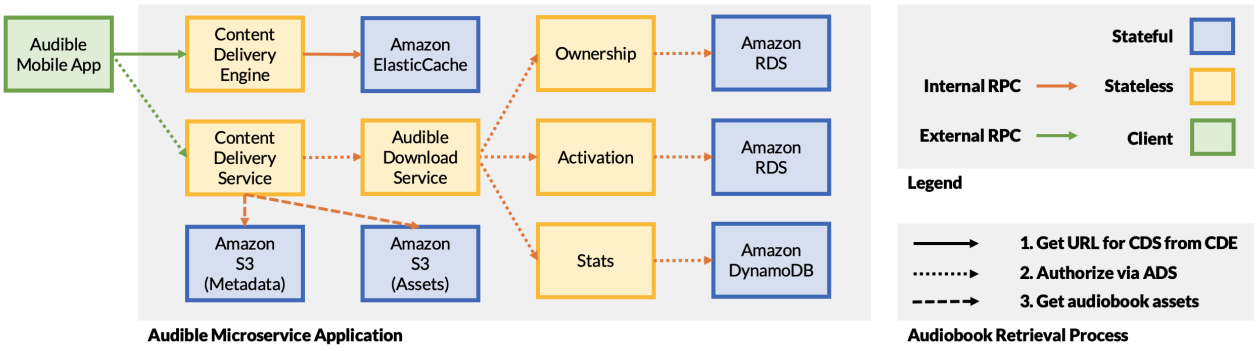


Figure 3: Audible microservice application with description of the audiobook retrieval process.

3 Motivating Example: Audible

To motivate this proposal, we provide an example of a real-world outage experienced by an MSA: an outage that Audible, an audiobook streaming service that is owned by Amazon, experienced between 2017-2018.

3.1 Application Structure

The Audible microservice application, depicted in Figure 3, is composed of several stateful and stateless services:

- **Content Delivery Service (CDS):**
  Given a book identifier and a user identifier, return the actual audio content and audio metadata after authorization;
- **Content Delivery Engine (CDE):**
  Returns the URL of the correct Content Delivery Service to contact to retrieve the audiobook using AWS ElastiCache;
- **Audible Download Service (ADS):**
  Orchestrates logging and authorization of the audiobook once ownership is verified;
- **Ownership Service:**
  Verifies ownership of the book using AWS RDS;
- **Activation Service:**
  Activates a DRM license for the user for the requested audiobook using AWS RDS;
- **Stats Service:**
  Maintains audiobook and DRM license activation statistics using AWS DynamoDB;

- **Asset Metadata Service:**
  Storage (AWS S3) for the audiobook asset metadata which contains information on chapter descriptions; and
- **Audio Assets Service:**
  Storage (AWS S3) for the audio files.

We highlight the audiobook retrieval process to demonstrate how all of the services in this microservice application work in concert to deliver end-user functionality.

When a user requests an audiobook using the Audible app, it first issues a request to the Content Delivery Engine to find the URL of the Content Delivery Service (CDS) that contains the audiobook assets. The app then makes a request to that CDS.

The CDS first makes a request to the Audible Download Service (ADS). The ADS is responsible for first verifying that the user owns the audiobook. Next, the ADS verifies that a Digital Rights Management (DRM) license can be acquired for that audiobook. Finally, it updates statistics at the Stats service before returning a response to the CDS. If ownership of the book cannot be verified or a license cannot be activated, an error response is returned to the ADS, which propagates back to the user through an error in the client.

Once the ADS completes it work, the CDS contacts the Audio Assets service to retrieve the audiobook assets. Then, the CDS contacts the Asset Metadata Service to retrieve associated metadata. Finally, these assets are returned to the user's app for playback.

3.2 Outage

In this section, we look at the specific outage that Audible experienced and identify how both fault prevention and fault tolerance could have been used to prevent the outage.

In the Audible application, there is an assumption that if an audiobook exists in the system and the assets are available, the metadata will also be available. However, this may not always be the case. In fact, this precise fault could be detected through fault prevention that uses library-level fault injection. If used, the developers may choose to either specifically write error handling for this case or ignore the fault under the assumption that this invariant will never be violated.

However, the invariant *was* violated, for reasons that are not disclosed by Audible and are presumably related to operator or database error: the asset may have been deleted accidentally, the database lost the file, or the database could have simply malfunctioned at the time of the request. This resulted in a cascading failure and subsequent outage of the Audible application.

This outage is a result of several, different faults, both latent and active that interacted with one another in some manner. Starting with the lack of error handling, a generic error is propagated back to the mobile application that, upon receiving the generic error, assumes the failure was transient and consequently retries the request a finite number of times. When all of the retries return failure, a generic error is provided back to the user in the mobile application that causes the user to issue a retry. This combination of user-initiated retries for requests that will ultimately fail, paired with the combination of a popular audiobook, is enough to exhaust available compute capacity and cause the application to fail.

Had the developers used a fault tolerance technique, such as a circuit breaker, they could have prevented the system from processing client-initiated retries upon repeated failure. However, use of a circuit breaker in this case requires the proper granularity. For example, a circuit breaker would have to prevent further requests for *that specific audiobook*; a circuit breaker that is too coarse — applied to the request that is used to retrieve *any audiobook* — would effectively induce an application outage while simultaneously trying to prevent one.

This example demonstrates the need for the use of both fault prevention and fault tolerance techniques in an MSA. First, fault prevention is necessary for anticipating the error and alerting the developers to the possible invariant violation. Second, fault tolerance is necessary as the remote service may, by way of malfunctioning return an unexpected error, or if the developers fail to manually specify error handling code under the assumption that the invariant will never be violated. By combined application of these techniques, application developers would have ensured a higher level of resilience for their application.

4 Contribution: Microservice Application Corpus

My work on construction of an MSA corpus was published at the ACM Symposium on Cloud Computing in 2021 [67]. *At the end of this section, I present proposed extensions of this work towards fulfillment of this research proposal.*

The first challenge in addressing the lack of fault prevention and fault tolerance research in MSAs, was to understand the structure of industrial microservice applications and the type of faults that affected them. In other fields

such as software testing and distributed systems, this is typically performed using open source application and bug corpora: unfortunately, this is not available for MSAs.

For example, much of the research into testing DDS for resilience relies on open-source infrastructure projects (*e.g.*, Apache Zookeeper, Apache Cassandra) where all development is performed using a public bug tracker and public software repository that contains all historical revisions of the software and the reasons for change. These resources allow researchers to reproduce previously encountered fully-documented bugs. However, most of these projects are monolithic in design. That is, they are single applications that, while distributed, typically are constructed in a monolithic style and deployed in replica sets. Rather unfortunately, they do not reflect the type of microservice applications being built today: where each service provides its own unique business logic and modularization, for developer productivity, is a core design tenet. As an example, Uber, a ride-sharing service, in 2020, had 2,200 microservices, each providing its own unique functionality. [5]

As another example, most of the research into software testing uses bug databases [50, 46, 55, 56, 90, 51, 81] assembled by the research community — collections of software projects with documented bugs harvested from open-source code repositories. However, these projects suffer from two issues that make them unsuitable for use in microservice testing; (A) they are also monolithic in design, where there is a lack of communication over the network between different components of the system, and (B) and many, if not all, of the bugs contained in the major bug repositories contain bugs that could be identified through traditional software testing using regular unit or functional tests. Said simply — these bugs are not specific to resilience issues in microservice architectures.

It would seem that such corpora do not exist today for microservice applications for (at least) two reasons. First, microservices are typically adopted within organizations to facilitate growth by breaking large applications into distinct services with independent teams. These services are usually core intellectual property of the company and are therefore not open source. Second, companies that experience bugs typically do not publicly disclose the details of the root cause of the fault. While there are some notable exceptions that provide some technical details, there are not enough to replicate in a bug corpus (*e.g.*, AWS Post-Event Summaries, GitHub Blog.) In fact, one employee of a large internet service whom we talked to told us that legally these bugs could not be disclosed for a publicly traded company[5].

4.1 Corpus Creation

To create my corpus, I focused on conference talks at industry events such as Chaos Conf and AWS re:Invent, where it is common for industry practitioners to discuss (and advocate for) the use of chaos engineering. In order to find these talks, I searched for terms such as "chaos engineering" or "resilience", "chaos", or "fault injection". Building upon this, I also identified companies that sold chaos engineering services and looked at the clients that were listed on their web pages. From there, I did a backwards search for these companies to find a presentation or blog post discussing their use of chaos engineering.

In total, I systematically reviewed 50 presentations (representing 32 companies[6]) on chaos engineering. These included technical talks hosted on YouTube and blog posts. This review demonstrated that chaos engineering is used by companies of all sizes, in all sectors, including but not limited to; large tech firms (*e.g.*, Microsoft, Amazon, Google), big box retailers (*e.g.*, Walmart, Target), financial institutions (*e.g.*, JPMC, HSBC), and media and telecommunications companies (*e.g.*, Conde Nast, media dpg, Netflix.)

In most of these presentations, companies had two major concerns; (i) the reliability of software under development, and (ii) the reliability of the cloud infrastructure that the company was running their software on. To create the corpus, I looked for presentations that met any of the following criteria:

- Did the presentations provide detail on a real bug that they discovered using chaos engineering?
- Did the presentation run a chaos engineering experiment that could have been performed locally?

Finally, I ruled out bugs where the bug did not occur in application code, but instead was related to incorrect cloud configuration. I noted several of these examples ranging from incorrect configuration of authorization policies (*c.f.*, AWS IAM) to missing auto-scaling rules (*c.f.*, AWS EC2.)

In the end, I settled on 4 presentations from the following companies: Audible, Expedia, Mailchimp, and Netflix.

- **Audible** [8] is a company that provides an audiobook streaming mobile application. In their presentation, they present the description of a bug where the application server does not expect to receive a `NotFound` error when reading from Amazon S3. This error is unhandled in the code and propagated back to the mobile client with a generic error message. They discovered this bug using chaos engineering.

[5]Private communication with employee of large Internet service, April 2021.

[6]I provide the list to a single representative presentation, each of the 32; in one case, we could find information that chaos engineering was used, but no talk or blog post discussing its use: `https://pastebin.com/qB7gdg45`.

- **Expedia** [11] is a company that provides travel booking. In their presentation, they discuss using chaos engineering to verify that if their application server attempts to retrieve hotel reviews from a service that sorts them based on relevance, and that service is unavailable, that they will fallback to another service that provides chronological sorted reviews.

- **Mailchimp** [14] is a product for e-mail communication management. In their presentation, they discuss two bugs; (A) legacy code that does not handle the case where their database server returns an error code to indicate that it is read-only, and (B) one service becomes unavailable and returns an unhandled error back to the application. Both of these bugs were discovered using chaos engineering.

- **Netflix** [16] is a media streaming product. We reviewed two presentations from Netflix discussing the services involved in loading a Netflix customer's homepage. Netflix doesn't disclose the actual fallback behavior for each service in these talks, but instead alludes to possible fallback behavior. In our implementation, we took some liberties supposing what this behavior is, but kept it realistic.

  In one presentation, Netflix discusses several bugs that they discovered using their chaos engineering infrastructure. These are: (A) *misconfigured timeouts*, where nested service calls aren't configured correctly to allow requests that take longer than expected, but remain within the timeout interval, (B) *fallbacks to the same server*, where services are configured with fallbacks that point back to the failed service, and (C) *critical services with no fallbacks*, where critical services do not have fallbacks configured. We introduced all three of these bugs in our Netflix implementation.

The corpus, at the time of publication, contained 8 small microservice applications, the **cinema examples**, each demonstrating a particular pattern we observed in microservice applications during our survey. It also includes 4 recreations of **industry examples:** Audible, Expedia, Mailchimp, and Netflix.

Each example contains unit tests as well as functional tests that verify functional behavior of the application. Since the functional tests were not discussed in most of the talks, I wrote a functional test that I believe correctly reflects the what the application should do. For the cinema example, I have a single functional test that attempts to retrieve the movie bookings for a particular user.

All of the examples in the corpus are implemented in Python using the Flask web framework [12]. Each example can be run locally in-process, or can be run in Docker containers. Using Docker [10] containers, each example can also be run in any Kubernetes environment (*e.g.*, minikube [15], AWS Elastic Kubernetes Service [7]) as deployment and service configurations are provided for each service.

4.2 Cinema Examples

For each of the cinema examples, I started using a microservice application taken from a tutorial [3] on writing them. This application mimics an online cinema service where users can look up information on the movies that they have bookings for.

It's composed of 4 services: **showtimes**, which returns the show times for movies; **movies**, which returns information for a given movie; **bookings**, which, given a username, returns information about the user's movie bookings; and **users**, which stores user information and orchestrates the request from the end user by first requesting the users's bookings, and for every booking performs a subsequent request to the movies service for information about the movie. The functional test exercises this behavior.

A diagram of the structure of this application and the request path for this functional test in Figure 4. In this diagram, the showtimes service is not contacted, because that service is not involved in this functional test.

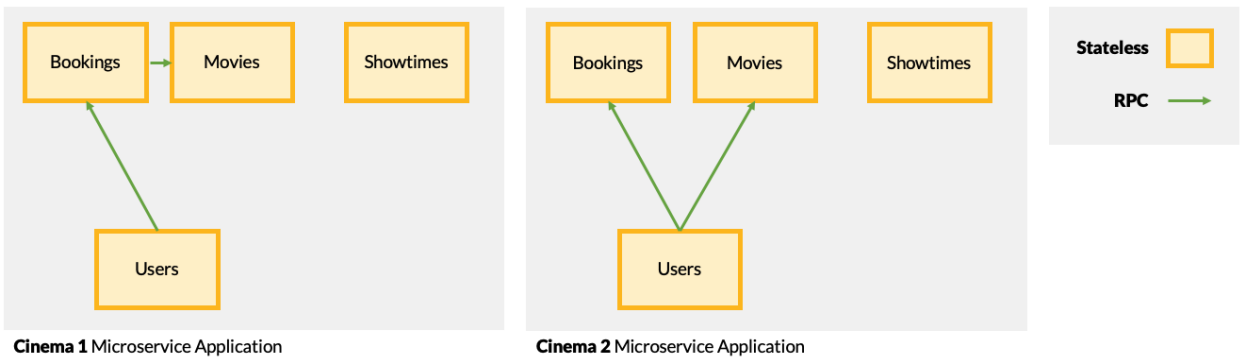


Figure 4: Cinema examples in the MSA application corpus.

In the published version, there are 8 cinema examples. Each demonstrates a different pattern observed in microservice applications. Here, I provide the additional cinema examples, all examples are modifications to `cinema-1`, unless specified.

- **cinema-2**:
  Modifications: bookings talks directly to movies.
- **cinema-3**, derived from **cinema-2**:
  Modifications: users service has a retry loop around its calls to the bookings service.
- **cinema-4**, derived from **cinema-2**:
  Modifications: each service talks to an external service before issuing any requests: the users service makes a request to IMDB; the bookings service makes a request to Fandango; the movies service makes a request to Rotten Tomatoes.
- **cinema-5**, derived from **cinema-1**:
  Modifications: all requests happen regardless of failure; in the event of failure, a hard coded, default, response is used.
- **cinema-6**, derived from **cinema-1**:
  Modifications: adds a second replica of bookings, that is contacted in the event of failure of the primary replica.
- **cinema-7**, derived from **cinema-6**:
  Modifications: the users service makes a call to a health check endpoint on the primary bookings replica before issuing the actual request.
- **cinema-8**:
  Modifications: example is collapsed into monolith where an API server makes requests to the it with a retry loop.

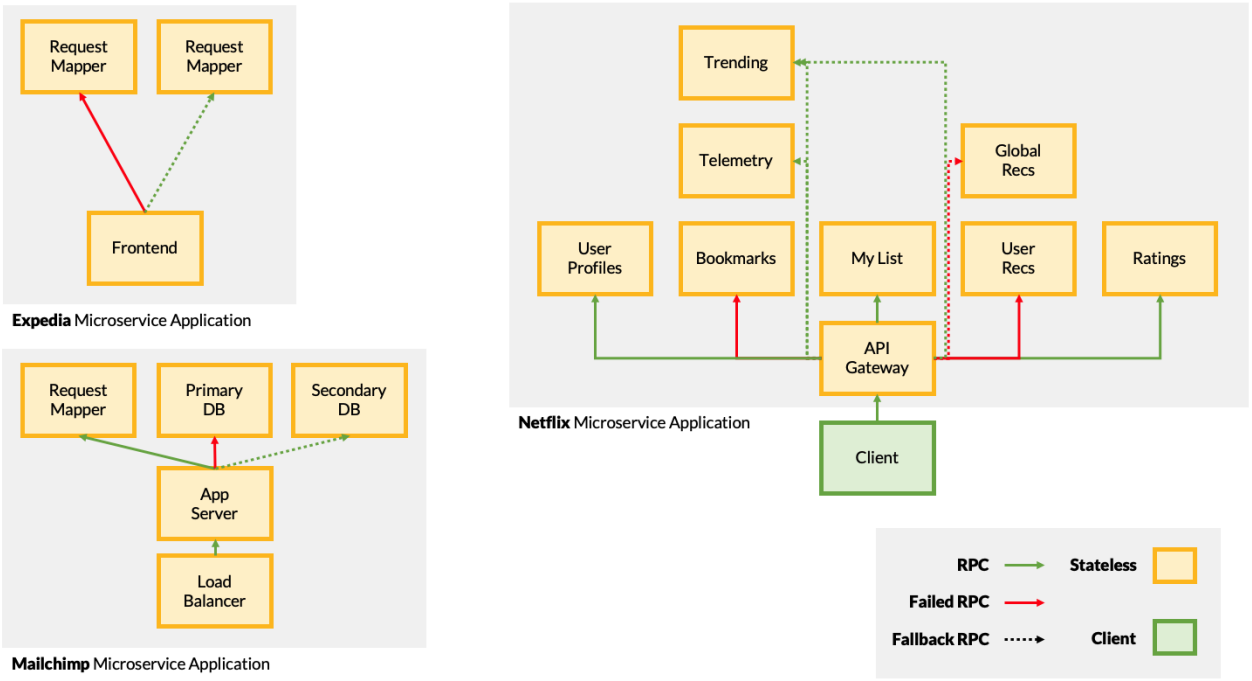


Figure 5: Industry examples in the MSA application corpus.

4.3 Industry Examples

In this section, I provide a description of the four industrial examples that we implemented: Audible, Expedia, Mailchimp, and Netflix. These examples are not meant to reproduce the entire microservice architecture of these companies: I focus only on the services involved in a particular chaos experiment that they performed.

4.3.1 Audible

The Audible example, presented in Figure 3, has 8 services along with a mobile app. To simplify the example, I use a service to stand in for the behavior of the mobile application.

The services in the Audible example are: **Content Delivery Service** (**CDS**), which given a book identifier and a user identifier, return the actual audio content and audio metadata after authorization; **Content Delivery Engine** (**CDE**), which returns the URL of the correct CDS to contact; **Audible Mobile App**, which simulates the mobile application by first, issuing a request to the CDE to find the URL for the appropriate CDS instance to contact based on the book's identifier and then issues a request to it; **Audible Download Service**, which orchestrates logging and DRM authorization once ownership is verified; **Ownership**, which verifies ownership of the book; **Activation**, which activates a DRM license for the user; **Stats**, which maintains book and license activation statistics; **Asset Metadata**, which is the storage for the audio asset metadata which contains information on chapter descriptions; and **Audio Assets**, which is the storage for the audio files.

Compared to Audible's actual deployment, some of the components I represent as services are actually cloud services. I enumerate those differences and adjustments made here. First, the Asset Metadata and Audio Assets services are AWS S3 buckets. To simulate this, I created HTTP services that either returns a `200 OK` containing the asset if available, or a `404 Not Found` if the asset isn't present. Second, the Ownership and Activation services are AWS RDS instances. To simulate this, I created HTTP services that implement a REST pattern: a `403 Forbidden` is returned if the user does not own the book, a `404 Not Found` if the book doesn't exist, otherwise, a `200 OK`. Third, the Stats service is an AWS DynamoDB instance. To simulate this, I created an HTTP service that returns a `200 OK`.

For the functional test, I have a test that attempts to download an audiobook for a user. For the bug, the Asset Metadata service can return a `404 Not Found` response if the chapter information for a book is missing: this is the bug discussed in the Audible presentation and causes a generic error to be presented to the user in the mobile application.

4.3.2 Expedia

The Expedia example, presented in Figure 5, has 3 services: **Review ML**, which returns reviews in relevance order; **Review Time**, which returns reviews in chronological order; and **API Gateway**, which returns reviews to the user from either Review ML or Review Time based on service availability.

The Expedia example has one functional test that loads the information for a hotel from the API gateway. In this example, there isn't a specific bug, but a replication of a chaos experiment that Expedia did actually run.

4.3.3 Mailchimp

The Mailchimp example, presented in Figure 5, has 5 services: **Requestmapper**, which maps pretty URLs in e-mail campaigns to actual resource URLs; **DB Primary**, which is the primary replica of their database; **DB Secondary**, which is the secondary replica of their database; **App Server**, which makes a request to the Requestmapper service to resolve a URL and then perform a read-then-write request to the database, with fallback to secondary database replica is the primary replica is unavailable; and **Load Balancer**, which load balances requests.

Compared to Mailchimp's actual deployment, some of the components that I am representing as services are actually non-HTTP services. I enumerate those differences and adjustments made here. First, the DB Primary and Secondary services are MySQL instances. To simulate this, I created an HTTP service that either returns a `200 OK` on a successful read or write or a `403 Forbidden` if the database is read-only. Second, the Load Balancer service is an HAProxy instance. To simulate this, I created an HTTP proxy.

For the functional test, I attempt to resolve a URL. For the bugs, the Mailchimp example contains two:

- **Bug #1: MySQL instance is read-only.**
  When the MySQL instance is read only, the database returns an error that is unhandled in one area of the code. Since Mailchimp uses PHP, this error is rendered directly into the output of the page and we simulate this by turning the `403 Forbidden` response into output that's directly inserted into the page.

- **Bug #2: Requestmapper is unavailable.**
  When the Requestmapper service is unavailable, the App Server fails to properly handle the error, returning a `500 Internal Server Error` to the Load Balancer. However, the Load Balancer is only configured to handle a `503 Service Unavailable` error by returning a formatted error page. This is an example of missing or incorrect failure handling.

4.3.4 Netflix

The Netflix example, presented in Figure 5, has 10 services. Similar to the Audible example, I simulate the Netflix mobile application with a service, here called **Client**.

The services in the Netflix example are: **Client**, which simulates the mobile client; **API Gateway**, which assembles a user's homepage; **User Profile**, which returns profile information; **Bookmarks**, which returns last viewed locations of movies; **My List**, which returns the list of movies in the user's list; **User Recommendations**, which returns recommendations specific to the user; **Ratings**, which returns the user's movie rating; **Telemetry**, which records telemetry information; **Trending**, which returns trending movies; and **Global Recommendations**, which returns recommendations for all users of the application.

The list of services we implement come from the multiple presentations that I watched from Netflix. However, in their presentations, the fallback behavior that they present is just provided as an example. Therefore, in my implementation, I make a number of decisions on what the fallbacks should be that seemed to reflect possible fallback behavior. I don't believe a specific fallback matters when testing for bugs; but rather we just want to implement a reasonable fallback.

Here are two examples of the fallback behavior that I implement: when Bookmarks are unavailable, load Trending content instead and an log error to Telemetry; and When User Recommendations. are unavailable, load Global Recommendations.

For the functional test, I have a single functional test that attempts to load the Netflix homepage for a user. For the bugs, the Netflix example contains three, that can be activated with an environment variable.

- **Bug #1: Misconfigured timeouts.**
  The User Profile service calls the Telemetry service with a timeout of 10 seconds; however, the API Gateway calls the User Profile service with a 1 second timeout.
- **Bug #2: Fallbacks to the same server.**
  If the My List service is unavailable, the system will retry again.
- **Bug #3: Critical services with no fallbacks.**
  The User Profile service does not have a fallback.

4.4 Proposed Work

I plan to extend this corpus through this research agenda to support the contributions contained within.

First, the examples currently contained in the corpus are implemented in Python, a language that, while contains concurrency primitives, does not support true parallel execution of concurrent processes. With true parallel execution, any successful fault injection approach will need to address the challenges of correct fault injection in the presence of scheduling nondeterminism. Therefore to assess the (A) viability of application of our contributions to MSAs that leverage heterogeneous implementation languages across its services, and (B) to truly concurrent and parallel programs, it is necessary to extend this corpus with additional languages. I have already started this process by implementing several examples in Java and plan to contribute these to the corpus.

Second, the only fault tolerance technique that is currently used by the examples in the corpus are fallbacks. Therefore, it is critical that I extend this corpus with examples that use both circuit breaking and load shedding. This will allow me to first evaluate the problems with circuit breaker and load shedding use before deriving new fault tolerance techniques that address these deficiencies.

Third, and finally, all of the current examples use services to mimic the behavior of stateful services (*e.g.*, DDSs.). Therefore, to show that my fault injection approach also works with stateful services that may be closed source and outside the control of the developer, I plan on implementing additional examples that use stateful DDS.

5 Contribution: Service-level Fault Injection Testing

My work on Service-level Fault Injection (SFIT), a technique for fault prevention in MSAs, was published at the ACM Symposium on Cloud Computing in 2021 [67]. At the end of this section, I present proposed extensions of this work towards fulfillment of this research proposal.

SFIT takes a developer-first approach, integrating fault injection testing into the *development* process as early as possible without requiring developers to write specifications in a specific specification language. This decision is key to adoption, as it seamlessly integrates our approach with developers' existing development environments and tools. The architecture of our SFIT prototype, Filibuster is shown in Figure 6.

SFIT builds on three key observations made about how microservice applications are being developed today:

1. **Microservices developed in isolation.**
   Microservice architectures are typically adopted when teams need to facilitate rapid growth, thereby breaking the team into smaller groups that develop individual services that adhere to a contract. This contract typically

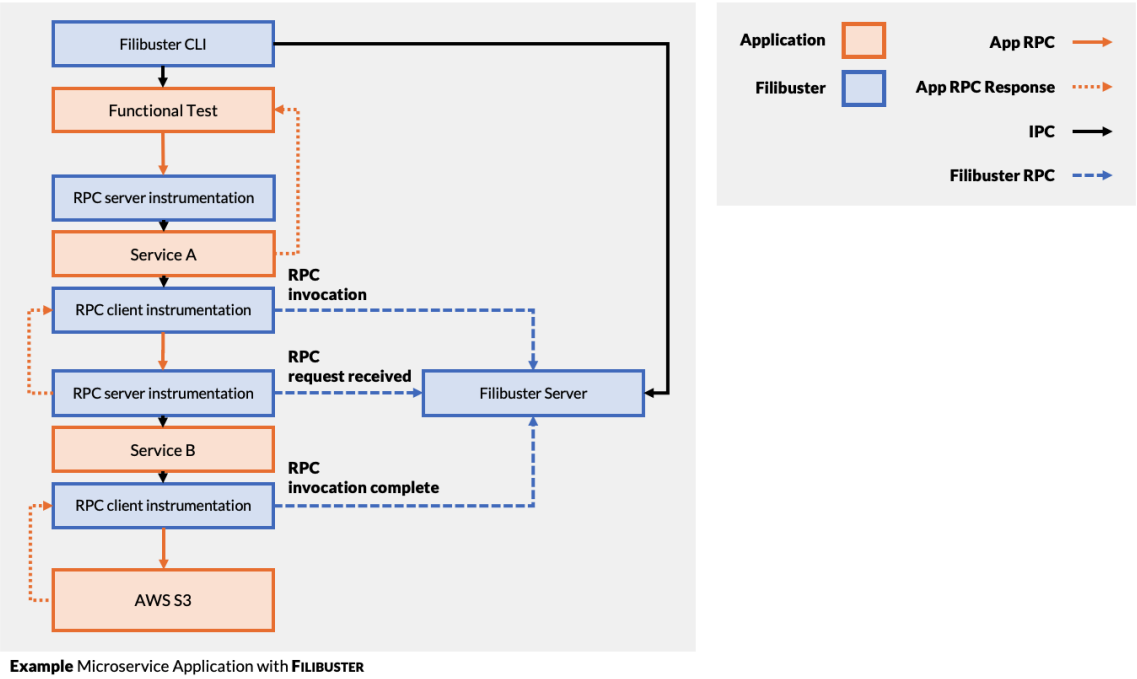


Figure 6: Architecture of FILIBUSTER. Instrumentation calls are made from each service to the test server to identify where remote calls are issued from, where they are received, and to inject faults during test execution.

requires that two or more teams meet and agree to an API between the services that they manage. Therefore, individual team members typically do not understand the state or internals of services outside of their control well enough to write a detailed specification of the application to automatically verify it with a model checker.

2. **Mocking could prevent failures.**
   As can be observed from the applications that I re-implemented as part of our corpus (Section 4), all of the bugs we discovered and discussed could have been detected earlier if the developers had written mocks that simulated the failure of the remote service in a testing environment. I cannot speak to why these tests were not written, but I assume that this might be the case for two reasons; (A) writing tests with mocks is a time consuming process with minimal apparent benefit to the developer as the failure case may be rare, or (B) the failure case is not known to the developer at the time of development.

3. **Functional tests are the gold standard.**
   In lieu of writing specifications, developers write multiple end-to-end functional tests that verify application behavior, when no failures occur, is correct. Therefore, developers already believe that the investment in end-to-end testing is worthwhile, and I believe any successful fault injection approach should start there.

5.1 Approach

SFIT is based on the three key observations about how microservices are being developed today. In this presentation, I make two simplifying assumptions: services communicate over HTTP, which is not a limiting factor of the SFIT design, and that a single functional test exercises all application behavior. In practice, applications will have an entire suite of functional tests to cover all application behavior.

5.1.1 Overview

SFIT assumes a passing functional test, written by the developer, that executes the application under some non-failing scenario and verifies some application behavior. SFIT also assumes that this passing test has already ruled out logical errors.

When running the initial passing execution, at each point where SFIT reaches a location where communication happens with another service, SFIT will schedule another test execution that will re-execute the test and inject a failure for this request. If this request can fail multiple ways, SFIT schedules an execution for each possible failure. These subsequent executions are placed on a stack during execution and this strategy applied recursively until all paths have been explored. This algorithm is inspired by the concolic testing algorithm from DART [53].

In Section 5.1.2, I discuss specifically how fault injection is performed when running the subsequent tests. The stack of test executions to run is maintained by a server process that all services communicate with. This server is responsible for the actual execution of functional tests.

Consider the sample architecture from Audible presented in Figure 3. In this example, the request from our functional test originates at the Audible Mobile App. The first request issued is to the Content Delivery Engine which can fail with a `Timeout` or `ConnectionError`. SFIT adds two executions on the stack of executions to explore and continue executing the test. In Section 5.1.3, I discuss how SFIT determines what errors each call to a remote service can throw or return.

Next, SFIT reaches the Content Delivery Service and schedules the two executions where Content Delivery Engine was successful and the call to Content Delivery Service fails. This is performed for the entirety of the initial request. As SFIT executes all tests in the stack, SFIT may discover new paths by triggering failures. For example, failure of the Content Delivery Engine could cause an additional path to be exposed to a logging service. SFIT continues to explore until all paths have been fully explored.

In this example, several services have multiple dependencies. For example, the Audible Download Service talks to the Ownership service, the Activation service, and the Stats service. In this case, we have to schedule executions that cover the entire space of failures — all of the ways each service can fail independently with all of the combinations of how they can fail with one another.

As the developer runs these generated tests, they will have to adapt their functional test accordingly to consider failure. To do this, SFIT provides a helper module that allows the developer to write conditional assertions when a failure is present. In Section 5.1.4, I discuss precisely how functional tests can be adapted. SFIT also provides a mechanism to replay a counterexample: a single failing generated test.

5.1.2 Fault Injection

The SFIT approach relies on the ability to inject failures for remote calls and therefore it is essential that SFIT can instrument the library used for making remote calls to alter their response.

This ability to interpose on remote calls is already rather commonplace: many popular telemetry systems (*e.g.*, opentelemetry, opentracing) already provide libraries that automatically wrap calls to common libraries used for remote communication (*e.g.*, HTTP, GRPC) in order to assist developers in understanding application performance by sending telemetry information to a remote telemetry service (*e.g.*, jaeger). SFIT leverages this instrumentation design for fault injection: instead of returning the actual response from the remote service, SFIT returns a failure response instead based on the fault that was injected. This instrumentation communicates with a server process that aggregates information collected by the instrumentation in order to determine the next test to run.

5.1.3 Fault Identification

SFIT injects faults that represent the failures that can occur for a given service. This relies on knowing two different types of failures:

- **Failures originating at the call site.**
  SFIT first has to consider faults that can originate at the call site. For example, when using the `requests` library in Python for performing HTTP requests, there are 23 exceptions that the library can raise when performing a request. To address this concern, the developer can either specify the module containing the exceptions or specify them manually in configuration. For this presentation, I will only consider the two most common: `Timeout` and `ConnectionError`.

- **Failures originating at the remote service.**
  A service might handle a failure of one of its dependencies and return a failure. For example, if a service that is a dependency of another service throws a `Timeout` exception, it may be caught and a `500` returned. SFIT uses a static analysis on the service's source code to over-approximate the responses that the service can return. In Flask, this is possible using looking for `return` or `raise` statements.

One of the difficulties with HTTP is that requests made between different services use a URL provided as a string. This string may not be a unique identifier of the actual service that is being contacted, as these URLs may use IP addresses or unrelated DNS names. To solve this, SFIT uses additional instrumentation to record the service that is actually reached when a call is made. This instrumentation, instead of being used on the caller's library used for remote communication, is placed on the web framework that receives the request. Therefore, the instrumentation can record the callee's service name before the request is processed by the application code. Similar to the instrumentation that SFIT uses on the caller's side, SFIT leverages the same design as the common telemetry systems (*e.g.*, opentelemetry) take and transmit this information to the server to determine the next test to execute.

5.1.4 Test Adaptation

As developers will be starting with a functional test that assumes no failures, developers will need to update their functional test to contain proper test oracles for the cases where dependent services fail.

To do this, SFIT provides a helper module for writing conditional assertions. This helper lets the developer write a conditional statement such as *if a fault was injected on Service A* and place appropriate assertions on what the behavior of the system should be under failure. Developers will add these conditional assertions into the existing functional test. This is a much less intrusive process when compared to the alternative: manually duplicating passing functional tests, using mocks to simulate failure, each with custom assertions about application behavior under failure. For a similar reason, SFIT avoids static test generation and favor a dynamic approach where large numbers of tests do not need to be consistently regenerated during software development.

A typical development workflow using SFIT is imagined as follows. First, developers start with a passing functional test and SFIT begins injecting faults. As faults are injected, the functional test will fail with assertion errors. Using the assertion helper provided by SFIT, developers will write an conditional assertion to capture desired failure behavior. An example of one such assertion for the Audible application might say *"if a fault was injected on the stats service, the service should still play the audiobook."* From there, the developer can use counterexample replay the previous failing test to validate these newly added assertions.

5.2 Proposed Work

SFIT, as published, has a number of limitations that we plan to overcome in this research proposal.

The prototype of SFIT, Filibuster, is currently limited to microservices implemented in Python that communicate using RPC over HTTP. Therefore, the first step in validating the practical applicability of SFIT is to extend Filibuster with different RPC implementations. I plan to extend Filibuster with support for Google's GRPC to validate that the SFIT design is RPC framework agnostic.

Similarly, one of the defining characteristics of industrial MSAs is the use of multiple implementation languages. Therefore, Filibuster must also be extended to support different programming languages as well: I plan to extend Filibuster with support for Java to validate that the SFIT design is programming language agnostic.

When considering the extension of Filibuster to different programming languages, the choice of Java is not random selection. As Filibuster currently tests Python microservices, it rarely has to deal with the complexities introduced by true concurrency: while Python does provide some concurrency primitives, Python's Global Interpreter Lock prevents parallel execution of concurrent code.

Finally, industrial MSAs can contain hundreds of services. Therefore, systematic, exhaustive exploration of the fault space may be prohibitive in practice. In order for SFIT to remain a practical technique at scale, some form of test case reduction is necessary. I address this by first creating a novel indexing algorithm for RPCs in MSAs (§6) and use this indexing algorithm to drive a test case reduction strategy (§7).

6 Contribution: Distributed Execution Indexing

My work on Distributed Execution Indexing (DEI) is a work-in-progress. More specifically, the synchronous variant of distributed execution indexing was both implemented and evaluated as part of our SFIT publication at the ACM Symposium on Cloud Computing (SoCC) in 2021 [67]. Preliminary work on an asynchronous variant of DEI, including the full formal definition of synchronous DEI that was omitted from our publication at (SoCC), was submitted to the USENIX Annual Technical Conference in 2022, but was rejected. At the end of this section, I present the preliminary work on the asynchronous variant of DEI, towards fulfillment of this research proposal.

Fundamental to the SFIT approach is the need to uniquely (and deterministically) identify each RPC. For example, in Figure 6, specific identification of the RPC issued between Service A and Service B. In order for SFIT to know when the systematic search of the fault space is complete, this specific RPC — between Service A and Service B — must be identified the same across all test executions. While this may seem rather trivial, as it only requires the identification of a single edge in a microservice graph, it becomes more complex when multiple RPCs between the same pairs of services may exist in the same test execution. The programming patterns that cause this behavior are rather commonplace: loops, branching, function indirection, and concurrency.

6.1 Signatures Are Too Coarse-Grained

Consider one simple way of identifying RPCs, used by other fault injection approaches, the RPC's signature. We formally define an RPC signature as follows:

Definition 6.1. A *signature* is a triple (m, f, a) where

```
1  @service_a.method("helloworld")
2  def service_a_helloworld():
3      hello = echo("Hello")
4      world = echo("World")
5      s = hello + " " + world
6      return s
7
8  def echo(s : String):
9      try:
10         res = rpc(service_b, "echo", s)
11         log_success(res)
12         return res
13     except Exception as e:
14         log_error(e)
15         return s
16
17 @service_b.method("echo")
18 def service_b_echo(s : String):
19     return s
```

Figure 7: RPC signature alone cannot distinguish between the RPCs issued on lines 3 and 4; call stack or invocation count must be combined with signature.

- m is the module or class name of the RPC stub;
- f is the method or function name; and
- a is the parameter names and types.

This technique is agnostic to the RPC framework, and in fact can be easily used to represent both of the two most common: HTTP and GRPC. With gRPC, the class name and method map directly; parameters are the parameter types and names for the gRPC endpoint. With HTTP, the URI and HTTP method can be combined to form the signature as it contains the target service, method name, and parameter names and types, which are assumed to be `String`. Let us see how the RPC signature is too coarse-grained to uniquely (and deterministically) identify an RPC.

Consider the example in Figure 7. In this example, a microservice application composed of 2 services written in pseudocode. Service A exposes a single RPC endpoint, `helloworld`, which issues two RPCs to B's RPC endpoint, `echo`, before combining the responses and returning a response. In the event that Service B is down, a default response is returned by the function wrapping the RPC, `echo`, on line 8.

In the case of the RPC invocation at line 10, the signature, would be composed of the target service name B, the method `echo`, and the parameter `(s,String)`. In this application, the signature for both of the RPCs invoked by Service A, on lines 3 and 4, would be identical: (`B`, `echo`, `(s,String)`). SFIT would not be able to distinguish between the first and second RPCs for systematic fault injection; that is, the RPC signature alone is too *coarse-grained* for identifying a particular RPC.

6.2 Increasing Granularity: Invocation Count or Call Stack

One solution for resolving the issue where identical identifiers are assigned to different RPCs is to increase the granularity of the identifiers that we assign. Here, I examine two different ways that this could be accomplished and demonstrate that they must be used together. In the following discussion, since the presentation goes beyond just signature-based identifiers, I assume (for ease of presentation and without loss of generality) that a service (say `A`)) makes RPC invocations to only one other service (say `B`) and only a single RPC endpoint (*e.g.* `echo`) per service. Thus, I use only the *invoking service name* (*e.g.*, `A`) as a shorthand for an outgoing RPC from `A` that stands in for the full signature which would contain the target service, method name, and parameters.

1. *Invocation count.*
   3MileBeach [87] keeps track of the number of invocations for each RPC call site in order to distinguish multiple calls to the same call site. In Figure 7, the same RPC is invoked twice. The symbol "|" is used to indicate the invocation count of an RPC signature. For example, the identifiers $A|_1$ and $A|_2$ distinguish the 1st and 2nd RPC invocations made from service `A` at lines 10.

2. *Call stack.*
   Another approach is to increase the granularity of the identifier with some representation of the call stack. In Figure 7, the RPC is invoked twice at line 10, however, with different calling contexts for the `echo` function (lines 3 and 4). A *superscript* is used to indicate the line number(s) corresponding to call stack at the time of

```python
@service_a.method("helloworld")
def service_a_helloworld(ss : List[String]):
    rs = []
    failure = False

    for s in ss:
        try:
            r = rpc(service_b, "echo", s)
            rs.append(r)
        except Exception as e:
            failure = True
            break

    if failure:
        s = "Hello World"
        r = rpc(service_b, "echo", s)
        return r
    else:
        return rs.join(" ")
```

Figure 8: Signature combined with invocation count insufficient in distinguishing 2nd iteration of loop from 1st invocation of failure handler; signature combined with call stack insufficient in distinguishing loop iterations.

invocation. For example, the two RPC invocations in Figure 7 can be distinguished by identifiers $A^{3,10}$ and $A^{4,10}$.

For the example in Figure 7, either invocation count or call-stack based identification works to disambiguate the two RPCs. However, neither approach is sufficient on its own in general. A better approach is to use a *combination* of invocation count *and* calling contexts for identifying RPCs, e.g. $A^{3,10}|_1$, denoting the first invocation of RPC from A with the calling context $(3, 10)$.

To demonstrate the need for both these terms, the reader is referred to Figure 8. In Figure 8, the reader is presented with a different implementation for A; it is assumed the same implementation of B from Figure 7. In this example, A's RPC endpoint `helloworld` takes, as parameters, a list of `String`. For each `String` that is provided, a RPC is invoked to B's `echo` endpoint. In the event that the RPC to B throws an exception, the remainder of the list traversal is aborted and a final RPC is made to B using a default value and that value returned by A. When no exceptions are thrown, the aggregated results are joined and returned by A.

Consider a functional test that invokes `helloworld` with a list containing two `Strings`. For simplicity, it is assumed that each RPC can only throw a single runtime exception. Therefore, SFIT must run 5 different executions of the test to fully exhaust the fault space. First, consider the execution where both loop iterations execute and all RPCs are successful, which we denote as a sequence of RPC invocations: $e_1 : (A^8|_1, A^8|_2)$. Next, consider the executions where the RPC throws an exception, using the $\neg$ symbol to denote a failed RPC invocation. When a fault is injected in the 2nd iteration of the loop, there are two cases when the fallback RPC either completes successfully or fails: $e_2 : (A^8|_1, \neg A^8|_2, A^{16}|_1)$, $e_3 : (A^8|_1, \neg A^8|_2, \neg A^{16}|_1)$. Finally, consider the executions where the RPC throws in the 1st iteration and the fallback RPC either completes successfully or fails: $e_4 : (\neg A^8|_1, A^{16}|_1)$, $e_5 : (\neg A^8|_1, \neg A^{16}|_1)$.

Using this example and these test executions, I now examine why invocation count and call stack are, by themselves and in combination with the signature, insufficient for ensuring correctness based on our criteria. Therefore, they must be combined.

- **Invocation Count Alone is Insufficient.**
  Consider executions e_1 and e_4. Using *only invocation count* these executions would instead be represented as $e_1 : (A|_1, A|_2)$ and $e_4 : (\neg A|_1, A|_2)$. However, $A|_2$ in e_1 refers to the invocation at line 8 and $A|_2$ in e_4 refers to the invocation at line 16. Therefore, to properly assign identifiers to these RPCs, the granularity must be increased to include the call stack that resulted in the RPC invocation.

- **Call Stack Alone is Insufficient.**
  In e_1, both requests would be assigned the same identifier: $e_1 : (A^8, A^8)$. Therefore, to properly assign identifiers to these RPCs, the granularity must be increased to include the number of times each RPC invocation statement is reached.

We can use the combination of RPC signature and the calling context to create a dynamic *invocation signature*. This allows us to handle both looping constructs and conditional control flow, as presented in Figure 8. We define this as follows:

Definition 6.2. The *invocation signature* for an RPC invocation is a triple (s, t), usually denoted as s^t, where:

```
1  @service_a.method("helloworld")
2  def service_a_helloworld(ss : List[String]):
3      rs = []
4      failure = False
5      failures = []
6
7      for s in ss:
8          try:
9              r = rpc(service_b, "echo", s)
10             rs.append(r)
11         except Exception as e:
12             failure = True
13             failures.append(len(rs) - 1, s)
14             rs.append("")
15
16     if failure:
17         for (i, s) in failures:
18             try:
19                 r = rpc(service_b, "echo", s)
20                 rs[i] = r
21             except Exception as e:
22                 pass
23
24     return rs.join(" ")
25
26 @service_b.method("echo")
27 def service_b_decorate_echo(s : String):
28     try:
29         r = rpc(service_c, "echo", s)
30         return r
31     except Exception as e:
32         return s
33
34 @service_c.method("echo")
35 def service_c_echo(s : String):
36     return s
```

Figure 9: RPC signature, when extended with invocation count and call stack, is insufficient when RPC invocation is triggered by different incoming RPC requests.

- s is the signature of the RPC;
- t is a representation of the call stack of the RPC.

Thus, the notation $s^t|_k$ refers to the k-th invocation of an RPC with invocation signature s^t. An important point to note is that while RPC signatures (Definition 6.1) can be statically determined, the invocation signatures (Definition 6.2) are determined only based on observed executions.

6.3 Increasing Granularity: Path to Currently Invoked RPC

In Figure 9, another variation of the `helloworld` microservice application is presented. Similar to Figure 8, Service A receives a list of `Strings`, invokes an RPC on Service B for each member in the list, and accumulates the result. In the event of an exception, a placeholder value is accumulated and the failure is recorded. The recorded failures are then iterated in a retry loop and, if successful, the value replaces the placeholder. Different from Figure 8, Service B invokes an RPC on a third service, Service C, and decorates the response somehow before returning a response to Service A.

The same functional test is assumed. However, for brevity, the parameter name `s` in the invocation signatures is omitted.

Using the technique from the previous section, the execution where the list iteration completes and no faults are injected is: $e_1 : (A^9|_1, B^{29}|_2, A^9|_1, B^{29}|_2)$. For each iteration, A issues an RPC from line 9 to B; when B receives the RPC from A, it issues an RPC to C from line 29. Recall from the previous section that the entire call stack of the application is encoded; in this example, each service only contains a single method, and therefore the call stack only includes a single line number. When looping or other conditional control flow is used, inclusion of the invocation count for each call site captures each loop iteration.

Now, consider the functional test execution where a fault is injected on the RPC in the 2$^{\text{nd}}$ iteration of the loop. This execution looks like the following: $e_2 : (A^9|_1, B^{29}|_1, \neg A^9|_2, A^{19}|_1, B^{29}|_2)$. As before, during the 1$^{\text{st}}$ iteration of the loop, Service A issues an RPC to Service B at line 9; Service B then issues an RPC to Service C at line 29. When

the 2nd iteration of the loop is reached, a fault is injected for the RPC from Service A to Service B. Then, the failure condition is met and a subsequent RPC is issued from Service A to Service B on line 19; Service B then issues an RPC to Service C on line 29 before returning a response.

The issue experienced in this example is that the RPC identified by $B^{29}|_1$ in test execution e_1 is *not* the same as the RPC identified by $B^{29}|_1$ in test execution e_2. In execution e_1, the RPC from Service B to Service C at line 29 is caused by the RPC issued by Service A on line 9. In execution e_2, the RPC from Service B to Service C at line 29 is caused by the RPC issued by Service A on line 19. These are not the same, even though they issue the same RPC with the same arguments and payload. They represent distinct call sites in different parts of the code: one is part of the normal operation of the RPC endpoint where no failure occurs, and one represents error handling code that needs to be tested to ensure correct operation of the application under failure. Therefore, associating the same identifier to these RPCs results in both unsound and incomplete behavior: either, the injection of faults on the incorrect RPC or the failure to explore the fault space during exhaustive search.

To resolve this issue, the path of RPC invocations that resulted in the current RPC must be included, as this information is not captured by the call stack. To achieve this, a list of identifiers is accumulated as RPCs are invoked from service to service as part of handling a received RPC invocation: for example, $[A^9|_1 :: B^{29}|_1])$ to indicate that the 1st invocation of invocation signature B^{29} occurred as a result of the 1st invocation of invocation signature A^9.

We can reformulate test executions e_1 and e_2 as follows:

- $e_1 : ([A^9|_1], [A^9|_1 :: B^{29}|_1], [A^9|_2], [A^9|_2 :: B^{29}|_2])$
  The RPC invocations from A to B on line 9 are denoted with the prefixes $A^9|_1$ and $A^9|_2$ to include the enclosing RPC from A.
- $e_2 : ([A^9|_1], [A^9|_1 :: B^{29}|_1], [\neg A^9|_2], [A^{19}|_1], [A^{19}|_1 :: B^{29}|_2])$
  The RPC invocation from B to C on line 29 is prefixed by $A^{19}|_1$ which distinguishes it from the 2nd RPC in execution e_1 from A to B on line 9 that triggered the RPC from B to C on line 29.

Definition 6.3. The *distributed execution index* (DEI) for an RPC invocation is a sequence $[r_1|_{c_1} :: r_2|_{c_2} :: \cdots :: r_n|_{c_n}]$ where:

- r_n is the invocation signature of the current RPC invocation; and,
- the current RPC invocation is the c_n-th invocation of r_n with the path having DEI $[r_1|_{c_1} :: r_2|_{c_2} :: \cdots :: r_{n-1}|_{c_{n-1}}]$.

The definition of a DEI is thus recursive, with the base case being the top-level entry point to the application, whose path is the empty sequence $[\,]$.

This variant of distributed execution indexes enables a test case reduction strategy for SFIT, called Dynamic Reduction (SFIT-DR) (§7). However, this variant only applied to *synchronous* RPC, and does not support application code that uses concurrency.

6.4 Proposed Work

In order to address the lack of support for asynchronous RPC in application code, common to industrial MSAs, I present preliminary work on an asynchronous variant of DEI below.

In the previous discussion, we refer the reader to our definition of invocation signatures (Definition 6.2). Recall from our discussion that call stack, and invocation count were enough to distinguish RPC invocations in the presence of loops and function indirection. These are however, not enough to distinguish RPCs in the presence of concurrency and the resulting scheduling nondeterminism from the use of concurrency.

For example, consider Figure 10, a modified version of Figure 8, where line 7 invokes an RPC using the `async` primitive and the results are `awaited` on line 11. In this example, the invoked RPCs execute concurrently and both their execution order is susceptible to *scheduling nondeterminism*.

Similar to before, a functional test is assumed that invokes the `helloworld` RPC endpoint with two `Strings`. For example, the first test execution should read as follows: $e_1 : (A^7|_1, A^7|_2)$: $A^7|_1$ is the RPC invoked in the 1st iteration of the loop, where $A^7|_2$ is the RPC invoked in the 2nd iteration of the loop. However, on repeated execution of this test through deterministic replay, or when performing exhaustive search, scheduling nondeterminism may result in the 2nd iteration of the loop being assigned $A^7|_1$, if the 2nd block happens to execute first.

Model checkers for distributed systems [85, 62, 64] also face the problem of scheduling nondeterminism. However, these model checkers were originally designed for identifying concurrency bugs — before later being extended for failure testing (*e.g.*, message omission) and therefore rely on control of the thread scheduler. This is an unrealistic for large, microservice applications where (A) they may not be able to run all services on a single machine during testing, and where (B) services are implemented in a number of different languages. Therefore, ideally a solution to this problem will not require control of the thread scheduler.

There are three different ways that this could be accomplished: unfortunately, none are sufficient.

```
@service_a.method("helloworld")
def service_a_helloworld(ss : List[String]):
    rs = []

    for s in ss:
        r = async {
            return rpc(service_b, "echo", s)
        }
        rs.append(r)

    awaitAll rs
    return rs.join(" ")
```

Figure 10: Scheduling nondeterminism can permute assignment of identifiers. In this case, $A^7|_1$, can refer to the RPC invocation from either the 1st or 2nd loop iteration.

1. *Cloning per block.*
   One approach is to *clone* the state that is used to generate identifiers for each asynchronous block. This would ensure that each block would count invocations for each RPC signature, and associated call stack, independently. However, this approach does not work. In Figure 10, this technique would result in identical identifiers for each of the RPCs executed during the loop: $(A^7|_1, A^7|_1)$.

2. *Encode thread creation.*
   DeadlockFuzzer [60], a system for detecting deadlocks in concurrent programs using execution indexes, takes an alternative approach where thread creation is included in the identifier. This approach does not work in the case of asynchronous blocks, as they may execute on an existing thread pool provided by the system or framework where the threads have already been created.

3. *Cloning per thread.*
   If we were to follow this line of thinking, we could also *clone* the state that is used to generate the identifiers for each thread. This does not work either. In Figure 10, scheduling nondeterminism may cause two of the RPCs to execute on a single thread in one execution $(A^7|_1, A^7|_2)$ and on two different threads in a subsequent execution: $(A^7|_1, A^7|_1)$.

The approach that seems most practical stems from a *key observation* about microservice applications: while these applications may issue concurrent RPCs with the same signature, these concurrent RPCs will rarely contain the same payload: the precise argument values supplied at invocation time. Therefore, the *key insight* is that, through the inclusion of the payload in each RPC's identifier, identifiers will be assigned deterministically without requiring control of thread creation or the thread scheduler. To achieve this, the stated that is used to derive identifiers is shared across all threads that are used to execute concurrent code by reference.

This is referred to as the *invocation payload*.

Definition 6.4. The *invocation payload* p for an RPC with n parameters is a sequence $(k_1, v_1)(k_2, v_2)...(k_n, v_n)$ such that for each i in $[1, n]$, the term k_i is the i-th argument's name and v_i is the i-th argument's value.

For gRPC, these are the precise argument values at invocation time. For HTTP, these are the combination of query-string arguments and request body.

In Figure 10, and assuming the concrete argument provided to the function is the list `["Hello", "World"]`, the execution can be represented as follows: $e_1 : (A(\texttt{(s,Hello)})^7|_1,\ A(\texttt{(s,World)})^7|_1)$. It is important to note that the invocation count in both of these identifiers is 1, as it considers both the call stack and payload together. This ensures deterministic assignment regardless of scheduling nondeterminism.

Using can use the combination of RPC signature, the calling context, and the invocation payload the *invocation signature* for an RPC can be redefined as follows:

Definition 6.5. The *invocation signature* for an RPC invocation is a triple (s, p, t), usually denoted as $s(p)^t$, where:

- s is the signature of the RPC;
- p is the invocation payload of the RPC; and
- t is a representation of the call stack of the RPC.

Thus, the notation $s(p)^t|_k$ refers to the k-th invocation of an RPC with invocation signature $s(p)^t$. While this presentation has been framed using `async/await`, many other concurrency primitives (*e.g.*, futures, coroutines) exist that have the same challenges: this technique extends to all of them.

This proposed extension assumes that a key observation holds true: MSAs do not issue concurrent RPCs from the same call site, with the same calling context, to the same service, containing the same payload. In this research proposal, we have started to empirically verify this and plan to evaluate our solution on a real-world, industrial MSA to validate our approach.

7 Contribution: Dynamic Reduction

My work on Dynamic Reduction, using the synchronous variant of Distributed Execution Indexing, was published at the ACM Symposium on Cloud Computing (SoCC) in 2021 [67].

In order to identify corner case bugs, SFIT must ideally explore *combinations* of service failures. To achieve maximum coverage of the failure space — for a single functional test, where service responses are deterministic and there are no data dependencies on previous failures — the number of test executions that are required is quite large.

A straightforward approach of injecting failures in each combination of intra-service RPCs requires executing tests in a magnitude that is *exponential in the number of service requests*. However, by leveraging the decomposition of an application into independent services, it is possible to dramatically reduce the search space without loss of completeness.

To explain, revisit the Audible example that is presented in Figure 3. Excluding the complete failure space for readability, the reader should consider just the failures of a subset of the services: Audible Download Service (ADS) and its dependencies and Content Delivery Service (CDS) and its dependencies.

When exploring failures of the ADS, SFIT must consider the failure of its 3 dependencies: the Ownership, the Activation, and the Stats services. If either of the Ownership or Activation service calls fail, the entire request is failed. However, if the call to the Stats service fails, that failure has no impact on the result of the request. After testing, SFIT will know that any failure of the Ownership or Activation service will cause the ADS to return a `500`. However, a failure of the Stats service will not impact the response of the ADS; regardless of its failure, the service will return `200` as long as both Ownership and Activation provide a successful response.

This property, where the failure of a transitive dependency is encapsulated behind the responses of its direct dependencies, is called *service encapsulation*. Service encapsulation leverages a core design tenet of microservices where databases and other storage is *not* shared between services; therefore, the only visibility a service has into the operation or state of another service is through its API when communicating with it directly using RPC.

With the CDS, at a minimum SFIT have to consider the failure of the Asset Metadata service independently, the failure of the Audio Assets independently, and then the combinations of the ways each service can fail together. However, in order to fully explore the failure space, SFIT must consider the failure of the Stats service combined with all possible failures of the Asset Metadata service and the Audio Asset service. These are failures that SFIT already knows the impact (and outcome) of, and therefore should not have directly test.

For example, SFIT has the knowledge, through direct observation and testing, of:

- a failure of the Stats service has no impact on the ADS; and
- the impact of any combination of failures of the Asset Metadata and Audio Asset services.

It is critical then for SFIT to leverage this knowledge in order to reduce the number of required test cases to exhaust the failure space. To do this, SFIT takes advantage of the following 3 key observations:

1. First, SFIT must fully explore all of the ways a service's dependencies can fail. This ensures that SFIT observes the behavior of a single service when one or more of its dependencies fail and what the resulting failures returned by that service are. With the Audible example in Figure 3, SFIT must fully explore the combination of the ways that ADS's dependencies can fail (as well as the way the CDS's dependencies can fail, etc.)

2. Second, if SFIT is about to inject faults on at least *one* dependency of *two or more* different services, SFIT will already have observed the impact that those failures will have on the services who takes them as dependencies. With the Audible example in Figure 3, SFIT already has observed what the ADS will return when it's dependencies fail in any possible combination. SFIT has already observed what the CDS will return when it's dependencies fail in any possible combination for the same reason. Therefore, SFIT does not have to inject the fault at the dependencies; it can inject the appropriate response directly at the ADS or CDS directly.

3. Third, if SFIT has already injected that fault at that service, then the test is redundant, as SFIT has already observed that behavior of the application. With the Audible example in Figure 3, SFIT does not need to test the Stats service failing in combination with failures of the Audio Assets or Audio Metadata services, as it already has observed the outcome of those failures on the services that take them as dependencies; SFIT has also already observed those outcomes.

Algorithm 1: Dynamic Reduction

```
t: a test to run containing faults to inject
pts: the list of tests already run
Function ShouldReduce(t, pts):
    all_found = True
    for each service and it's dependencies in t
    for (s, d) in deps (t):
        found = False
        find a previous execution
        for pt ∈ pts:
            where the outcomes match for all deps
            if d ∈ deps (pt):
                found = True
        if not found:
            all_found = False
    if all deps are found, t can be skipped.
    return all_found
```

Algorithm 1 presents the dynamic reduction algorithm. This algorithm reduces the exponent in the size of the test execution space from the total number of service requests to *the maximum number of outgoing requests from any given service.* In Figure 3, this reduces the exponent from 8 (the total number of edges) to 3 (the maximum branching factor.) Since microservice applications typically scale in depth rather than breadth, dynamic reduction makes SFIT tractable. Dynamic reduction is automatic and requires no additional information from the application developer. It is important to note that dynamic reduction is not sound in general, and refer to the aforementioned assumptions on the behavior of a single functional test: service responses are deterministic for a single functional test and that service code does not contain data dependencies on previous failures.

7.1 Proposed Work

Towards my research agenda, I plan to expand the evaluation of dynamic reduction to programs with concurrency once the work on asynchronous variant of distributed execution indexes (§6) is completed.

8 Contribution: Study of Fault Tolerance Techniques

This section presents work currently in submission to the ACM Symposium on Cloud Computing 2022.

No fault prevention technique is sufficient for preventing all application outages that result from faults. Therefore, developers typically rely on the use of various fault tolerance techniques to augment fault prevention. These techniques are: fallbacks, circuit breakers, and load shedding.

From a certain perspective, circuit breakers and load shedding are duals: circuit breakers prohibiting an RPC call when the invoked service is under duress; load shedding avoiding processing a received RPC invocation when the service is under duress. In fact, previous academic work has posited the existence of server-side circuit breaking; this can be seen as a specific instance of load shedding, where instead of limiting RPC invocations based on oustanding requests, invocations are limited based on observed error responses that flow back to the invoker.

Under this assumption, I examine the state-of-the-art in circuit breaker implementations and propose how, in many cases and depending on application design, the current designs and implementations of circuit breakers are not sufficient to contain common faults that can occur in MSAs.

We begin with a review of circuit breaker technology.

8.1 Circuit Breakers

Generally speaking, circuit breakers interpose on the RPCs that are issued between two different services. They observe the responses from each RPC issued by the service and accumulate counters for various metrics such as response time, number of errors received, and number of outstanding requests using a sliding window.

Circuit breakers start in the *closed* state where RPCs are issued as normally. If response times or error counts exceed a threshold within this window, the circuit moves into an *open* state where RPCs are short-circuited and a predetermined error response is returned to the application to indicate that the circuit is open. From there, circuit

breakers eventually, dependent on their configuration, move into the *half-open* state where some requests are allowed through in order to determine if the circuit should move back into a closed state: this is the circuit breaker's initial state. From there, any subsequent failure moves it back into the open state.

Implementations may provide the ability [35, 34] to perform this process of determining whether or not a circuit should move back into the closed state asynchronously using an endpoint on the remote service: this is referred to as a health check. These health check endpoints may be provided by a cluster manager (*e.g.*, Kubernetes [28]) or by the application itself, if the health of the service is based on application logic or the availability of dependent services or data stores. Therefore, they return either a success or error response based on whether or not the service is able to accept more requests using cluster state, internal state to the service, or other custom application metrics.

Circuit breakers may be implemented in either the application- or in the infrastructure-layer.

8.1.1 Infrastructure-Level

Some organizations opt for infrastructure based implementations of circuit breakers. This design choice has the benefit of allowing the organization to bundle the circuit breaker into the container or underlying infrastructure of a given service, eliminating the issue where developers may forget to install circuit breakers in their code, risking the overall reliability of the application with respect to cascading failures. These implementations represent the most recent developments in circuit breaker design.

One example of such a circuit breaker is Envoy's adaptive concurrency control filter. [24] With Envoy, circuits in an open state can do one of either of two things. First, the error code returned by Envoy to the application can be used in conditional logic to drive different application behavior when the circuit is in an open state. Second, eviction of a malfunctioning node from a load balancing pool can be driven by a circuit in the open state, if failures and excessive response times happen to be specific to one node.

8.1.2 Application-Level

Application-level circuit breakers can be used at different abstraction levels depending on both desired performance and code reuse.

From an implementation perspective, application-level circuit breakers typically sit between the application, which issues the RPC, and the underlying communications library. Often, the communication libraries that circuit breakers build on provide interposition mechanisms (*e.g.*, GRPC interceptors [17]) that ease implementation or integration. [18] In the application itself, some circuit breakers provide the application developer with the ability to automatically retry failed requests or specify alternate application logic that should execute when the circuit is in an open state: the latter is commonly referred to as fallback logic. [34, 18, 35]

In terms of circuit breaker usage, they may be used *explicit*ly or *transparent*ly. Transparent usage does not require developers to encode circuit breakers themselves, but are typically installed through some manner of automatic instrumentation. Explicit usage requires that developers manually install them in application code. Circuit breakers may be installed at the *callsite*, the encapsulating *method*, or on the *client* that is used to issue the the RPC. Typically, circuit breaker libraries can always be used explicitly at each callsite that invokes an RPC; often, the developers have provided decorators in order to reduce developer overhead. This makes sense: decorators can be thought of as macro application that explicitly encodes the circuit breaker usage at the desired location.

8.1.3 Categorization of Application-level Circuit Breakers

I propose the following categories based on the existing open source circuit breaker implementations.

1. **Callsite-explicit.**
   A *callsite-explicit* circuit breaker is installed directly at an RPC invocation site and wraps the invocation using a conditional. This conditional reads the state of the circuit breaker to determine whether or not to actually invoke the RPC. Developers are responsible for incrementing both success and failure counters that are used by the conditional's predicate. For example, Ameria's `CircuitBreaker` [23], App-vNext's Polly for .NET [25], Akka's `CircuitBreaker` [22], Comcasts's `jrugged` [26], `circuitbreaker` for Go [33], `pybreaker` [27] can all be used explicitly by the developer at each callsite.

2. **Method-explicit.**
   A *method-explicit* circuit breaker is installed using annotations on a method invoking an RPC. This annotation indicates the thrown exceptions or return values that should increment the error counter used by the circuit breaker. This represents the most commonly used circuit breaker implementation today. For example, `Hystrix` [30], Resilience4j [38], Polly [25], and `pybreaker` [27] all provide the ability to decorate methods or other functional (*e.g.*, lambda) interfaces.

3. **Client-explicit.**
   A *client-explicit* circuit breaker is installed by explicitly adding a decorator on the RPC client (*e.g.*, GRPC interceptor, HTTP client decorator) to enable the circuit breaker. This circuit breaker can be shared across multiple clients, if in a programming language that shares objects by reference. For example, Resilience4j [38], AppvNext's Polly for .NET [25], `circuitbreaker` for Go [33], and Armeria's `CircuitBreaker` [23] all allow circuit breakers to decorate a RPC client. Client-explicit circuit breakers can also be used differently:

   (a) **1 client-explicit.**
   With a *1 client-explicit* circuit breaker, a single client is used for multiple RPCs from different call sites. This may be important to reduce overhead of establishing a new RPC client for RPCs that incurs the overhead of establishing a TCP connection, resolving DNS, and potentially setting up required TLS connections.

   (b) **N client-explicit.**
   With an *N client-explicit* circuit breaker, a new RPC client is used for each RPC invocation.

4. **Client-transparent.** A *client-transparent* circuit breaker operates the same as a *client-explicit* circuit breaker using decorators or interceptors, but is automatically installed through automatic instrumentation (*e.g.*, `javaagent`) or inclusion of a third party RPC library with the circuit breaker already attached. For example, DoorDash's `Hermes` [18], a RPC library that wraps GRPC invocations, automatically attaches circuit breakers to each RPC client using interceptors.

A survey of circuit breaker implementations did not find any instances of implementations of *method-transparent* or *callsite-transparent*. In fact, I believe that the *callsite-transparent* circuit breaker design encompasses the best of the aforementioned designs: it avoids the developer overhead of the explicit designs and is scoped to an individual RPC invocation site.

8.2 Case Study: Method Indirection and Circuit Breakers

In order to understand the impact of application design on both circuit breaker choice and its effectiveness at fault tolerance, I present the first of two case studies on circuit breakers. These case studies are inspired by observations on application design and circuit breaker usage at a large food delivery service built on an MSA, that relies on circuit breakers for fault tolerance. These case studies have been both abstracted and simplified for confidentiality and brevity.

For this case study, I consider the simplified case where customers can place delivery orders through a mobile application. In this example, the creation, modification, and cancellation of orders is performed by a service in their microservice architecture: orders.

One possible implementation of the orders service, presented in Python*esque* pseudocode for brevity, is depicted in Figure 11. Here, orders exposes three RPC endpoints: create, update, and delete. Each of these three endpoints takes a dependency on the auth service in order to manage the lifecycle of the payment that is associated with the order. When the orders service receives a request to cancel an order, it first performs some business logic and then issues an RPC to the auth service to cancel the payment for the cancelled order. If that succeeds, it responds with a success; otherwise, it returns an error that propagates back to the user and asks them to try again. In Figure 12, an application of method indirection used to reduce duplication: the changes from Figure 11 are highlighted.

8.2.1 Applying Circuit Breakers

At this point, the developer might want to install a circuit breaker to guard against the unavailability or malfunctioning of the auth service. To do this, the developer chooses a popular circuit breaker library; in our example, we use the `circuitbreaker` library for Python, one example of a *method-explicit* circuit breaker.

In Figure 13, the modifications needed to install this circuit breaker are highlighted. Here, the circuit breaker annotation that is placed on the method issuing the RPC denotes that any time a `RPCException` is thrown, the circuit breaker's error counter should increase; any other thrown exceptions should not increment the circuit breaker's counters. The `RPCException` represents a generic exception base type for all possible RPC exceptions. There are two problems with this approach.

1. First, this design assumes that all RPCs issued by the orders service using the `issue_auth_rpc` helper to different methods of the auth service, will all throw the same exceptions. This simply may not be true. For example, RPCs may use different exception types to indicate different error conditions, where only a subset of types, depending on invoked service or method, should affect the circuit breakers counters; similarly, exceptions may be parameterized with different error codes, as is the case with GRPC, where only a subset of the possible parameterizations should affect circuit breaker counters. In the case of GRPC specifically, the application may not want to affect circuit breaker counters on a GRPC exception where the error code indicates resource not

```
@orders.method("create")
def order_creation(...):
    // ...

    try:
        res = rpc(auth, "create", [order_id, amount])
        return order_id
    except Exception as e:
        log_error(e)
        // ...
        return as_error(e)

@orders.method("update")
def order_modification(...):
    // ...

    try:
        res = rpc(auth, "update", [order_id, amount])
        return True
    except Exception as e:
        log_error(e)
        // ...
        return as_error(e)

@orders.method("delete")
def order_cancellation(order_id : String):
    // ...

    try:
        res = rpc(auth, "delete", [order_id])
        return True
    except Exception as e:
        log_error(e)
        // ...
        return as_error(e)
```

Figure 11: *Orders* service with 3 RPC methods.

found, as this may be an error condition that doesn't indicate failure; whereas the application would want to increment counters on a connection error error code. Therefore, the method indirection used in this example implies that all RPC failures should be treated in the same manner.

2. Second, if a latent application fault in the auth service happens to cause just one particular RPC method to return errors (*i.e.*, delete) , the circuit breaker will short-circuit *all* RPCs to the auth service (*i.e.*, create, update) even when these two endpoints may not be malfunctioning. In short, our resilience measures have disabled correctly functioning endpoints of the application in trying to prevent against the malfunctioning of one specific endpoint. Therefore, the method indirection used in this example implies that all RPC failures exhibited by the specific method executing the RPC should be treated in the same manner.

In short, *if all failures are treated similarly* (1), and *some failures only occur on some of the RPC endpoints* (2), then *failures of one endpoint will affect the circuit breaker for all.*

Historically, this makes sense. Circuit breakers were initially designed under the assumption that services fail completely: for example, in the case of an instance termination. Therefore, as long as indirection is used in a way where different functions are responsible for RPCs to different services, circuit breakers operate as expected; otherwise, the system risks further unavailability.

- **Key observation:**
  With code that has been structured to take advantage of method indirection, circuit breakers may both reduce and increase resilience of the application. This occurs because the circuit breaker may inadvertently disable properly functioning components of the application when trying to disable malfunctioning components.

- **Key insight:**
  For granular fault tolerance, developers should refactor code to isolate RPC invocations that need separate circuit breaking, depending on circuit breaker choice.

```
@orders.method("create")
def order_creation(...):
    // ...
        res = issue_auth_rpc("create", [order_id, amount])
    // ...

@orders.method("update")
def order_modification(...):
    // ...
        res = issue_auth_rpc("update", [order_id, amount])
    // ...

@orders.method("delete")
def order_cancellation(order_id : String):
    // ...
        res = issue_auth_rpc("delete", [order_id])
    // ...

def issue_auth_rpc(method, args)
    return rpc(auth, method, args)
```

Figure 12: Figure 11 with function indirection.

```
@circuit(expected_exception=RPCException)
def issue_auth_rpc(method, args):
    // ...
```

Figure 13: Figure 12 with *method-explicit* circuit breaker.

8.2.2 Refactoring for Granular Fault Tolerance

Aligned with the key insight from the previous section, to provide fault tolerance for each RPC method, the code must be refactored so that each RPC invocation to a different RPC method has its own encapsulating method with its own circuit breaker.

This refactoring is depcited in Figure 14. By structuring the code in this manner, it allows developers to specify precisely the failures that should affect the circuit breaker for each individual method. This is demonstrated using a different exception type for each method.

Rather obviously, and as made clear by this example, this is a rather counterintuitive implementation choice: in fact, this implementation choice only makes sense when circuit breakers are present as it goes against many common programming conventions regarding code reuse. In fact, considering progression from Figure 11 to Figure 14, the resulting implementation is arguably the most verbose, done only to support circuit breaker behavior.

However, in the presence of *method-explicit* circuit breakers it makes sense: using a method-explicit circuit breaker implies that circuit breaker behavior is scoped to the invoking method. Similarly, a *client-explicit* circuit breaker would require different clients; and a *callsite-explicit* circuit breaker would require different call sites for each RPC invocation for precise circuit breakers.

- **Key observation:**
  When using *explicit* circuit breakers, code duplication is required at whatever level the circuit breaker operates at for granular fault tolerance: *callsite*, *method*, or *client*.

- **Key insight:**
  Developers must carefully add new RPC endpoints to ensure that the invoking services properly tolerate faults. This may be difficult in practice, due to the nature of decentralized development inherent to microservices.

8.3 Case Study: Data Nondeterminism and Circuit Breakers

In the previous section, a minor refactoring of the orders service, in order to abstract the method used for RPC invocation, introduced a number of complexities when it came to circuit breakers. In that example, existing circuit breaker designs were only sufficient when the application was designed with circuit breakers in mind. In this section, I demonstrate how the use of abstraction, to support a minor variation on the same set of core application behaviors, complicates the use of circuit breakers. In short, if applications use this style of abstraction, existing circuit breakers are only sufficient under one, of many, possible different application designs.

```
@orders.method("create")
def order_creation(...):
    // ...
        res = issue_auth_create_rpc([order_id, amount])
    // ...

@orders.method("update")
def order_modification(...):
    // ...
        res = issue_auth_update_rpc([order_id, amount])
    // ...

@orders.method("delete")
def order_cancellation(order_id : String):
    // ...
        res = issue_auth_delete_rpc([order_id])
    // ...

@circuit(expected_exception=AuthCreateRPCException)
def issue_auth_create_rpc(args):
    // ...
    return rpc(auth, `create`, args)

@circuit(expected_exception=AuthUpdateRPCException)
def issue_auth_update_rpc(args):
    // ...
    return rpc(auth, `update`, args)

@circuit(expected_exception=AuthDeleteRPCException)
def issue_auth_delete_rpc(args):
    // ...
    return rpc(auth, `delete`, args)
```

Figure 14: Figure 14 with proper granularity.

For this case study, consider the case where the food delivery application is expanded to support takeout orders in addition to delivery. This might sound straightforward; however when a takeout order is cancelled, a different process needs to be performed to cancel the order. As most of the code needs to be parameterized on whether or not an order is a takeout or delivery order, the developers have a number of design decisions that they now face, which we will discuss below. As a starting point, the refactored implementation presented in the previous section is assumed: see Figure 14.

When implementing this new functionality, the developers of the application realize that the code needs to be parametrized based on whether the order is a takeout or delivery order. Therefore, they are left with six possible choices for this parametrization:

1. First, the developers have to decide on whether or not they want to parameterize the cancellation method's name on whether it is delivery or takeout.
   (*e.g.*, `takeout_order_cancellation(order_id)`)
   If they decide this, they then have to decide on the following:
   (a) Do they want to subsequently parameterize the RPC service that they invoke, by creating a new service specific to takeout and delivery? (*e.g.*, `takeout_auth`)
   (b) Do they want to parameterize the RPC method that they invoke, by creating different methods on the same auth service specific to takeout or delivery orders?
   (*e.g.*, `takeout/delete`)
   (c) Do they want to overload the same method at the same `auth` service by supplying an argument that indicates whether the order is takeout or delivery?
   (*e.g.*, `[order_id, type]`)

2. Second, the developers may also choose to include the order type in the parameters of the cancellation method.
   (*e.g.*, `order_cancellation(order_id, type)`)
   They would have to make the same three choices for the second parameritization: (a) the RPC service they invoke; (b) the RPC method they invoke; or (c) the invocation arguments.

In the following, to the method invoked on the orders service is referred to as *invoking*: this indicates that it is

```
@orders.method(`takeout/cancel`)
def takeout_order_cancellation(order_id : String):
    // ...
        res = issue_takeout_auth_delete_rpc([order_id])
    // ...

@circuit(expected_exception=RPCException)
def issue_takeout_auth_delete_rpc(args):
    // ...
    return rpc(takeout_auth, "delete", args)

// ... omitted delivery variation ...
```

(a) by *Invoking* ***Method*** and *Invoked* ***Service***

```
@orders.method(`takeout/cancel`)
def takeout_order_cancellation(order_id : String):
    // ...
        res = issue_auth_delete_rpc(`takeout/delete`,
                                    [order_id])
    // ...

@circuit(expected_exception=RPCException)
def issue_auth_delete_rpc(method, args):
    // ...
    return rpc(auth, method, args)

// ... omitted delivery variation ....
```

(b) by *Invoking* ***Method*** and *Invoked* ***Method***

```
@orders.method(`takeout/cancel`)
def takeout_order_cancellation(order_id : String):
    // ...
        res = issue_auth_delete_rpc("delete",
                                    [order_id, `takeout`])
    // ...

// ... omitted delivery variation ....

@circuit(expected_exception=RPCException)
def issue_auth_delete_rpc(method, args):
    // ...
    return rpc(auth, method, args)
```

(c) by *Invoking* ***Method*** and *Invoked* ***Args***

```
@orders.method("delete")
def order_cancellation(order_id : String, type : String):
    // ...
        res = issue_auth_delete_rpc(type, [order_id])
    // ...

@circuit(expected_exception=RPCException)
def issue_auth_delete_rpc(type, args):
    // ...
    return rpc(`{}_auth`.format(type), "delete", args)
```

(d) by *Invoking* ***Args*** and *Invoked* ***Service***

```
@orders.method("delete")
def order_cancellation(order_id : String, type : String):
    // ...
        res = issue_auth_delete_rpc(type, [order_id])
    // ...

@circuit(expected_exception=RPCException)
def issue_auth_delete_rpc(type, args):
    // ...
    return rpc(auth, `{}/delete`.format(type), args)
```

(e) by *Invoking* ***Args*** and *Invoked* ***Method***

```
@orders.method("delete")
def order_cancellation(order_id : String, type : String):
    // ...
        res = issue_auth_delete_rpc([order_id, type])
    // ...

@circuit(expected_exception=RPCException)
def issue_auth_delete_rpc(args):
    // ...
    return rpc(auth, "delete", args)
```

(f) by *Invoking* ***Args*** and *Invoked* ***Args***

Figure 15: Possible parameterizations to support both delivery and takeout.

currently executing. When discussing the method on the auth service, it is referred to as *invoked* to indicate that it is called by the invoking service.

1. by ***Invoking Method*** and
   (a) ***Invoked Service.*** (Figure 15a)
   Requires that developers both duplicate the invoking method's functionality, once for each order type.
   (b) ***Invoked Method.*** (Figure 15b)
   Improves on **1a**, as while it still requires duplication of code on the invoking side, it does not require creation of a new service as it parameterizes the method that it calls.
   (c) ***Invoked Args.*** (Figure 15c)
   Further improves on **1b**, as while it still requires the same duplication, it does not require creation of a new method on the invoked service, but rather allows the developer to use arguments for control flow.

2. by ***Invoking Args*** and
   (a) ***Invoked Service.*** (Figure 15d)
   Reduces the need for function duplication by only modifying the argument list for the invoking method to contain the order type. From here, the developer can use the type parameter to derive the service that should be invoked; however, it does require the creation of a new service and that may require the same duplication as we saw in **1a**.
   (b) ***Invoked Method.*** (Figure 15e)
   Improves on **2a**, as while it still requires the modifications to include a new argument to the invoking method, it parameterizes the invoked method, similar to **1b.**
   (c) ***Invoked Args.*** (Figure 15f)
   Further improves on **2b**, as while it still requires the modifications to the invoking method, all other changes are made to the method on the invoked service, similar to **1c**.

8.3.1 Applying Circuit Breakers

In this section, a latent application bug is introduced to demonstrate the challenges of granular fault tolerance with circuit breakers, using our application designs from the previous section.

This bug only affects *one* type of order: in this case, the latent application bug causes only takeout orders, not delivery orders, to return an error when cancelled; however, the bug does not affect when orders are created or modified. This bug is localized in the auth service: in the examples where the auth service has been duplicated with two variants for each order type, we assume the bug only exists in the service for takeout orders. This application bug is inspired by an actual bug experienced by the large, industrial MSA that inspired this section, where order cancellation was broken for all orders because of a bug affecting one particular order type.

The methods that encapsulate the RPC invocations in Figure 15 are annotated with *method-explicit* circuit breakers. This reflects the most common circuit breaker implementation that is in use today. While this presentation is focused on this type, the issues discussed apply to both *callsite-* and *client-explicit*: this is consistent with our second key observation and insight.

Path-Sensitivity. To start, consider the case of **1a**. Example **1a**, presented in Figure 15a, duplicates both the invoking method and the method encapsulating the invoked RPC. Therefore, since the RPC's encapsulating method is only used for takeout order cancellations, a *method-explicit* circuit breaker work perfectly for disabling the malfunctioning method on the takeout auth service.

Compared to the rest of the designs, examples **1b** through **2c**, all of these application designs suffer from the aforementioned problems of method indirection. Therefore, when these circuits open, they will disable invocations to *all methods* on both the takeout and delivery auth services (or, in the case where there is a singular auth service, it.) This confirms the first key observation and reinforces the first key insight: that, to achieve precise method-based fault tolerance using method-explicit circuit breaking, the encapsulating methods must be duplicated for each method.

In examples **1b** and **1c** however, the invoking RPC methods are parameterized to indicate the target RPC service, method, or argument. For example, in **1b**, the invoking method is `takeout/cancel`. This would indicate that a circuit breaker that is aware of the RPC invocation path would be able to perform the precise circuit breaking needed to disable the malfunctioning method.

- **Key observation:**
  When method indirection is used, the RPC invocation path may provide enough information to distinguish different invocations from the same method.

- **Key insight:**
  Circuit breakers aware of the invocation path, can improve the precision of existing circuit breaker designs. This property is referred to as *path-sensitivity*.

Context-Sensitivity. Examples **2a**, **2b**, and **2c**, prove to be the most difficult application designs. In each of these examples, a shared encapsulating method is used for each RPC invocation and the methods that call these shared methods are also shared. The only differentiation in this example is done through a parameter provided in the argument list: this is a textbook example of data nondeterminism where a provided argument dictates subsequent control flow. In the specific case of **2a**, the provided argument determines the service to invoke; in **2b**, the provided argument determines the method to invoke; and in **2c**, the provided argument is passed through to the auth service in it's argument list.

The only way to distinguish these RPCs, sufficiently to use method-explicit circuit breaking to provide fault tolerance for a single method on the specific auth service invoked, is to inspect the contents of the RPC payload at the invoking service. It is important to remember that since the method-explicit circuit breaker is checked upon entry into the enclosing method, the exact arguments of the RPC that is about to be invoked are not yet known: for example, string interpolation, as used in both **2a** and **2b**, may change the service or method *after* the circuit breaker is checked.

- **Key observation:**
  When RPC control flow is dictated by arguments to minimize code duplication — a specific case of data non-determinism — RPCs can only be distinguished through payload inspection.

- **Key insight:**
  Circuit breakers, that are aware of the invoking RPC's payload, can improve the granularity of circuit breaking. This property is referred to as *context-sensitivity*.

8.4 Implications

In order to provide guidance to the developers of resilient microservice applications, it is necessary to first provide a more abstract view on the impact of application design on circuit breaker selection.

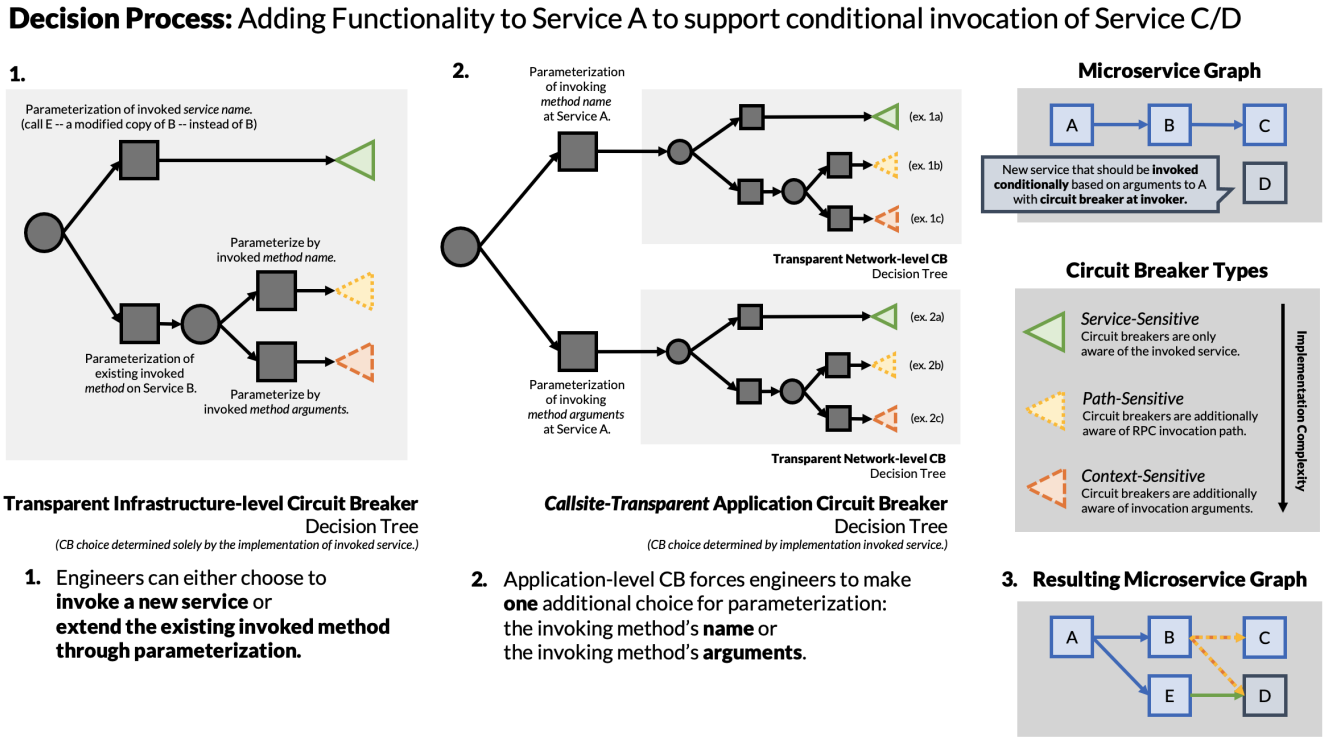
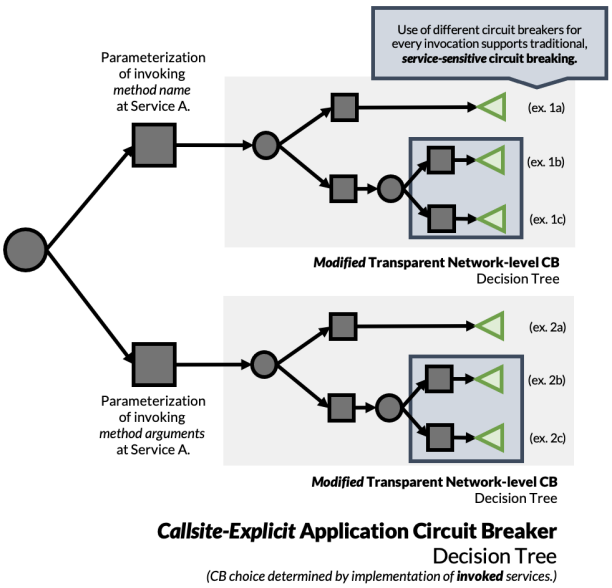
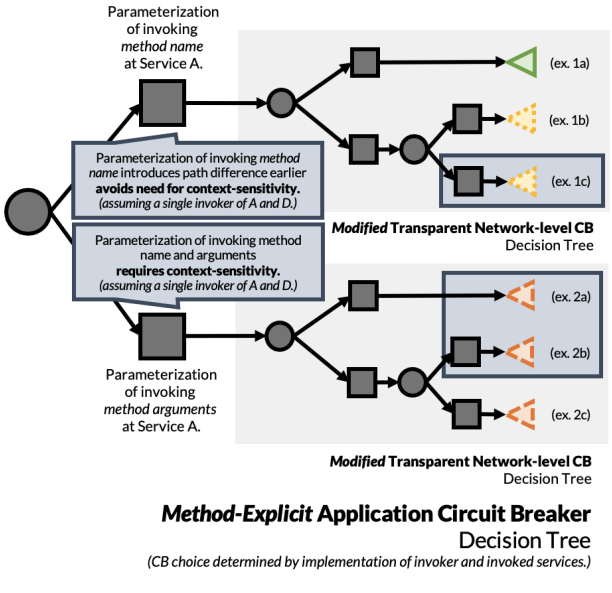


Figure 16: Decision tree relating abstraction choices to circuit breaker selection.

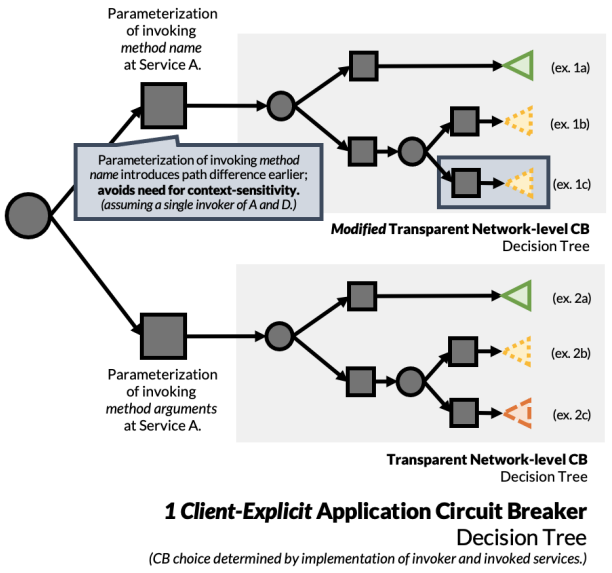
To do this, consider the example of an application composed of 3 services: **A**, **B**, and **C**. In this application, **A** issues an RPC to **B**; when **B** receives and RPC from **A**, it first issues an RPC to **C** and waits for a response before responding to **A**.



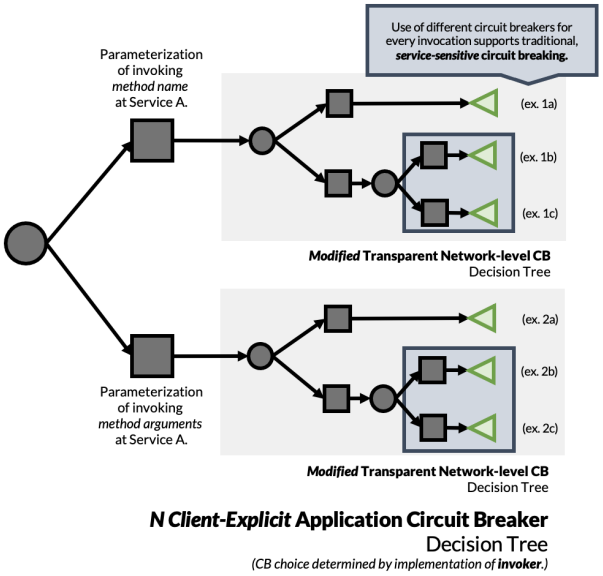
(a) *callite-explicit*



(b) *method-explicit*



(c) *1 client-explicit*



(d) *N client-explicit*

Figure 17: Decision trees determining circuit breaker choice based on application structure.

Now, the developer wants to extend **A** with conditional functionality where **B** will invoke **D** instead of **C** depending on what arguments are provided to **A**. Additionally, the application developer wants to use circuit breakers to ensure either the failure of **C** or **D** are properly tolerated. For this example, a single RPC method for both **C** and **D** is assumed.

The developer has to make several choices at this point. First, do they prefer *infrastructure* or *application* level circuit breakers? If they prefer application, do they prefer *transparent* or *explicit*? If they opt for explicit, do they prefer *client*, *method*, or *callsite* circuit breakers? Furthermore, how do they know that their selection will provide granular fault tolerance?

8.4.1 Decision Trees

To understand the implication of application design on circuit breaker choice, consider the decision tree presented in Figure 16. In this decision tree, the implications of choosing a *callsite-transparent* circuit breaker are presented: an idealized design that combines the best of all existing circuit breakers. This circuit breaker is local to each RPC site in the application code and requires no modifications to application code, as it is automatically installed through runtime instrumentation.

First, consider the choice of a transparent infrastructure-level circuit breaker, as it forms the core component of how an application-level circuit breaker functions. When a new service, **E**, is created to support the additional functionality of **D**, a infrastructure-level circuit breaker works just fine as the circuit breaker is scoped to a service. However, when an existing service, **B**, is parameterized to support the conditional invocation of **D**, either path or context sensitivity is needed. Further parameterization of **A** to support the conditional invocation of either **C** or **D** has no further effect on circuit breaker selection. However, the *callsite-transparent* circuit breaker is an ideal design that does not exist; therefore, this serves as a reference point to understand the implications for circuit breaker designs that do exist.

In Figure 17, the decision trees for the four circuit breaker designs that concrete implementations were identified for are presented: *callsite-explicit*, *method-explicit*, *1 client-explicit*, and *N client-explicit*. The differences in each diagram from the *callsite-transparent* design are highlighted.

- **callsite-explicit** (Figure 17a)
  If a different circuit breaker is used and manually installed at each call site of an RPC, a developer can manually configure that circuit breaker accordingly so that it only fires at the proper level of granularity. This is by far the most expensive approach: it requires manually creating a circuit breaker for the proper granularity, manually incrementing failure and success counters, and writing the appropriate conditionals.

- **method-explicit** (Figure 17b)
  As discussed previously, path sensitivity is needed to distinguish invocations that use shared methods and differ only by the RPC invocation path. When the path differs only by arguments — introducing data nondeterminism — context sensitivity is required.

- **1 client-explicit** (Figure 17c)
  The introduction of a different path early in the RPC invocation chain allows for path sensitivity instead of context sensitivity.

- **N client-explicit** (Figure 17d)
  If developers are willing to stomach the performance penalties and development overhead of using a new RPC client for each invocation, existing *client* circuit breaker designs work just fine.

In this example, the only case that is considered is adding functionality where a single new method is added and therefore can be added, in isolation, to a new service. In the even that there is shared functionality, as discussed in our second case study, further duplication depending on circuit breaker choice is required. This is consistent with the second key observation and insight.

8.4.2 *Path-* and *Context-Sensitivity*

How might one implement *path-* and *context-sensitive* circuit breakers?

Path-Sensitivity. Path-sensitivity requires two things. First, it needs the ability to identify and propagate an identifier that represents the RPC invocation path across multiple RPC invocations. Second, circuit breakers need to be aware of this identifier when modifying state in order to provide the proper granularity.

For example, many popular and widely deployed distributing tracing frameworks (*e.g.*, OpenTelemetry [32]) already have a mechanism for tracking and assembling request traces in microservice applications. They achieve this by propagating tracing metadata along with the RPC: in the case of HTTP, this is done with request headers;

with GRPC, a metadata facility is provided for this very purpose. This very mechanism can be used to propagate the required identifiers for path sensitivity: in fact, many academic fault prevention techniques already leverage this associated RPC metadata to determine if a fault should be injected (*e.g.*, 3MileBeach [87]) or to uniquely identify RPC locations when performing an exhaustive search of the fault space.

Deriving these identifiers, however, most likely requires further research. To understand the issue, three examples using the aforementioned technologies is provided.

- OpenTelemetry's request traces could be used to produce identifiers through a hashing function. However, these traces suffer from three problems. First, branching control flow can alter the identifier for the same RPC path. For example, consider an optional RPC invocation to from A to C before A invokes B: depending on whether or not A invokes C, the invocation from A to B may be identified differently. Second, scheduling nondeterminism may permute the order of requests in the path executed concurrency. Third, data nondeterminism may result in fully unique identifiers for every request if information such as thread identifier or request start time is included.

- 3MileBeach's [87] identifiers are significantly more coarse: while they contain the causal history of RPC invocations, they are only encoded by RPC service and method. This alleviates the problems introduced by data nondeterminism, but does not address scheduling nondeterminism or branching control flow.

- Distributed execution indexes (DEI) improve on 3MileBeach's identifiers by introducing an indexing scheme that avoids the problems of branching control flow through abstraction. However, they do not yet address data nondeterminism.

Finally, circuit breakers must be made aware of these identifiers to properly select and modify the correct circuit breaker state. This seems straightforward: for application-level and infrastructure-level, the location of identifiers in the RPC metadata could be specified in circuit breaker configuration. In fact, Armeria's `CircuitBreaker` [23] already has a mechanism for the parameterization of circuit breakers using decorators. These decorators allow for both *per service* and *per method* granularity: however, they are currently limited.

With respect to HTTP, *per method* only distinguishes between HTTP method (*e.g.*, `GET`, `POST`) and not the full URL of the request, problematic if the invoked service has more than one HTTP endpoint per method. With respect to GRPC, *per host* and *per method* achieves a higher level of granularity, but only accounts for a path containing the immediate invoker and invokee: this may be problematic if the invocation path differs earlier in the RPC invocation chain. Regardless, it seems that a similar decorator-based approach achieving the desired granularity when provided with the complete RPC invocation path as an identifier.

Context-Sensitivity. As context-sensitivity subsumes path-sensitivity, it also requires the same path identifiers and modifications to circuit breakers for path awareness. However, it additionally needs to be aware of the payload of the RPC invocation. This may prove problematic in practice for two reasons.

First, inspecting the RPC payload requires deserialization of the payload which, with many RPC frameworks, is typically binary data and may incur a performance penalty. This may be problematic depending on how the RPC framework handles deserialization and whether the application code also needs to deserialize the RPC. If the result of deserialization is memoized, this cost may be amortized. If not and deserialization is lazy, deserialization for circuit breaking may force deserialization where otherwise not necessary if the request's arguments are just passed forward to the next RPC. For an example of where such a pass-through is possible, see Figure 15f.

Second, it seems as if only a subset of fields in the RPC payload would be useful for circuit breaking. For example, in the case of our second case study, circuit breaking would most likely want to be based on order type, but perhaps not a user identifier or line item in a takeout order, as the application bug most likely would not be dependent on these fields — but could be. While decorator-based approach that uses a lambda expression to project the appropriate fields of the RPC's payload would work, in practice this solution may prove inflexible.

Due to the implementation challenges and overhead in *context-sensitivity*, context-sensitivity does not seem like an ideal solution. In fact, it may be that *path-sensitivity* meets the sweet spot of minimal overhead and granular fault tolerance. Context-sensitivity fulfills a need where an application has already been designed and must be retrofitted with circuit breakers to increase resilience until the code can be refactored. Therefore, developers shouldavoid easier to implement application designs that require context-sensitivity over slightly more verbose designs that can use path-sensitivity.

- **Path-sensitivity** can leverage the metadata that is already propagated by distributed tracing frameworks to hit the sweet spot of abstraction and granular circuit breaking.

- **Context-sensitivity**, due to its overhead, may best be suited to existing code that cannot be immediately refactored, but needs fault tolerance until refactoring can be done.

Infrastructure- vs. Application-level Circuit Breakers. Finally, it is important to discuss the industrial trend towards *transparent* infrastructure-level circuit breaking. While the transparency provided by this style of circuit breaking is quite desirable from a development point of view, it is rather inflexible in its current state. Currently, infrastructure-level circuit breaking is only scoped to a failing *instance* of a service: therefore, it has no awareness of methods or arguments and can adversely impact the application when one method, accessed at a high enough throughput rate, happens to fail. Therefore, without *path-* or *context-sensitivity*, infrastructure-level should be avoided.

- **Application-level** circuit breaking can provide fault tolerance granularity; however, may require code duplication until robust *path-* and *context-sensitive* implementations emerge.
- **Infrastructure-level** circuit breaking should be avoided until these designs emerge due to its inflexibility.

8.5 Proposed Work

For this research proposal, I plan to extend the application corpus with application corpus with concrete implementations of MSAs that demonstrate the problems exhibited by both of the case studies presented in this section. From there, I intend to build upon the foundational indexing technique I have created, DEI, to implement the proposed *path-* and *context-sensitive* circuit breaker designs presented in this section. I plan to both validate these new circuit breaker designs using the new examples added to the application corpus, and evaluate their performance.

9 Proposed Contribution: Testing of Fault Tolerance Techniques

This section presents both work currently in progress and proposed work.

The integration of fault tolerance techniques in an MSA is only the first step to building a MSA resilient to faults. Those fault tolerance techniques must be tested to ensure that, when faults inevitably occur, they operate correctly. Rather unfortunately, little research exists on testing fault tolerance techniques in MSAs. My plan with this research proposal is to address this deficiency through an extension of the SFIT technique.

The three main fault tolerance techniques that we are concerned about are: fallbacks, circuit breakers, and load shedding. When it comes to fallbacks, specifically fallbacks that are executed upon a single RPC failure, SFIT is already sufficient at testing and verifying that they operate correctly. SFIT need only induce a single RPC failure in order to provoke the application into executing this fallback behavior. However, as discussed in our section on circuit breakers (§8), fallback logic that is associated with circuit breakers may require repeated failure of the same RPC in order to execute. Similarly, the same constraint applies to the testing of both circuit breakers and load shedders. Therefore, it would seem that an extension of SFIT that performs repeated fault injection, of the same RPC, is necessary to test these MSA-specific fault tolerance techniques.

I describe my proposed approach for addressing this deficiency below.

- First, I plan to extend the SFIT approach to be able to target a particular test scenario for repeated execution. This would involve selecting one of SFIT-generated test cases, where one or more failures are injected in an MSA, for repeated execution. From there, developers would be able to manually specify the failure injection interval (to trigger the circuit breaker), what the recovery period is (where no faults are injected), and the invariants that should hold both when the circuit is open and after the circuit has closed. I have already prepared a preliminary prototype of this design and evaluated it on a sample Python application; I am currently in the process of evaluating on an industrial MSA, DoorDash.
- Second, I plan to investigate how to automatically determine test cases that should be selected for this style of testing. As clear from the aforementioned approach, it still requires that developers manually select the test execution where a circuit breaker (or load shedder) is present. It remains unclear if this automatic determination can be done (*i.e.,* may be undecidable), but I believe that several heuristics may be useful for either prioritizing the search or ensuring optimal test case coverage. I plan to explore each of them:
  - First, and rather obviously, a strategy that randomly samples test cases as part of a continuous integration process could ensure coverage over some arbitrary number of runs. This approach, while the most straightforward, only ensures fault prevention across a predefined time period where all cases are guaranteed probabilistically.
  - Second, it may be possible to start this style of testing deeper in the application graph (*i.e.,* RPC depth, with respect to the application graph and executing functional test) to detect failures earlier in the call chain. For example, by triggering a circuit breaker at services 3 levels deep in the call chain, it may be possible to provoke circuit breakers earlier in the call chain.

– Third, and most promising, it may be possible to use the type of circuit breaker (*c.f.*, this proposal's circuit breaker classification) in order to reduce the test cases that need to be explored. For example, when method-explicit circuit breaking is used, it is not necessary to all of the test cases that issue RPCs through the same method, as testing only a single RPC is sufficient to determine if a circuit breaker is present on that method. This is similar to the type of heuristic that is used by the Dynamic Reduction algorithm, presented in (§7) and can leverage the information provided in DEIs (§6).

I plan to evaluate my solution for this using an extension of the application corpus that employs circuit breakers, as discussed in (§8).

10 Evaluation

In this section, we describe the implementation of our prototype of SFIT, called Filibuster. Then, we discuss the completed preliminary evaluations of this prototype and present proposed work, towards completion of this research proposal.

10.1 Prototype Implementation: Filibuster

My prototype of SFIT, named Filibuster, is primarily implemented in Python and is currently open source. Filibuster currently supports fault injection with services that are written in either Python or Java[7], and functional tests for applications can be written in any language through the use of a shell script wrapper.

10.1.1 Instrumentation

In terms of instrumentation for specific libraries, Filibuster supports:

- **Python.** Filibuster provides instrumentation libraries for:
  - `requests`: a popular client library for HTTP;
  - `grpc`: the Google implementation of GRPC for Python; and
  - `flask`: the web service framework.

  With Python, manual instrumentation of services is required; however, as the Filibuster implementation is derived from the `opentelemetry` instrumentation for these services, code modification for testing with SFIT only requires a modification to the import statement.
- **Java.** Filibuster provides instrumentation libraries for:
  - `com.linecorp.armeria`: a popular web framework used for building microservices and issuing RPCs with HTTP and GRPC; and
  - `io.grpc`: the Google implementation of GRPC for Java.

  With Java, manual instrumentation of applications can be performed through the use of decorators. However, Filibuster also provides automatic instrumentation, by building upon the automatic instrumentation capabilities of the Java `opentelemetry` library. Use of automatic instrumentation requires no code modification through the use of a runtime parameter.

Filibuster's instrumentation libraries assign DEI's, as well as vector clocks, to each intra-service RPC. This information is forwarded between RPCs using the RPC's protocol-specific metadata facilities: in the case of GRPC, this is performed using GRPC metadata; with HTTP, this is performed using HTTP headers. While DEI's identify a RPC for fault injection and support dynamic reduction, vector clocks are only used to identify the structure of the MSA's call graph. his is an artifact of the current implementation and can be fully replaced by DEI's; however, remains in the current instrumentation which predates the use and invention of DEI's.

[7] or, any language that runs on the JVM.

10.1.2 Command Line Utility

The Filibuster command line utility (CLI), starts the Filibuster server, which each service communicates with, and is used to execute functional tests with fault injection. Given a functional test, provided as a shell script or any program that returns error codes that indicate whether or not the test passes, the Filibuster CLI will perform repeated execution while injecting faults until the search space is exhausted. Filibuster's CLI also allows the developer to, upon a failed test execution, extract a counterexample that can be used to replay that specific failure. These counterexamples are specific to a given implementation of a service, as they encode RPCs using DEIs, but are portable. This functionality can also be used to extract a trace of DEI's for any given Filibuster execution.

For Java specifically, Filibuster's CLI tool provides Java and `junit` specific features: for example, the ability to directly run a `gradle` test by name or to pause to allow attaching a remote debugging session.

For Python specifically, developers can opt to use the Filibuster CLI to also start the services that are being tested. When this is done, Filibuster can use the coverage aggregation mechanism to produce combined coverage reports across all generated Filibuster test executions. For Java, no Filibuster specific coverage mechanism currently exists.

10.1.3 Static Analysis

Filibuster also provides a secondary CLI tool that is used for performing the required static analysis of services to determine the error codes they return. This tool produces a JSON file that is provided to the main Filibuster CLI used to determine what faults to inject. This tool performs a purely lexical analysis that over-approximates errors by using abstract syntax tree traversals for the supported Python and Java libraries.

In the event that static analysis is not possible, a default analysis file is provided that contains common errors to both GRPC and HTTP applications. These include the *unchecked*, runtime exceptions for HTTP and GRPC that indicate timeout or connection error as well as the common HTTP error codes that are returned in response objects indicating server error and service unavailability.

In the event that developers want more specific control over fault injection (*i.e.*, specific timeout configurations or custom error conditions), manual modification or crafting of a file can be used. In the worst case, developers can opt to test for all possible errors, as HTTP and GRPC have a small, finite space of possible error codes.

10.1.4 Conditional Assertions

Filibuster's conditional assertions are provided by a HTTP API exposed by the Filibuster server. With Java, a helper module wraps these API calls to provide easy integration into existing `junit` tests; with Python, the same exists for `pytest`.

10.1.5 Load Generator

Filibuster provides a third CLI for load generation. Given a counterexample file that describes a single test execution, developers can use this to repeatedly re-execute the functional test using a configurable load generator in order to test the system under load, with faults present.

10.2 Service-level Fault Injection Testing

My initial evaluation of my prototype implementation Filibuster was published at the ACM Symposium on Cloud Computing (SoCC) in 2021 [67].

Table 1, presents results from running Filibuster on the corpus; in the table we shorten Dynamic Reduction to DR. For faults, it is assumed that all remote calls can return a connection error. When a timeout is specified, timeout exceptions are considered. Any service-specific failures are also included, as determined by our static analysis.

All of the examples were run on a AWS CodeBuild instance with 15 GB of memory and 8 vCPUs. At the start of the Filibuster run, all of the services were started for each example; Filibuster waited for those services to come online and the services were manually terminated at the end of the run. As most of the applications in the corpus have no side-effects, they seed the system with values and verify they can be read, so the services are not restarted in between test executions. However, this option is available. Given that the cost of the service restart is fixed, that cost is excluded when comparing the performance of the system with and without dynamic reduction.

10.2.1 Tests Generated and Increased Coverage

In order to determine the benefit to developers in identifying resilience issues, it makes sense to first consider the number of tests generated by Filibuster and the resulting increase in code coverage.

Example	Test Gen/DR Gen	Coverage After (%)	Time w/DR (s)	DR Overhead (ms)	TG Overhead (ms)
cinema-1	9/9 (-0)	90.72 (+5.67)	8.83 (+1.16)	0.46 (0.02)	0.60 (0.06)
cinema-2	10/9 (-1)	90.76 (+5.64)	8.81 (+1.15)	0.43 (0.01)	0.64 (0.06)
cinema-3	91/37 (-54)	91.08 (+6.43)	13.21 (+5.54)	34.10 (0.02)	4.09 (0.04)
cinema-4	34/21 (-13)	91.34 (+8.17)	12.11 (+4.23)	3.25 (0.01)	2.31 (0.06)
cinema-5	25/25 (-0)	90.72 (+5.16)	11.17 (+3.51)	2.23 (0.01)	1.57 (0.06)
cinema-6	41/41 (-0)	91.35 (+9.05)	13.99 (+6.28)	5.91 (0.01)	2.57 (0.06)
cinema-7	45/45 (-0)	91.28 (+6.64)	14.41 (+6.71)	6.37 (0.01)	2.71 (0.06)
cinema-8	21/21 (-0)	92.70 (+8.33)	10.47 (+2.88)	1.66 (0.01)	1.37 (0.06)
Audible	69/31 (-38)	96.04 (+12.75)	15.28 (+6.35)	13.35 (0.01)	4.72 (0.06)
Expedia	17/17 (-0)	98.54 (+15.33)	9.87 (+6.35)	1.15 (0.01)	1.06 (0.06)
Mailchimp	135/134 (-1)	98.96 (+11.54)	59.83 (+52.01)	473.48 (0.02)	44.07 (0.32)
Netflix					
– no bugs	1606/1603 (-3)	96.31 (+17.25)	513.83 (+504.85)	94566 (0.09)	6748.93 (4.20)
– w/ bugs (#2, #3)	18653/4670 (-13983)	97.38 (+15.67)	2303.84 (+2293.8)	748750 (0.07)	62100.34 (3.32)
– w/ bugs (#1, #2, #3)	18653/4670 (-13983)	97.38 (+15.67)	2363.84 (+2353.8)	744052 (0.07)	60002.91 (3.31)

Table 1: Evaluation results: FILIBUSTER on the corpus. Includes number of generated tests with and without dynamic reduction; coverage before and after using FILIBUSTER, overhead of dynamic reduction algorithm, and overhead of test generation.

The "Test Gen/DR Gen" column presents the number of tests both generated and executed by FILIBUSTER. Since each example only has a single functional test, these numbers include that test in the total, as FILIBUSTER must execute the initial passing functional test first to identify where to inject failures. In all of the examples containing bugs in the corpus, the bugs were able to be identified using FILIBUSTER.

The "Coverage After" column shows the increase in statement coverage. By generating the tests that cover possible failures, FILIBUSTER is able to increase coverage of the application. These numbers only account for functional tests. The generated tests increase coverage related to error-handling code not exercised by the unmodified functional test.

Takeaway: FILIBUSTER was able to prevent developers from having to write time-consuming mocks by automatically generating tests that introduce failures at all of the remote call sites. As demonstrated by the Netflix example, some of these applications are large enough to require a large number of tests to properly ensure coverage of the failure space. For most organizations, manually writing this many tests without a system to automatically generate these tests would be expensive in terms of development time. Similarly, the cost of test adaptation is also low. In the Netflix example, FILIBUSTER executed 1,606 tests, but required only 9 conditional assertions to capture all behavior. FILIBUSTER also found all of the bugs in a *development setting*, without having to run chaos experiments in a live, production environment. Recall from Section 4, all of these bugs were discovered using chaos engineering and were used as use cases to advocate for the adoption of chaos engineering. Using FILIBUSTER, chaos engineering can be avoided.

10.2.2 Dynamic Reduction

The "Test Gen/DR Gen" column shows the benefits of dynamic reduction: yellow cells are used to identify impact; green cells are used to identify significant impact.

Dynamic reduction excels when graphs have more depth and less breadth. In the Audible example, there are deep paths containing nested requests that can allow FILIBUSTER to avoid running redundant test executions. However, in the Netflix example (without bugs), the graph has a large breadth with little-to-no depth. In this case, all combinations of failures have to be tested, as control flow in the application could be based on a request failure. Furthermore, in the Netflix example (with bugs) where deeper paths are introduced through additional fallback behavior, the benefits of dynamic reduction become valuable–only 25% of the tests have to be executed to reach the same failure coverage.

Takeaway: When applications are structured in a way where there is depth over breadth to the service graph, applications can significantly benefit from dynamic reduction. This occurs because our design can observe the behavior of services when their dependencies fail earlier in the exploration of the failure space — this information can be used used to avoid running subsequent tests where that behavior is already known. This insight can guide the design of microservice architectures to decrease the cost of testing — deeper service graphs allow for reuse of results across test executions. This results in reduction of overall test time required to exhaust the space of possible failures.

10.2.3 Mocks

During the initial corpus implementation, unite tests were written for each service in each example using mocks to account for possible remote service failures. When writing these tests, only independent failures were considered. Refer to Figure 3 and consider the Audible Download Service. In this example, unit tests were only written that each containing a single mock for the failures of the three dependencies: Ownership, Activation, and Stats. The list of service specific failures is omitted here here, and the reader referred to the diagram for the list; for exceptions, a mock was written for each of the two exceptions: `Timeout` and `ConnectionError`.

Not only was this process time consuming, from learning the mocking framework to writing and verifying they worked correctly, it was a significant amount of additional code. These failures also under-approximate the actual failures that could occur in the application: mocks were not written that verified all possible combinations of failures. For example, the failure of both the Stats service and the Asset Metadata service would require a combination of two mocks on two different services. As an example of how much code is required to write these mocks, the implementation of all Netflix services was 936 LOC. In total, an additional 743 LOC (+79.3%) of test code was written to verify failure behavior.

Takeaway: Filibuster can be used to verify resilience without the time consuming, ad-hoc and error prone effort of writing mocks for what failures the developers believe are possible. Filibuster can automatically generate these tests with minimal effort and accounts for more complicated mocking scenarios, where multiple mocks across different services are required to execute a particular error handling code path.

10.2.4 Execution Time

The "Time w/DR" column shows the execution time with dynamic reduction enabled. This column shows the total execution time for all tests, excluding setup and teardown time. In parentheses, the difference between running the initial single functional test and running all of the tests generated by Filibuster is presented.

Comparing this difference to the number of tests both generated and executed with dynamic reduction, it is clear, and is expected, that the execution time scales linearly with the number of tests that have to be executed. his per test execution time accounts for starting a Python interpreter, performing whatever setup and teardown is required and executing the test.

In the "TG Overhead" column, the total overhead (in milliseconds) for test generation is presented. This test generation process, running inside the Filibuster server, schedules new test executions each time a new request is reached and the Filibuster server learns about this call through the instrumentation call from the service. As is clear, this overhead is very small. In parentheses, the overhead for each test that is generated is presented: which in the worst case is 3.2 milliseconds. In the "DR Overhead" column, the total overhead (in milliseconds) introduced by the dynamic reduction algorithm is presented. This algorithm has to, for each test that is generated, determine if this test is redundant with a previous test execution. As is clear, this overhead is very small. In parentheses, the overhead per test is presented: in the most complicated examples it is 90 microseconds.

Takeaway: Filibuster's execution time scales linearly with the number of tests that are generated. However, the test generation overhead is significantly less than the cost of the development time required in manually writing these tests using mocks. Additionally, Filibuster provides higher coverage by automatically writing mocks for combinations of failures across service boundaries.

10.2.5 Misconfigured Timeouts

In order to identify misconfigured timeouts, where Service A calls to Service B with a timeout that is less than Service B's timeout to a Service C, is performed by sleeping the timeout interval plus 1 additional millisecond, before returning a `Timeout` exception. This ensures that Filibuster waits at least long enough to account for the timeout interval.

In Figure 1, the difference in execution time when testing timeouts is highlighted in red. In order to identify Netflix bug #1, Filibuster must execute the timeouts while sleeping the timeout interval. Compared to the execution where timeouts are not considered, the difference in time of the cumulative timeout interval during testing can be observed.

Takeaway: Filibuster can detect incorrectly configured timeouts at the cost of additional execution time, equivalent to the injected timeout durations.

10.3 Proposed Work

In this section, I detail the additional evaluations I would like to perform towards fulfillment of this research proposal.

10.3.1 Evaluation of Design Choices in DEI

In (§10.2), and as part of the SFIT evaluation, we evaluated the use of the synchronous variant of Distributed Execution Indexes (DEI) in enabling the SFIT-DR optimization.

In (§6), we proposed an asynchronous variant of DEI that augments the synchronous variant of DEI with inclusion of the RPC's payload. Through this inclusion, SFIT can avoid control of the thread scheduler when performing an exhaustive search, under the assumption that MSAs do not issue concurrent RPCs, from the same call site and calling context, to the same service, with the same payload.

Towards validation of this design choice, I performed an initial evaluation using a large industrial MSA and found no evidence of this programming pattern, thereby justifying the design decision. However, I have not been able to publish this work yet.

I believe there are several possible paths forward:

1. First, it may be possible to perform a more comprehensive empirical analysis to justify these decisions, possibly through the inclusion of additional, industrial MSAs, should it be possible to get access to them. Along this same line of thinking, I believe it may be possible to strengthen the existing empirical analysis.

2. Second, it may be possible to use existing indexing techniques (*e.g.*, opentelemetry, 3MileBeach [87]) to demonstrate how existing indexing techniques are either too granular (*c.f.*, opentelemetry) or too coarse (*e.g.*, 3MileBeach) to properly identify RPCs in the face of scheduling nondeterminism in order to further justify our design decisions along with my initial empirical analysis.

3. Third, it may be possible to use an empirical evaluation that identifies actual bugs in an industrial MSA to justify the applicability of the technique, if I was able to discover bugs through fault injection and the application of SFIT to their code.

10.3.2 Fault Tolerance

As discussed in (§8), I proposed two new designs of circuit breakers that address the deficiencies in the implementations that are currently in use in industrial MSAs. I plan to both implement and evaluate these new designs using an extension of the application corpus (§4) to demonstrate that they do address these deficiencies and are practical in their design.

As discussed in (§9), I proposed an extension of SFIT that specifically targets testing of the primary fault tolerance techniques in use by industrial MSAs: circuit breakers and load shedding. I plan to perform an evaluation that is similar to the evaluation presented in (§10.2) that examines the usefulness in (A) identifying fault tolerance bugs, (B) the resulting increase of code coverage, and (C) the algorithm I plan to use to identify optimal testing of these fault tolerance techniques.

10.3.3 Application to Industrial MSAs

Finally, I plan to perform one additional evaluation: either a study on the application of Filibuster to an industrial MSA, or a qualitative study of an industrial MSA in order to demonstrate that the faults that SFIT targets are faults that result in application outages.

In terms of the *qualitative study*, I plan to perform it by studying the postmortem documents of an industrial MSA that describe outages that they have experienced with the root causes identified. From there, I plan to use grounded theory to identify the faults that resulted in application outages, and identify a theory that explains the connection between the lack of use of fault prevention techniques and failed or missing fault tolerance techniques.

In terms of the *application* of my research to industry, it remains unclear what this study will look like at the moment. Ideally, I would like to have concrete results that demonstrate that SFIT can be used to identify bugs on an industrial MSA, but it may be difficult within the provided time to identify bugs, given the complexities of applying the approach to an industrial MSA. Therefore, I imagine this evaluation could take several different forms.

1. **Developer interviews.**
   Developer interviews could be used to gauge how easy it is to enable services for use with Filibuster and successfully perform testing using the Filibuster tool and SFIT technique. This approach could target one area that needs further evaluation: the process of writing assertions required by SFIT, as we currently do not know if developers actually want to write assertions in this style (or, if they are valuable at all.) To perform

this evaluation, I believe that the Standard Usability Scale (SUS), augmented with a number of open-ended response questions targeted as specific components of the SFIT process using Filibuster.

2. **Discussion of integration challenges.**
   Another approach that seems viable is to, after completing an initial integration of Filibuster to one or more services in their platform, is the discussion of the technical challenges that integration had to overcome. I have already identified a number of these challenges — concurrency, automatic instrumentation, etc. — and imagine that more will be discovered the deeper the integration is performed. One possible outcome of this is perhaps a toolkit to enable future researchers a basis for integrating new fault injection approaches for microservices or at a minimum a set of design principles for practical research in the area moving forward.

3. **Discussion of discovered faults.**
   Finally, the ideal evaluation is the presentation of discovered bugs and the process I undertook to identify them. This is the most challenging, but would be the most impressive form of validation for the designs presented in this proposal.

I have already started this process of integration as part of an internship this summer and have an initial prototype of Filibuster running on one of their critical services.

11 Dissertation Timeline

After the thesis proposal, I plan to apply for ABS status and begin working at DoorDash full-time, as a Researcher on the Platform Evolution team, specifically working on integration of the Filibuster prototype into their platform.

This role has provided me with the opportunity to work half-time on completion of this thesis, while also providing me with access to an industrial MSA that I can use for completion of several of the proposed evaluations in this research proposal: (A) the justification of our DEI design choices; (B) the qualitative study on faults in industrial MSAs; and (C) the study of the applicability of SFIT to an industrial MSA. While at DoorDash, I plan to continue to asynchronously meet with my advisor and members of my committee, ideally weekly or bi-weekly, towards completion of this thesis.

Given that I believe the proposed research and proposed additional evaluations should take approximately one year of work to complete, I estimate that completion of this thesis will take between 1 - 2 years to complete, with an ideal, estimated defense date of May, 2024.

References

[1] The Netflix Simian Army - The Netflix Technology Blog. https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116, 2011. Accessed: 2022-06-05.

[2] FIT: Failure Injection Testing. https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2, 2014. Accessed: 2022-06-05.

[3] Building Microservices in Python and Flask. https://codeahoy.com/2016/07/10/writing-microservices-in-python-using-flask, 2016. Accessed: 2021-05-21.

[4] LinkedOut: A Request-Level Failure Injection Framework. https://engineering.linkedin.com/blog/2018/05/linkedout--a-request-level-failure-injection-framework, 2018. Accessed: 2022-06-05.

[5] Introducing Domain-Oriented Microservice Architecture. https://eng.uber.com/microservice-architecture/, 2020. Accessed: 2021-05-21.

[6] Rethinking How the Industry Approaches Chaos Engineering. https://www.infoq.com/presentations/rethinking-chaos-engineering, 2020. Accessed: 2021-05-21.

[7] Amazon EKS | Managed Kubernetes Service. https://aws.amazon.com/eks/, 2021. Accessed: 2021-05-21.

[8] Audible. https://www.audible.com, 2021. Accessed: 2021-05-21.

[9] Chaos Engineering Saved Your Netflix Extreme stress testing of online platforms has become its own science. *IEEE Spectrum*, 58(3):4–10, March 2021.

[10] docker. https://www.docker.com/, 2021. Accessed: 2021-05-21.

[11] Expedia. https://www.expedia.com, 2021. Accessed: 2021-05-21.

[12] Flask web framework. https://flask.palletsprojects.com/en/2.0.x/, 2021. Accessed: 2021-05-21.

[13] Gremlin. http://www.gremlin.com, 2021. Accessed: 2021-05-21.

[14] Mailchimp. https://www.mailchimp.com, 2021. Accessed: 2021-05-21.

[15] minikube. https://minikube.sigs.k8s.io/docs/, 2021. Accessed: 2021-05-21.

[16] Netflix. https://www.netflix.com, 2021. Accessed: 2021-05-21.

[17] A Guide to gRPC and Interceptors - The Edgehog Portal. https://edgehog.blog/a-guide-to-grpc-and-interceptors-265c306d3773, 2022. Accessed: 2022-06-05.

[18] Building a gRPC Client Standard with Open Source to Boost Reliability and Velocity. https://doordash.engineering/2021/01/12/building-a-grpc-client-standard-with-open-source/, 2022. Accessed: 2022-06-05.

[19] Chaos Blade. https://chaosblade.io, 2022. Accessed: 2022-06-05.

[20] Chaos Mesh. https://chaos-mesh.org, 2022. Accessed: 2022-06-05.

[21] ChaosToolkit. https://chaostoolkit.org, 2022. Accessed: 2022-06-05.

[22] Circuit breaker — Akka documentation. https://doc.akka.io/docs/akka/current/common/circuitbreaker.html, 2022. Accessed: 2022-06-05.

[23] Circuit breaker — Armeria documentation. https://armeria.dev/docs/client-circuit-breaker/, 2022. Accessed: 2022-06-05.

[24] Circuit Breaking with Envoy. https://blog.turbinelabs.io/circuit-breaking-da855a96a61d, 2022. Accessed: 2022-06-05.

[25] GitHub: App-vNext/Polly. https://github.com/Comcast/jrugged, 2022. Accessed: 2022-06-05.

[26] GitHub: Comcast/jrugged. https://github.com/Comcast/jrugged, 2022. Accessed: 2022-06-05.

[27] GitHub: danielfm/pybreaker. https://github.com/danielfm/pybreaker, 2022. Accessed: 2022-06-05.

[28] GitHub: kubernetes/kubernetes. https://github.com/kubernetes/kubernetes, 2022. Accessed: 2022-06-05.

[29] GitHub: Netflix/chaosmonkey. https://github.com/Netflix/chaosmonkey, 2022. Accessed: 2022-06-05.

[30] GitHub: Netflix/Hystrix. https://github.com/Netflix/Hystrix, 2022. Accessed: 2022-06-05.

[31] GitHub: Netflix/SimianArmy. https://github.com/Netflix/SimianArmy, 2022. Accessed: 2022-06-05.

[32] GitHub: OpenTelemetry - CNCF. https://github.com/open-telemetry, 2022. Accessed: 2022-06-05.

[33] GitHub: rubyist/circuitbreaker. https://github.com/rubyist/circuitbreaker, 2022. Accessed: 2022-06-05.

[34] Hystrix : How to implement fallback and circuit breaker. https://medium.com/@kullik2/hystrix-how-to-e41cabf34d40, 2022. Accessed: 2022-06-05.

[35] Implementing a Circuit Breaker with Resilience4j. https://reflectoring.io/circuitbreaker-with-resilience4j/, 2022. Accessed: 2022-06-05.

[36] Improving Fault Tolerance with RPC Fallbacks in DoorDash's Microservices. https://doordash.engineering/2022/06/07/improving-fault-tolerance-with-rpc-fallbacks-in-doordashs-microservices/, 2022. Accessed: 2021-05-21.

[37] Litmus. https://litmuschaos.io, 2022. Accessed: 2022-06-05.

[38] resilience4j. https://resilience4j.readme.io/docs/examples, 2022. Accessed: 2022-06-05.

[39] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications* (*SOCA*), pages 44–51, 2016.

[40] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 17–28, New York, NY, USA, 2016. Association for Computing Machinery.

[41] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 331–346, New York, NY, USA, 2015. Association for Computing Machinery.

[42] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[43] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[44] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, and Theo Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018.

[45] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 281–294, New York, NY, USA, 2012. Association for Computing Machinery.

[46] Phiradet Bangcharoensap, Akinori Ihara, Yasutaka Kamei, and Ken-ichi Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *2012 Fourth International Workshop on Empirical Software Engineering in Practice*, pages 10–15, 2012.

[47] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. Automating chaos experiments in production. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice* (*ICSE-SEIP*), pages 31–40, 2019.

[48] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[49] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. Fig: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*. Citeseer, 2002.

[50] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 433–436, New York, NY, USA, 2007. Association for Computing Machinery.

[51] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories* (*MSR 2010*), pages 31–41, 2010.

[52] Cleber Jorge Lira de Santana, Brenno de Mello Alencar, and Cássio V. Serafim Prazeres. Reactive microservices for the internet of things: A case study in fog computing. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1243–1251, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[54] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[55] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[56] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4), September 2014.

[57] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, 2016.

[58] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[59] Christina Terese Joseph and K Chandrasekaran. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Software: Practice and Experience*, 49(10):1448–1484, 2019.

[60] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, 44(6):110–120, June 2009.

[61] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, April 2007. USENIX Association.

[62] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[63] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131:106449, 2021.

[64] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Paul D. Marinescu and George Candea. Lfi: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 379–388, 2009.

[66] Caitie McCaffrey. The verification of a distributed system: A practitioner's guide to increasing confidence in system correctness. *Queue*, 13(9):150–160, dec 2015.

[67] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 388–402, 2021.

[68] Nabor C. Mendonca, Carlos M. Aderaldo, Javier Camara, and David Garlan. Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 114–124, 2020.

[69] Fabrizio Montesi and Janine Weber. From the decorator pattern to circuit breakers in microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1733–1735, New York, NY, USA, 2018. Association for Computing Machinery.

[70] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):3–15, 2020.

[71] Aashay Palliwar and Srinivas Pinisetty. Using gossip enabled distributed circuit breaking for improving resiliency of distributed systems. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 13–23, 2022.

[72] Aurojit Panda, Mooly Sagiv, and Scott Shenker. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 30–36, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Riccardo Patriarca, Johan Bergström, Giulio Di Gravio, and Francesco Costantino. Resilience engineering: Current status of the research and future challenges. *Safety Science*, 102:79–100, 2018.

[74] Dewmini Premarathna and Asanka Pathirana. Theoretical framework to address the challenges in microservice architecture. In *2021 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 4, pages 195–202. IEEE, 2021.

[75] Jesse Robbins, Kripa Krishnan, John Allspaw, and Thomas A. Limoncelli. Resilience engineering: Learning to embrace failure: A discussion with jesse robbins, kripa krishnan, john allspaw, and tom limoncelli. *Queue*, 10(9):20–28, sep 2012.

[76] Casey Rosenthal and Nora Jones. *Chaos engineering: system resiliency in practice*. O'Reilly Media, 2020.

[77] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. Service mesh circuit breaker: From panic button to performance management tool. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, page 4–10, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, Vancouver, BC, October 2010. USENIX Association.

[79] Kridanto Surendro, Wikan Danar Sunindyo, et al. Circuit breaker in microservices: State of the art and future prospects. In *IOP Conference Series: Materials Science and Engineering*, volume 1077, page 012065. IOP Publishing, 2021.

[80] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[81] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015.

[82] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. O. Ocharán-Hernández. Patterns related to microservice architecture: a multivocal literature review. *Programming and Computer Software*, 46(8):594–608, 2020.

[83] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Aakash Ahmad, and Ali Rezaei Nassab. On the nature of issues in five open source microservices systems: An empirical study. In *Evaluation and Assessment in Software Engineering*, pages 201–210. 2021.

[84] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Trans. Comput. Syst.*, 28(1), aug 2010.

[85] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association.

[86] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.

[87] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 458–472, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[89] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[90] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering* (*PROMISE'07: ICSE Workshops 2007*), pages 9–9, 2007.