Resilient Microservice Applications, by Design, and without the Chaos

Christopher S. Meiklejohn

CMU-CS-TODO

(1/17/2024)

Software and Societal Systems
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Heather Miller, Chair
Claire Le Goues
Rohan Padhye
Peter Alvaro, University of California, Santa Cruz

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: fault prevention, fault injection, fault tolerance, microservice architectures, microservices, testing, circuit breakers, remote procedure call, RPC

Abstract

Fault injection testing is vital for assessing the resilience of distributed microservice applications against infrastructure and service failures. Typically performed in production settings, this testing can risk customer experience and may overlook issues, particularly infrequent ones or those affecting only a subset of users. Although academics recognize these detection challenges, their research is often limited by restricted access to industrial systems, leading to solutions that may not fully align with industry needs.

This dissertation demonstrates that latent bugs in microservice applications – dormant until activated by infrastructure or downstream dependencies failures – can be effectively identified *during development and before deployment* through a developer-centric approach aligned with industrial practices.

This dissertation begins by constructing a microservice application corpus and introducing a tracing technique to capture all inter-service communications. Combined with the corpus, this technique leads to the development of a comprehensive fault injection technique designed explicitly for microservice environments. The method is refined by implementing a test case reduction technique to minimize redundant fault injection scenarios. The practicality of these techniques is validated using a case study taken from an industrial microservice application: a large food delivery service in the United States.

This case study confirms the fault injection technique's effectiveness and uncovers areas for enhancement. These insights inform further refinement, increasing the technique's relevance and usability for industrial microservice practitioners. Additionally, this process enriches the microservice corpus with more realistic industrial application behaviors, addressing a significant gap in existing research. The dissertation also showcases a successfully mechanized approach in industrial settings for identifying resilience-related bugs in real-world applications.

Thus, this dissertation makes substantial contributions to the field of microservice resilience, paving the way for future research in microservice systems. It offers valuable methodologies and insights for advancing the study of microservice applications.

For `argv0`.

Acknowledgements

"This has all been wonderful, but now I'm on my way."

Anastasio & Marshall, *Down with Disease*

First, I want to thank my parents, Deborah (*i.e.*, Meiklemom) and Gordon for first introducing me to the Commodore 64 at the age of (probably) 4, and their continued support when I quit college to take an "Internet" job. Especially, I want to thank them their continued support when I quit my senior software engineering job to go to graduate school and the many Hitchcock marathons, every Christmas on the Criterion Channel: essential for avoiding the ever-present threat of graduate school burnout. Most notably, they only complained once that I spent all summer on IRC as a teenager – when they threatened to send me to a summer camp which, thankfully, never happened – if it had, I would not be writing this today. Thanks to my sister, Haley, for all of the laughs and bad 80's music videos we have shared over the years.

Greg and Laura, what can I say? You both have been in my life almost as long as I have been working on computers professionally. You've both been great friends, and I hope that we can continue to rock out to many live Goose shows together for many more years to come everywhere between Boston and Pittsburgh, and beyond.

Thanks to all of my former colleagues at Basho Technologies who both mentored and pushed me from a mere JavaScript developer to a accomplished, published Erlang developer: including, but not limited to: Joe Blomstedt, Russell Brown, Tanya Cashorali, Sean Cribbs, Reid Draper, Joe Devivo, Bryan Fink, Andy Gross, Jon Meredith, Jared Morrow, John Muellerleile, Tom Santero, Justin Sheehy, Seth Thomas, Steve Vinoski, Jordan West, Ryan Zezeski, any many more. We had some extreme times, we worked insanely hard, but never once did it feel like work. Not only did you teach me distributed systems, you taught me to build robust, tested, distributed systems and that formed the foundation of this dissertation. You also gave me the first opportunity to speak, on a large stage, on what I was working on, giving me the visibility I needed to start a blog and actually pursue my Ph.D.

Thanks to everyone who was tangential to Basho Technologies in the actor space: Jamie Allen, Reuben Bond, Jonas Boner, Sergey Bykov, Roland Kuhn, Philipp Haller, Martin Odersky, and David Pollak. Thanks to everyone at Boundary: C. Scott Andreas, Cliff Moon, Kyle Kingsbury, and many others.

Thanks to everyone at Erlang Solutions (and beyond, but in arms reach) for support during my dissertation. Thanks to Francesco Cesarini, John Hughes, Konstantinos Sagonas, Eric Stenman, and Ulf Wiger for their support during my testing journey with all things Erlang. Thanks to all of the organizers of Erlang Workshop, Erlang User Conference, and Erlang Factory for helping me get established as a conference presenter, instrumental in my development as a conference speaker.

Thanks to everyone in the Clojure "house": but, most notably Chas Emerick, and Zach Tellman. Zach, you've been a great sounding board for ideas and have always helped me critically think through ideas.

Ben, thanks for all of the great times in Seattle (Pearl Jam!) the great meals, and just generally being a great friend.

Thanks to everyone in the SyncFree consortium for teaching me how to do industrial research, but especially: Carlos Baquero, Annette Bieniusa, Carla Ferreria, Marc Shapiro, Nuno Pregucia. Carla, I am sorry that I still have your copy of TAPL, and I promise to return it to you personally if I ever return to Lisbon for a conference. Thanks to all of my fellow graduate student colleagues: Deepthi Devaki Akkoorath, Manuel Bravo, Maryam Dabaghchian, Zhongmaio Li, and many others.

Ale, my SyncFree "workshop brother", this would not have been possible without you. Our times together across Europe and throughout several Microsoft Research internships was one of the highlights of my entire Ph.D. experience and, despite the constant friendly bickering, are undoubtedly some of the best moments of my life. I only wish I could celebrate my Ph.D. defense along the banks of the Seine with wine, cheese, and bread with you, as we did for your Ph.D. defense.

Thank you to both my mentors and collaborators I had during my time at Microsoft Research: Tom Ball, Phil Bernstein, Sebastian Burckhardt, and Jonahthan Goldstein. I think quite fondly of our summers together as some of the best times I had during my Ph.D. experience. Phil, if you are reading this, I still owe you a copy of your own book that I destroyed with a ill-placed cup of water.

Thanks to all of the academic folks along the way who helped me out either prior to or after I started my Ph.D., but most especially: Peter Bailis, Neil Conway, Pat Helland, Joe Hellerstein, Lindsay Kuper, and Jean Yang.

Thanks to so many folks at Berklee College of Music for giving me the chance to be a programmer: Cliff Anderson, Chris Giroir, Matt Horan, John Mileham, Patrick McNeill, and Robert Green. Thanks to my colleagues at Swipely for bootstrapping and supporting my journey on software testing and getting me to my first software

conerence: Anthony Accardi, Barnaby Claydon, Bright Fulton, Matt Gillooly, and Simon Hojberg.

Thanks to all of my current colleagues for their support throughout my Ph.D.: Moustafa Aly, Matt Anger, Armando Canals, Maggie Fang, Justin Lee, Venkataramanan Kuppuswamy, Aaron Livingstone, Ivy Liu, Amy Lu, Lev Neiman, Ivan Radakovic, Patrick Salami, Abhishek Sharma, Radha Krishna Ratnala, Ryan Sokol, Matt Zimmerman, and many others. Special thanks to Cesare Celozzi for his mentorship, help with understanding how all the pieces matter, and being such an great colleague and friend over the past two years.

Thanks to Sargun Dhillon, Caitie McCaffery, Tyler McMullen, Genevieve Patterson, Ines Sombra, and Ashley Williams for their help during my dissertation.

Thanks to my first set of Ph.D. advisors, Peter Van Roy and Rodrigo Rodrigues. While the situation may not have been logistically ideal, you helped to guide me to the right path on how to finish this Ph.D. Thanks to Matthias Felleisen for being a supportive mentor, who I had no affiliation with, but was always open to talk about interesting ideas and support me during my Ph.D journey.

Thanks to all of my colleagues at Carnegie Mellon University: specifically, Jonathan Aldrich, who helped get me into program analysis; Ben Titzer, for all of your horrible sci-fi movie recommendations; Elizabeth Gilbert, for asking good questions about fault injection; Matthew Weidner, for carrying the CRDT torch after I burned the bridge that I crossed with it; and Michael Hilton for all of his help on becoming a better teacher. Thanks to all the staff, but most notably: Aaron Caldwell, Connie Herold, and Dabney Schlea.

Zeeshan, boy, we we had some times. Thanks for being there as a sounding board for many years during my period with our late night "Butterjoint" sessions. I wish you the best in your own Ph.D. and wish you great success.

Thanks to Michael Issac Assad, Eunice Chen, Andrea Estrada, Lydia Stark, Yiwen Song, Haoyang Wu, and Peter Zhong for their work on Filibuster. Michael, your work on extending Filibuster for database testing helped make Filibuster stronger and improve its own foundations. This is work that will go on to have significant impact.

Matt, thanks for being such a great friend and colleague. Obviously, and as you know, before we were colleagues we had some great times together. I still recall one night many years ago, in San Francisco, when you were pondering if "parallel grep" was a good interview question and asking everyone if they could solve it. Since then, we recorded music together, debugged distributed applications together, and now work on keeping those distributed systems reliable through testing and improved microservice designs.

Thanks to my committee for their support: Rohan, Claire, and Peter. Rohan, you carried to torch on helping get Distributed Execution Indexing over the line and I

appreciate that. Claire, you helped to solidify things and keep my thesis grounded (and, within time, I hope.)

Peter, I'm not even sure what to say. Our relationship started as two random folks who met at a bar in the Tenderloin for a beer because a mutual friend said we should. Since then, it's progressed from friend, to almost being your first student, to you being a member of my thesis committee. We have spent many hours talking both in bars and on the phone about distributed systems, to discussing research directions, and finally my dissertation. I consider you a great friend and one of the best memories in my life is meeting you at Magnolia in the Haight for lunch and hanging out in the Panhandle after talking nothing but crazy distributed systems research ideas. I only hope to continue our relationship for many more years.

Last, and (obviously) not least, my advisor Heather. Heather, we have known each other for a very long time now: even before you were a professor, at summer school in Oregon, ten years ago! First off, I cannot thank you enough for getting me involved in Curry On organization, which helped me to meet many people who were very valuable to both my industrial and academic career(s). When I finally quit Basho due to bad circumstance, you helped me find possible job opportunities and when my first Ph.D. was not going well, you helped me find a way out by getting me admitted to Northeastern. Then, we ended up at Carnegie Mellon, and when I arrived at Pittsburgh, you and Daniel offered me a place to stay until I was able to find my own footing. Since then, we have had a lot of laughs and have done a lot of valuable, high-impact, work and I cannot thank you enough. I would not be here if it was not for you.

To everyone else not mentioned here: I apologize. It has been 10 years since I started down the path of graduate school, and a decade is a long time where one does not remember everyone who contributed to one's life. I barely remember everything that has happened myself.

Contents

Chapter 1

Introduction

"Without love in the dream, it'll never come true."

Garcia & Hunter, *Help On The Way*

Microservice architectures are currently the dominant architectural style for most consumer products and services [HSS23; Hil23; RJ20a; Hig20]. This style involves breaking down applications into various services, addressing the needs of large-scale development organizations. The primary aim is to enable developers to rapidly iterate on complex applications, often comprising millions of lines of code, while working independently at their own pace [Cel20].

Structurally, microservice architectures resemble monolithic ones, where the application is divided along module boundaries, termed *services*. In microservice architectures, these services interact through Remote Procedure Calls (RPC) across a network instead of method invocations within a single codebase as in monolithic architectures. A critical difference introduced by microservice architectures is the concept of *partial failure*. This occurs when a service the application relies on is down or unavailable during a customer request. Developers of these applications must, therefore, consider and manage these failure scenarios [Deu94; Wal+96].

To assess how microservice applications behave under fault, industry practitioners increasingly use coarse-grained fault injection experiments on microservice applications deployed in production, such as chaos engineering [RJ20a]. This process involves introducing low-level, infrastructure-specific faults (*e.g.,* dropping network connections or crashing nodes) into the application and monitoring application metrics to see if these faults negatively affect the customer experience. However, this approach has several significant drawbacks:

1. ***Latent Defects Admitted.***
   First, this approach admits latent defects into an application that will be de-

ployed into production. These defects may then later be activated and adversely affect the customer experience before discovery through fault injection experimentation if performed at all.

2. ***Cost-Prohibitive Experimentation.***
Second, with microservice applications growing to hundreds to thousands of services, with single end-to-end customer requests containing upwards of 100 RPCs, exhaustiveness becomes cost-prohibitive, as most approaches used by practitioners today rely on manually designed and executed experiments.

3. ***Ambiguous "Correct" Application Behavior.***
Third, and finally, any approach based on experimentation where metrics are observed *for bug detection* will be insufficient when the metrics are not designed specifically to capture possible latent application bugs. In short, coarse-grained application metrics cannot be used as a replacement for actual assertions on the expected behavior of the application.

These problems are far from understood by academic researchers; however, previous research on solving these problems does exist. To demonstrate, one example is provided for each of the drawbacks of the experimentation approach.

1. ***Latent Defects Admitted.***
Heorhiadi *et al.* [Heo+16a] explored testing microservice applications, in their entirety, in a development environment where faults could be injected freely to identify resilience bugs in microservice applications. However, their approach required that application developers specify *internal* behavior of the system under fault: for example, how many times a request would be retried under fault. In short, instead of verifying application behavior as a result of the fault, the application is checked for both the presence and correct operation of internal resilience mechanisms: it leaves open the question of what the application does when these resilience mechanisms do not work (*i.e.*, RPC retry fails.)

2. ***Cost-Prohibitive Experimentation.***
Alvaro *et al.* [ARH15a] proposed a smarter approach to exploration where, using a specification of the application behavior, fault injections would be targeted at locations in the application code likely to trigger resilience mechanisms and expose application bugs. While this approach was mechanized at scale [Alv+16a], it still relied on specifications written in a specific specification language outside of the reach of application developers working on new application features. In short, it shifts the burden of manually creating experiments to creating the specifications that will be checked.

3. ***Ambiguous "Correct" Application Behavior.***
   Zhang *et al.* [Zha+19] proposed an approach whereby each location where an RPC was made would be automatically subject to a fault injection through automatic test generation. However, this approach relies on the application developer specifically encoding, for each test, if the local application state in the service itself would either be the same, different, or some derivative state when a fault was not injected. In short, if no local application state change were performed due to RPC execution, these faults would have no impact, making it possible to miss latent application bugs during testing.

What this previous research, and most of the research on the majority of microservice architectures, suffers from is the lack of an industrial corpus to ground the research in how actual industrial microservice applications are designed, developed, maintained, deployed, and operated [Jam+18a; Was+21a]. Therefore, to move the research in microservice application resilience forward and show that it is possible to identify latent application bugs related to resilience in development and before deployment, this dissertation advances the previous work in microservice application resilience through both the design and implementation of a novel resilience testing technique rooted in actual industrial resilience bugs and a co-evolution of that technique with an industry partner, Foodly, which operates a large microservice application comprised of over 500 services behind one of the largest food delivery service in the United States [Mic23a].

1.1 Thesis Statement

The main goal of this dissertation is to identify latent application bugs, related to the resilience of microservice applications, *in development, and before deployment* of code to production. This should be performed automatically and without requiring that application developers author a specification, written in a specific specification language, for the purposes of resilience testing.

The work in this dissertation contributes towards this by taking a *developer-centric* approach. First, developers should be able to leverage their existing functional test suite — the test suite they are already writing when building new features — to identify bugs using fault injection with minimal additional effort; no other tests or specifications must be required. Second, when injected faults provoke functional test failures, application developers should only be required to think of the test failures in terms of the impact of the fault on application behavior, for example, as derivations of the fault-free behavior of the application. Third, faults should be automatically injected without requiring that application developers manually craft

fault scenarios, avoiding situations where the developer "misses" critical scenarios due to the overhead in test creation. Fourth, application developers should only have to run a minimal set of tests for fault scenarios that represent unique, interesting cases. Fifth, and finally, application developers should be required to think about the consequences of all injected faults: for example, when a fault has no (observable) impact, the developer must specifically indicate this is the case to avoid introducing bugs that are not immediately visible during execution, but may later have some effect. This naturally leads to the following thesis statement:

> ***Thesis Statement:*** *Identification of latent application bugs related to resilience in microservice applications, in development, and before deployment of code to production, is possible using a developer-centric approach and can surface critical application bugs in large-scale, industrial microservice applications.*

1.2 Contributions

To support this statement, this dissertation makes the following contributions:

- **Microservice Application Corpus.**
  To ground this dissertation in industrial applications and their bugs, first, a corpus is constructed from publicly available information on the internet concerning resilience bugs in industrial microservice applications.

- ***Distributed Execution Indexing*.**
  Next, an algorithm called *Distributed Execution Indexing* (DEI) is presented for uniquely and deterministically identifying all of the RPCs executed by a microservice application. This algorithm is fundamental to any fault injection approach, as it guarantees that under repeated (re-)execution of an end-user request, any resulting RPC to services inside the microservice application can be identified the same way.

- ***Service-level Fault Injection Testing*.**
  Next, a fault injection technique called *Service-level Fault Injection Testing* (SFIT) is presented that leverages the existing functional test suite of a microservice application to identify latent bugs related to application resilience. SFIT leverages DEI as its underlying fundamental technique for performing the necessary exhaustive search for demonstrating that applications are free from latent bugs.

- ***Encapsulated Service Reduction* for SFIT.**
  Then, SFIT is improved upon by applying a dynamic test case reduction strategy, called *Encapsulated Service Reduction* (ESR), that leverages how microservice applications are designed, *structurally*. ESR is then shown to make a significant performance improvement in the execution time of SFIT.

- **Industrial Evaluation of SFIT.**
  From there, a case study is performed at Foodly that demonstrates the weaknesses in the SFIT approach. However, despite these weaknesses, SFIT is shown to identify actual bugs in Foodly's large-scale industrial microservice application.

- ***Principled Service-level Fault Injection Testing*.**
  Next, a new testing process is presented, *Principled Service-level Fault Injection Testing* (*p*-SFIT), which improves on the original design and testing process of SFIT that avoids the weaknesses presented by ensuring that developers know precisely when faults are injected and encode specifically what their application should do when faults occur.

- ***p*-SFIT Tutorial.**
  Then, a tutorial is presented on how to apply *p*-SFIT to an application modeled after an industrial application scenario discovered containing a bug at Foodly.

- **Filibuster: Implementation of SFIT and *p*-SFIT.**
  Finally, Filibuster, the open-source implementation of both SFIT and *p*-SFIT is presented. Filibuster is implemented in both Python and Java; supports Google's gRPC and Netty's HTTP for RPC; has an associated IntelliJ IDE plugin, and is currently in use by many developers daily at Foodly in their development and continuous integration workflows.

1.3 Outline

The remainder of this dissertation is structured as follows:

- Chapter 2 presents both background material on microservice applications and work directly related to this dissertation. Background material focuses on microservice architectures and fault injection, each with a rich history. Related work targets fault injection in microservice applications and distributed data systems to compare the history and techniques between both types of distributed applications to highlight the deficiencies inherent in microservice architecture research.

- Chapter 3 presents two motivating examples, Audible and Netflix, that demonstrate the impact of failures on microservice applications and one way that microservice applications handle those failures. In the case of Audible, any user request fails if any of the dependent microservices are down: these are referred to as *hard* dependencies. In the case of Netflix, fallbacks are used to "compensate" for failure if a required service is non-responsive or unavailable; otherwise, the request fails: these are referred to as *soft* dependencies and are used to provide graceful degradation in the event of failure.

- Chapter 4 presents this dissertation's first of two foundational components: a microservice application corpus. This corpus, constructed from publicly available information on industrial microservice applications, contains eight (8) small synthetic examples demonstrating common microservice request patterns and four (4) recreations of industrial microservice applications where coarse-grained fault injection experimentation was used to identify, or reproduce an application bug related to resilience. This corpus is foundational because it addresses the deficiencies in academic work on microservice resilience: a lack of applications to evaluate new techniques.

- Chapter 5 presents the second of two foundational components of this dissertation: an algorithm for identifying RPCs in a microservice application. This algorithm is foundational, as it enables any fault injection technique for microservice applications to guarantee that an application has been tested for all possible faults that might occur concerning its downstream dependent services.

- Chapter 6 presents *Service-level Fault Injection Testing* (SFIT), a technique for testing microservice applications to identify latent application bugs *in development* before deployment of application code to production. SFIT is one of two core contributions of this thesis and addresses several of the notable drawbacks with existing academic techniques by taking a *developer-centric* approach. SFIT is then optimized using *Encapsulated Service Reduction* (ESR). This test case reduction technique avoids redundant fault injections, where the application has already executed the same application behavior as a result of another fault injection.

- Chapter 7 presents an evaluation of SFIT on the synthetic microservice application corpus presented in Chapter 4. It is demonstrated that SFIT can improve application coverage by exercising RPC-failure related error handling code paths and discovering all bugs seeded into the synthetic application corpus.

It is demonstrated that SFIT with ESR outperforms SFIT without avoiding the execution of redundant test cases.

- Chapter 8 provides an overview of the industrial microservice application behind Foodly, one of the largest food delivery services in the United States. This section shows how Foodly tests its microservice application, deploys changes, and remains resilient to faults of downstream microservice dependencies.
- Chapter 9 presents an evaluation of SFIT on the industrial microservice application behind Foodly. It is shown that while SFIT can be used to identify resilience bugs in industrial microservice applications, these may remain undetected if developers do not follow the SFIT testing precisely and investigate both all test failures induced by fault injection and tests that do not fail when fault injection is not applied. The results of this evaluation are then used to identify changes in the SFIT process necessary to identify these bugs.
- Chapter 10 presents a new application for contribution to the microservice application corpus, inspired by actual programming patterns used by microservice application developers at Foodly. This application demonstrates how hard dependencies are converted into soft dependencies, where microservice applications employ graceful degradation to avoid suffering from the impact of faults in production at runtime.
- Chapter 11 presents the design of *p*-SFIT. *p*-SFIT improves on SFIT by ensuring that application developers properly encode their application's failure behavior into their functional tests using an integrated testing process with SFIT. *p*-SFIT is supported by a new implementation of SFIT with an integrated IDE plugin that provides application developers with feedback during testing on how to proceed at each step of the testing process.
- Chapter 12 presents a tutorial on using *p*-SFIT. As *p*-SFIT is mainly a testing process for application developers, this chapter demonstrates how one can use the testing process to identify an application's behavior under fault through interactive prompts and supporting technology.
- Chapter 13 presents a conclusion to this dissertation and future directions, some of which have already been and are actively being explored.

Chapter 2

Background and Related Work

> *"I have my own methods; but I am human, and every time I see someone else's films I may be tempted to try their methods instead of my own. Theirs seem so logical. I do it, and I fail. Their methods may be good, but not for me."*
>
> Alfred Hitchcock

As it is important to frame this work in the context of the state-of-the-art in microservice knowledge and fault injection techniques, this chapter presents this dissertation's background and related work.

In the background material, it is shown that while microservice applications are well understood *structurally* in academic literature, failures within a microservice application and the potential impact of those failures remain a lesser understood phenomenon that has only been recently investigated by a limit set of academic researchers. It is posited that one possible limiting factor of this is access to industrial microservice applications.

In the related work material, it is shown that fault injection testing in industry and academia is converging: where practitioners work at the most coarse level, where manual faults are injected in production to identify application bugs, academic approaches focus at a granular level where faults are automatically injected on particular messages, in development, and state derivations in applications are used to detect bugs.

This work is then put in contrast to the rich literature that exists in academia on testing distributed data systems: where techniques are mechanized, automatically applied, and used to find deep bugs in applications when all services are actually homogeneous (*i.e.*, replicas of one another.)

2.1 Background

First, it is essential to understand what the existing research in microservice architectures understands about failures in microservice applications. Then, several methods for injecting faults into microservice applications are presented.

2.1.1 Microservice Architectures

Microservice architectures primarily improve development velocity, as they decompose the application into different services that can be developed independently by small teams and incrementally deployed as necessary [HSS23; Cel20; Hil23; RJ20a; Hig20]. However, while microservice architectures have been the subject of qualitative analysis since the definition of the term in 2014 [LF14] research fails to identify faults as a significant concern for microservice applications.

Systematic Literature Reviews. Since the proliferation of applications built using microservice architectures, several systematic literature reviews and mapping studies have identified the emerging concerns and challenges in microservice application development. Most of these studies have relied on existing academic research on microservices.

Pahl & Jamshidi performed a systematic mapping study that examined 21 different publications to identify the emerging concerns of microservice architectures [PJ16]. Out of the 21 publications studied, failure is only mentioned as a concern in a single publication.

Alshuqayran *et al.* performed a systematic mapping study of the challenges in microservice applications using 33 publications [AAE16a]. They identified eight general challenges as part of their study. Of these 33 publications, 28 directly mentioned fault tolerance; these were identified using the following keywords: "fault", "failure", "recovery", "tolerance", and "healing". However, the author's analysis did not go further than identifying that faults and fault tolerance were crucial challenges in microservice applications.

Soldani *et al.* performed a systematic literature study of 51 industrial publications to identify the pros and cons involved in the design, development, and operation of microservice applications [STVDH18]. The authors identified failure as an operational concern for microservice architectures and then further identified one specific technological pattern, circuit breakers, as a tool that can be used contain failures when they occur.

Waseem *et al.* performed a systematic literature study focused on testing microservice applications comprising of 33 publications [WLS20]. Their results demonstrate that research interest in testing microservice architectures is increasing. How-

ever, they only identify a single paper that explicitly tests applications for their reliability against failure.

Li *et al.* performed a systematic literature review of 72 publications to identify the quality attributes associated with microservice architectures [Li+21a]. The authors identified availability as one of the primary quality attributes associated with microservice architectures and used this more general term, availability, to encompass reliability and fault tolerance. More specifically, they identified an increasing trend towards using fault monitoring tools and circuit breakers to detect, react to, and contain failures, thereby increasing the quality attribute of interest: availability.

Waseem *et al.* performed a systematic literature study comprising 47 publications to identify the key themes in the intersection of DevOps practices and microservice architectures [WLS20]. This study identified circuit breakers as the predominant technological pattern for dealing with cascading failures, a significant concern in microservice architectures that result in outages (10.63%, five studies.)

All these studies indicate microservice resilience has only recently become of academic interest since the inception of microservices and has not been subject to in-depth theoretical investigation. Most notably, several papers identified the circuit breaker pattern as an increasingly useful for pattern for dealing with failures. This pattern explicitly injects failures into the application to indicate the failure of a dependency that will no longer be called, further necessitating the need for fault injection testing, of which little research exists.

Qualitative Methods. As a result of the lack of access to industrial microservice applications, researchers have resorted to various qualitative methods (*e.g.*, case studies, developer interviews) to study microservice applications.

For example, O'Connor *et al.* extended a previous research study that identified the software development processes that the developers of microservice architectures used to understand the situational contexts and factors that drive process selection [OEC16]. Leite *et al.* performed a grounded theory study involving interviews with 46 professionals to understand how organizations structure themselves when building microservice applications [Lei+20]. Ayas *et al.* performed a grounded theory study to investigate the human processes involved in migrating monolithic application design to microservices [MALH21]. Sorgalla *et al.* performed a comparative multi-case study where interviews were analyzed using a grounded theory approach to identify the process(es) that were used in small to medium-sized development organizations when building microservice applications [SSZ20]. However, outside of some notable exceptions, failure does not appear to be a concern identified by these studies in any aspect.

Taibi *et al.* used semi-structured interviews with 72 microservice application developers over two years to identify bad practices in the development of microservice-based systems and understand how they overcame them. Using open and selective coding, they identified 11 microservice-specific bad smells [TLP18]. However, only one of the identified smells is related to reliability: API versioning mismatches between different services where a service that depends on another might receive data that it is not expecting.

de Toledo *et al.* used a case study approach to identify technical debt in microservice architectures, specifically related to the communications layer, through document examination and participant interviews at a company that operated 1,000 different services in their microservice architecture [Tol+19]. Their study revealed several issues in the communications layer that they believe arose with rapid, decentralized development. The authors proposed a solution to these issues and identified the risks inherent to implementing those solutions.

Wang *et al.* performed a mixed-methods study that involved 21 practitioner interviews, with a follow-up online survey of 37 participants that covered 37 different companies to identify best practices, challenges, and existing solutions to the maintenance and operation of microservices [WKR21]. While this study acknowledged that failures complicate the maintenance and operation of microservices, the authors only present recommendations on debugging techniques.

2.1.2 Fault, Failures, and Errors in Microservices

There is little academic research on microservice application faults, failures, and errors [Avi+04]. Several authors believe that this is a result of academic researcher's limited access to industrial microservice applications [Jam+18b; Was+21b]. This is demonstrated by examining the qualitative methods used in recent literature: for example, analysis of bugs reported on open-source microservice applications, interviews with industrial microservice developers, and grey literature studies on microservice testing methodology.

Waseem *et al.* performed a thematic analysis of 1,345 issues gathered from 5 open source microservice applications on GitHub to build a taxonomy of the types of problems that occur in microservice applications [Was+21b]. The authors acknowledged the limited representation of the data set: specifically, it was restricted to smaller applications that are open source, but the authors still identify that communication failures represent 119 out of the 1,345 (8.84%) issues filed. However, when first starting the application, all the identified issues are around missing or incorrect configuration. Therefore, there is no discussion of failures resulting from software defects while the system is running.

Zhou *et al.*, to design a combined trace visualizer and visual debugger for microservice applications, surveyed 16 different developers across 13 other companies using semi-structured interviews to identify the bugs that they observed in microservice applications [Zho+18a]. They identified 22 different bugs and categorized them according to a taxonomy based on several criteria: most notably, whether they are specific to the architecture of microservice applications themselves (5 of 22) or would be found in more traditional, monolithic application designs. Their taxonomy draws a strict dichotomy between internal faults, where software defects cause one or more components of the microservice application to return errors, and interaction faults, where the unavailability of a dependent service causes the application to malfunction. One area where this dichotomy is deficient is its consideration of the intersection of internal and interaction faults, an area discussed in this dissertation.

In short, these papers reflect the state-of-the-art knowledge of why microservice applications fail but only identify fewer than ten application bugs, many of which do not result in customer-affecting application outages. Finally, it is essential to note that all authors have created open-source corpora for future researchers based on their findings; this is consistent with one of the two recommendations for improving research in the area [Zho+18a; Jam+18b].

2.1.3 Fault Injection

Fault injection has a rich history in academic research. Here, work that connects to the techniques used in this dissertation is highlighted.

Orchestra, [Daw+96], a system for message-level fault injection in distributed applications, lets the application developer intercept and arbitrarily delay, drop, or transform messages. Genesis2, [JD10] a system for performing fault injection in service-oriented architectures, an early predecessor of microservices, similarly supports arbitrary message transformation along with delaying service responses. Ferrari [KKA95] simulates low-level hardware faults by injecting faults in software and was the first system to identify the benefits of fault injection within different iterations of loops.

FIG, [BST02a] LFI, [MC09a] and AFEX [BC12a] all perform library-level fault injection. FIG focuses on `glibc`. LFI first proposed using static analysis on library code to identify the possible faults an application should be tested for. While LFI implements a more advanced analysis (*e.g.*, pointer aliasing, binary analysis), the authors rely on over-approximation to avoid missing potential faults. AFEX further extends LFI with a search prioritization strategy for large applications where an exhaustive search is impossible.

Both Enforcer [ABH06] and ChaosMachine [Zha+21b] perform fault injection on the JVM. Enforcer injects checked exceptions to verify the error handling code

associated with exception handlers. Most of the innovation in this tool is to support JVM-specific exception models. ChaosMachine injects all throwable exceptions and requires that developers use test annotations to specify how faults should impact the internal state. In contrast, Fate [Gun+11] is a fault injector that abstracts the injected faults to address the state space explosion problem when exploring combinations of different faults. Its counterpart, Destini [Gun+11], is a declarative specification language over abstracted system events for writing the system's behavioral specification. Similar to Destini, LDFI [ARH15a] is an optimized search strategy that uses a similar declarative specification language. The challenges of applying LDFI at Netflix (*e.g.*, specification language, behavioral specification, deterministic replay) have also been discussed [Alv+16a].

PreFail [JGS11] lets developers inject arbitrary faults and write custom pruning strategies to reduce test case explosion. Setsudo [Jos+13] uses high-level, declarative test specifications over the system state to drive fault injection. Gremlin [Heo+16a], is a system for programmatic specification and execution of chaos engineering experiments in microservice architectures.

2.2 Related Work

This section presents research and industrial practices related to both *fault injection* and *fault tolerance*. While fault injection is the direct focus of this dissertation, it is essential to frame its discussion within the context of fault tolerance, as fault tolerance often impacts the type of fault injection techniques chosen when testing microservice applications for resilience.

This related work is also presented *in contrast* with research in the same topics performed on distributed data systems: distributed systems underlying infrastructure products and not consumer services. These systems are more easily studied due to their open-source availability, public bug history with fixes, and which typically implement either one or several well-studied distributed protocols (*e.g.*, Zookeeper and Zookeeper Atomic Broadcast) or designs (*e.g.*, Apache Cassandra and Amazon's Dynamo.)

2.2.1 Industrial Practices

Industrial fault injection and fault tolerance techniques used by the developers of microservice applications today are first presented. The reason for this is that practitioners face faults daily and are highly motivated to identify practices that have a real impact on improving their application's resilience.

Fault Injection. Industrial fault injection practices in microservice applications have been chiefly performed *in production* on a running application to determine a system's tolerance to a given fault. The collection of tools, techniques, and processes that support this is colloquially known under the umbrella term of resilience engineering: a term that has its formal roots in the safety science [Res] community. It refers to the processes organizations and communities use 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 [Rob+12; McC15] to identify resilience issues in both their applications and infrastructure. Game Days acknowledges that failure is inevitable at the scale these companies operate. 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. [Rob+12]

Chaos engineering is another resilience engineering technique, initially pioneered by Netflix when first moving to the cloud [Chab]. The first iteration of chaos engineering, Netflix's Chaos Monkey [Netb], 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 [Netc; Nete], 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 [RJ20b] practiced by many different companies, where its supported by a variety of other open source tools (*e.g.*, ChaosToolkit [Chad], ChaosMesh [Chac], ChaosBlade [Chaa], Litmus [Lit], LinkedOut [Lin]), books [RJ20b], community meetups[1], and commercial software-as-a-service (SaaS) offerings (*e.g.*, Gremlin [Grea].)

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 the entirety of, a subset of, or mirror of production traffic and the hypothesis is falsified, if possible. Therefore, the key behind the chaos engineering approach is application observation. In the case of Netflix, the key performance metric observed during chaos engineering experiments is a metric that counts the number of movie streams started per second, which varies a little week over week, making it easy to detect deviations from the norm when running a chaos experiment. This directly contrasts traditional approaches where a *test oracle* – a

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

source of truth in the test suite that is used identifying application bugs – contains assertions about the application's desired behavior.

Netflix has continued to innovate in chaos engineering. Their Failure Injection Testing (FIT) [Neta] 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 microservice application. Their Chaos Automation Platform (CHAP) [Bas+19a] 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 indicator, the experiment is automatically terminated. MONOCLE [Bas+19a] 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 run with CHAP. It is important to note that MONOCLE was recently disabled [Ret] due to the large number of experimental configurations generated and the required overhead running those experiments at scale. Finally, GREMLIN, the chaos engineering SaaS company formed by former Netflix chaos engineers [Grea], 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.

Regarding the industrial practices for fault injection in microservice applications, several interesting observations can be made. First, the adoption of chaos engineering techniques in practice seems to be related to two critical 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 [Grea] 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, using key performance indicators (KPI) or metrics, also has low overhead when compared to either heavyweight specifications describing the application's behavior – in enough detail for mechanical verification – or comprehensive test suites, under fault. These two key aspects indicate that developers can quickly "try out" chaos engineering before moving to more advanced techniques for fault prevention, which has presumably helped increase the broad 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 [Bas+19a] automatically generates chaos experiments from software configuration: this resem-

bles a traditional exhaustive or systematic search commonly found in approaches built on some form of model checking. ALFI, the abandoned approach from Gremlin [Grea], sought to provide fault injection in the libraries that services used for issuing RPCs and communicating with distributed data systems 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 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 [Grea], 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 microservice application. 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 critical 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, to bring this experimentation style into the local development environment, one must first solve the problem of the missing test oracle.

Fault Tolerance. When it comes to fault tolerance, developers typically rely on a set of techniques specifically designed for microservice application's, in addition to the standard retries and timeouts often used by the developers of distributed data systems. 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 case of a fault, potentially with a degraded user experience.

Circuit breakers are also used when a remote service is malfunctioning or unavailable to relieve pressure on the remote service and to avoid waiting for a resource that will not respond promptly. 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, to close the circuit once the remote service begins functioning correctly, an RPC is allowed to happen. Eventually, once the remote service fully recovers, the circuit moves back into the closed state.

Load shedding is used when a remote service is overloaded and cannot respond promptly to reduce pressure on that service. Where circuit breakers are located at the invocation site of an RPC, load shedding is situated 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 whose 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.) If a circuit breaker is in place, These requests typically cause the invokers circuit breaker's counters to increment.

These three techniques are not a panacea of fault tolerance, however. For example, 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 turn off 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 must also consider faults within the fault tolerance measures employed by the application.

For most microservice applications, a combination of these three techniques, along with timeouts and retries, are used to prevent 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.

2.2.2 Academic Literature

This section presents both the fault injection and fault tolerance techniques for microservice applications that have been the subject of academic study. The related

work on fault prevention in distributed data systems is also presented to place this in the proper historical context.

Fault Injection. Academic research on fault injection testing and fault tolerance in distributed data systems 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 [ARH15b; Kil+07; Luk+19; Yab+10; Yan+09; SBG10; Lee+14; GY11]. However, successfully applying these techniques to microservice applications has been limited [Alv+16b]. Outside of mechanizing an approach that supports deterministic fault injection across multiple services implemented in various languages, one of the main challenges has been the lack of a test oracle that specifies behavior under failure. In industrial microservice applications specifically, specifications that are rich enough to support model checking are rarely if ever, written; similarly, the decentralized nature of microservice application development prevents the use of global state invariants placed across all stateful services in an microservice application. However, the developers of industrial microservice applications are writing functional tests. Therefore, it would seem that any successful approach should built on and extend to cover behavior under fault, the test oracles already being written.

One key observation about microservice applications is that intra-service communication is typically performed using client libraries (*e.g.,* AWS DynamoDB client) or RPC frameworks that are directly built on and/or extended 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 (*e.g.,* FIG, LFI, AFEX) [BST02b; BC12b; MC09b], which purports that low-level faults will manifest themselves as library-level errors in an application, may be a helpful starting point for automated fault prevention techniques for microservice applications. There is evidence of this: ChaosMachine [Zha+19], for example, uses a library-level fault injection approach to exercise and test an microservice applications exception handlers. Library-level fault injection strikes a good middle ground where library-specific errors can be simulated automatically and exhaustively without requiring 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, the academic system sharing the same name as an industry service Gremlin [Heo+16b] 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 [PSS17] proposes the use of software-defined networking (SDN) infras-

tructure 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 injecting specific faults, for instance, triggering a gRPC failed precondition error instead of 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 more challenging to inject certain types of faults and more costly in computing resources and required infrastructure.

Despite advances in the fault injection methodology, the problem of the missing test oracle remains unsolved. ucheck [PSS17], 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 microservice applications. In contrast, Gremlin [Greb], which also operates at the network layer but instead uses sidecar proxies for fault injection, has no visibility into the system state and only provides assertion languages over the request patterns between different services: this requires developers, in addition to writing end-to-end functional tests also encode how communication occurs. Finally, ChaosMachine [Zha+19] 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 toward commonly used industrial techniques (*c.f.*, chaos engineering). However, it remains somewhat disconnected from typical functional testing. A promising natural progression of this research is to explore the 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 distributed data systems specifically, test case reduction has typically relied on properties of the system under test: for example, symmetry reduction [Luk+19], 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 microservice applications [Jam+18a; Was+21a; Mei+21a], and the deficiencies in existing open-source corpora, which have not been operated at scale [Was+21a] nor contain realistic bugs specific to the choice of a microservice application [Zho+18b], have limited the ability of researchers to discover similar techniques for microservice applications. Therefore, with increased access to descriptions or implementations of realistic microservice applications and actual bugs experienced in microservice applications, the discovery of test case reduction techniques will be possible.

Most notably, and most recently, Rainmaker [Che+23] proposes an approach

closest to the innovations that are presented in this dissertation. However, instead of focusing on microservice applications, RAINMAKER's focus is on testing applications that use cloud services using HTTP APIs, leveraging the application's test suite for a test oracle, and employing a test case reduction strategy. This meets both the goals of a cheap library-level fault injection technique, where all HTTP calls can be uniformly instrumented using a network proxy with no application modification, and avoids the heavy requirement of test oracle creation.

Towards these goals, RAINMAKER makes several simplifying assumptions, thereby achieving an automated fault injection technique that can be used to surface bugs quickly in these applications. As these assumptions are relevant to this dissertation, they are briefly discussed:

- *Application Structure.*
  In order to reduce testing cost, redundant test cases are eliminated through an analysis. However, these redundant test cases are identified based on the application call site of a network call and under the assumption that the error handling code for that RPC is co-located with the RPC's invocation. In short, it is only necessary to test each individual RPC a single time to determine if failure of that RPC is handled properly. As will be demonstrated in this dissertation, this assumption is not true for microservice applications and two of the bugs identified during the development of this thesis were precisely in non-localized error handling.

- *Test Oracle.*
  To avoid the high cost in creating a failure-specific test oracle [AMS13] an application-agnostic oracle is used instead. This oracle assumes that any exception generated by the application code itself is not indicative of a bug as compared to an exception generated by a underlying framework code.) This is based on an assumption that application-thrown exceptions indicate that the exception was handled and a specific application error was thrown. Not only that this may not be true for microservice application code where the line between framework code and application libraries developed by large organization(s) to support development is blurry, it also fails to acknowledge silent failures: where something does not happen that should have but did not cause the test to fail. As will be demonstrated in this dissertation, this maps to an entire class of application behavior known as *graceful degradation*, where silent failures can adversely affect end users.

RAINMAKER's evaluation of these techniques is performed on open-source library code, found on GitHub by researchers. This further emphasizes the need for open-source corpora to further fault injection research.

Fault Tolerance. Research on microservice application specific fault tolerance is quite limited, again presumably due to the lack of access for academic researchers to realistic applications [Jam+18a]. While academics have seemingly studied the use, and improvement of, circuit breaker technology, none have considered it in the context of fault injection to determine what behavior an application has when a circuit breaker is open. This is despite circuit breakers provoking failures in microservice applications by design.

Several qualitative studies [AAE16b; Tig+20; Li+21b; JC19; Ner+20; Val+20; PP21; SMAP19; BHJ16; Bal+18] 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 [SS+21] identified a lack of existing research on circuit breakers compared to other microservice application topics.

Outside of qualitative studies, previous work has explored the transparent application of circuit breakers in an academic programming language [MW18] and either the dynamic tuning of [SKT21] or optimal configuration via model checking of [Men+20], a circuit breaker's configuration parameters. Palliwar *et al.* [PP22] 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. Heorhiadi *et al.* [Heo+16b] who proposed the fault prevention tool Gremlin, also considered circuit breaker testing as part of the design of Gremlin and provided a mechanism for asserting that they operated correctly, but failed to consider application behavior once activated.

Researchers have also motivated the design of data plane fault injection tools using circuit breakers; however, none of these designs contained anything specific for testing what an application does when a circuit breaker is open: a specific case where a fault is injected into an application. It is important to note that no relevant academic research could be found regarding load shedding as a fault tolerance technique in microservice applications: another case where faults are injected into an application when running by design.

2.3 Takeaways

Regarding fault prevention in microservice applications, there is an evident trend when you examine 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 library-level fault injection,

with automation for the generation of tests or chaos experiments, using granular fault injection to minimize the blast radius.

- *Library-level Fault Injection.*
  Developers realize that 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 microservice application.

- *Automated Test Generation.*
  Developers are beginning to use the application's configuration to identify remote calls and automatically create test scenarios. However, running these experiments in production, combined with the lack of test case reduction, makes running exhaustive exploration infeasible.

- *Granular Fault Injection.*
  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 not be possible to run these experiments in a local development environment where blast radius minimization is not necessary.

Academic research also seems to be evolving in the direction of contemporary industrial practices.

- *Network-level Fault Injection.*
  For example, academic research has sought to solve the problem of heterogeneous implementation language usage in microservice applications 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.)

- *Test Oracle Creation.*
  Similarly, recent academic research has acknowledged the problem of the missing test oracle, proposing solutions that build on application observation, as is familiar 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 [Greb] or whether or not the state will change and be observable by the end-user when a fault occurs [Zha+19]. Instead, unfortunately, manual experiment (*i.e.*, test) specification is still required

with many of these techniques. As a result, little research into microservice application specific test case reduction exists.

Industrial practices are influencing academic research, as clear by the innovations in mechanizing chaos engineering and tools like Rainmaker. Still, without access to implementations of industrial microservice applications, academics are left guessing at practical solutions to the most pressing of industry concerns.

This also extends to fault tolerance techniques for microservice applications. At the same time, qualitative research that relies on developer interviews, questionnaires, and surveys identified circuit breakers as a primary technique used for increasing the resilience of microservice applications, studies performed on open-source microservice applications failed to identify circuit breakers, load shedding, or fallbacks as resilience techniques used in microservice applications.

This seems to indicate that while academic research and industry practices are converging on a similar design based on observation of each other's techniques, academics, without access to industrial code bases, are left to infer the methods and solutions they feel will be helpful to practitioners.

Chapter 3

Microservices: Dependency Types

> *"We're building something here, detective. We're building it from scratch. All the pieces matter."*
>
> Detective Lester Freamon in *The Wire "...and all the pieces matter."*

This chapter presents the designs of two microservice applications to describe the types of service dependencies commonly occurring in microservice applications: *hard* and *soft* dependencies. Each example application is (re-)constructed from public discussions of each company's industrial microservice application.

Audible [Tyl18] is an audiobook streaming service. Their microservice application is used to demonstrate a microservice application that is composed of predominantly *hard dependencies*: dependencies where upon failure of an RPC, the failure forces the application to abort a customer's request with an error. While their application does contain a single *soft* dependency, its success or failure has no bearing on the customer experience of streaming an audiobook: it merely tracks internal statistics on what audiobooks customers are streaming.

Definition 1 (Hard Dependency)**.** *A hard dependency is a dependency where a failed invocation forces the request that triggered the dependency to return an error to its caller.*

Hard dependencies are specific to an invoked RPC service, method, arguments, and specific call site with calling context, as different call sites may treat the same RPC invocation as either hard or soft.

Netflix [Nor17] is a video streaming service. Their microservice application design presents an application where both *hard* and *soft dependencies* are used. With their *hard* dependencies, the failure of some RPCs forces the application to return an error to the customer. In contrast, in the case of *soft* dependencies, the failure of the RPC is ignored or "compensated" by issuing alternative RPCs.

Definition 2 (Soft Dependency)**.** *A soft dependency is a dependency where failed invocation is handled in the application to avoid the request that triggered the dependency from returning an error.*

Soft dependencies are specific to an invoked RPC service, method, arguments, and specific call site with calling context.

In both of these example applications, the effects of the type of dependency choice are directly visible to the customer when trying to use the service. For example, in the Audible example, almost all fault injections cause the customer to experience a failure. In the case of Netflix, some fault injections will cause a successful response to turn into an unsuccessful response; however, other fault injections will cause the successful response's *content to change.* This is a result of the application "compensating" for the failure by loading alternative content or omitting content that is not available.

The developers of microservice applications often prefer soft dependencies as they enable the "graceful degradation" of microservice applications: the ability for applications to reduce their functionality in the event of failure to provide still (some) service to the customer.

3.1 Audible: Hard Dependencies

The first motivating example is taken from Audible [Aud], an audiobook streaming service owned by Amazon, where the failure of a dependent service results in the customer receiving an error.

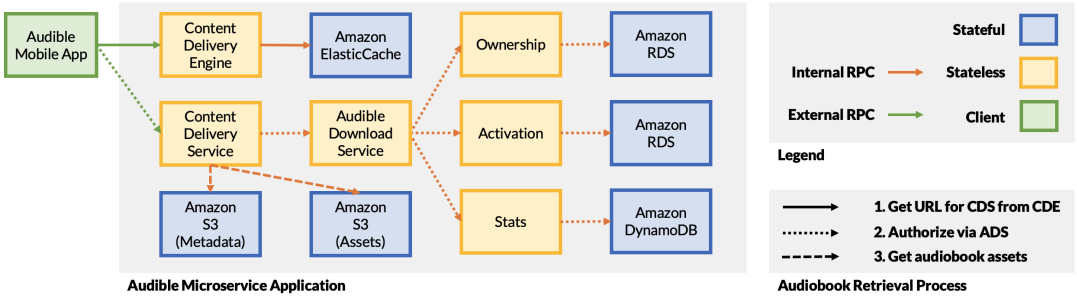

Figure 3.1: Audible application with a description of the audiobook retrieval process.

3.1.1 Application Structure

The Audible microservice application, depicted in Figure 3.1, comprises several stateful and stateless services, all of which are *hard dependencies*.

- **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 Digital Rights Management (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.

3.1.2 Application Behavior

When a customer requests an audiobook using the Audible mobile application, it first requests the Content Delivery Engine to find the URL of the Content Delivery Service (CDS) containing the audiobook assets. The app then requests that CDS. The CDS first requests the Audible Download Service (ADS). The ADS is responsible for first verifying that the customer owns the audiobook. Next, the ADS verifies that a 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 customer through an error in the client. Once the ADS completes its 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 customer's app for playback.

If *any* of the dependent downstream RPCs executed when processing a customer's request fails, modulo the Stats service, the entire request from the customer is failed and they are asked to try their request again. Now, let us examine how this can go wrong based on a real-life incident reported by Audible [Tyl18].

Outage. The Audible application assumes 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. This precise fault could be detected through fault injection testing; 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 malfunctioned at the time of the request. This resulted in a cascading failure and subsequent outage of the Audible application.

This outage results from several different faults, both latent and active, that interact 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 a failure, a generic error is provided back to the customer in the mobile application that causes the customer to issue a retry. This combination of customer-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.

In short, a *latent bug* can be defined as follows:

Definition 3 (Latent Bug)**.** *A latent bug is a bug that exists in application code but has not yet caused any observable error.*

This type of bug remains dormant due to specific conditions or scenarios not being met: the application's environment, which includes but is not limited to external dependencies (*e.g.*, other services), or invalid data being provided as an

input. In microservice applications specifically, latent bugs are hidden errors that do not necessarily cause problems initially but may lead to issues later, often after the application is already being used. In many cases, they are activated by the unavailability of downstream dependencies or infrastructure failures. These bugs are hard to spot and fix because they don't show up until specific, sometimes rare, conditions occur in the application, often once already deployed to production.

Latent bugs are in direct contrast to *active bugs*, defined as follows:

Definition 4 (Active Bug)**.** *An active bug in a microservice application is a noticeable software defect that affects the application's functionality or performance at present.*

Unlike latent bugs, active bugs are immediately apparent, often causing errors, performance degradation, or other unexpected behavior in the system. In microservice applications specifically, these bugs can impact individual services or the communication between services in a microservice architecture, leading to problems like service unavailability, incorrect data handling, or failure to execute certain requests.

3.2 Netflix: Hard and Soft Dependencies

The second motivating example is taken from Netflix [Netd], a video streaming service, where its homepage requires the resources of several dependent downstream services to populate its content. However, when a service is unavailable, they replace its content with content from another appropriate service to avoid failing on a single hard dependency: the only hard dependency in this part of their application is the customer profile service, which, when failed, prevents the page from loading. This is referred to as a *fallback*, and can be provided when making an RPC request using their fault tolerance library, Hystrix.

3.2.1 Application Structure

The Netflix example, presented in Figure 3.2, has ten services comprising hard and *soft dependencies*. Similar to the Audible example, the Netflix mobile application is represented as a service.

The services in the Netflix example are:

- **Client**:
  Simulates the mobile client;

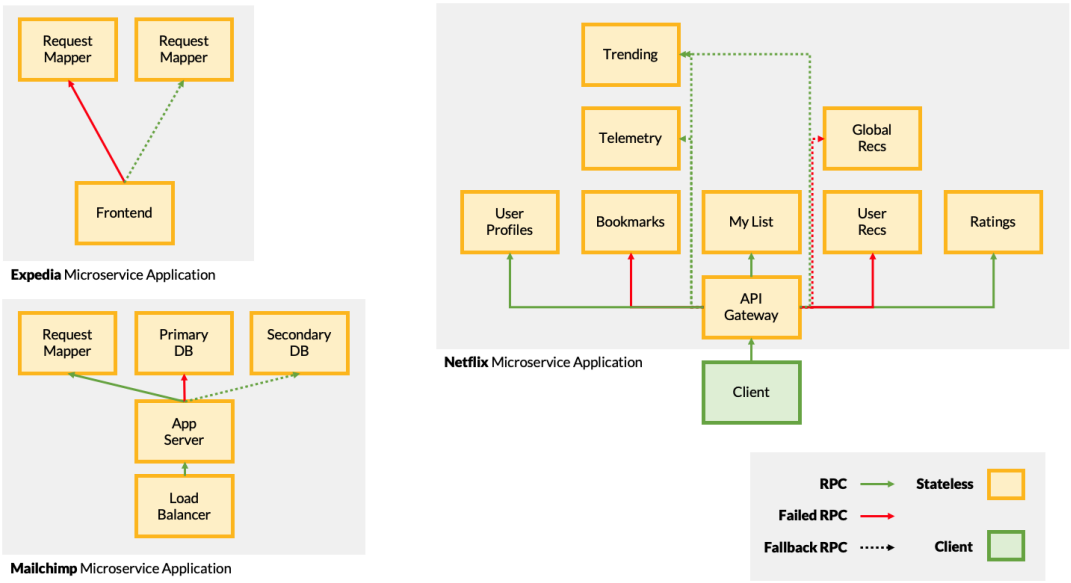

Figure 3.2: Industry examples in the microservice application corpus.

- **API Gateway**:
  Assembles a user's homepage using content from various downstream dependencies, both hard and soft;
- **User Profile**:
  Returns profile information;
- **Bookmarks**:
  Returns last viewed locations of movies;
- **My List**:
  Returns the list of movies in the user's list;
- **User Recommendations**:
  Returns recommendations specific to the user;
- **Ratings**:
  Returns the user's movie rating;
- **Telemetry**:
  Records telemetry information;
- **Trending**:
  Returns trending movies; and

- **Global Recommendations**:
  Returns recommendations for all users of the application.

3.2.2 Application Behavior

In this example, loading the Netflix homepage requires several services: a high fan-out that touches multiple services to retrieve their current locations in television shows or movies they were watching, what they want to watch, trending content, user recommendations, and their ratings. However, Netflix does *not* want to fail the entire homepage load if one of these services is down. Instead, they prefer to omit or load alternative content to provide a homepage that may not be as rich in the presence of failure.

While the list of services presented comes from several Netflix presentations on chaos engineering, the precise fallback behavior is only known to Netflix employees. However, it is known from these presentations that fallback behavior for each service exists, so liberties were taken to "re-imagine" what reasonable behavior might be in the presence of dependency failure. One such example of this is the assumption that when the user recommendation service is unavailable, the global recommendation service is used instead by showing customers what's being recommended to everyone if user-specific recommendations are unavailable. Similarly, if the customers recently watched but unfinished movies were unavailable, display movies or television shows currently trending worldwide instead.

In all these cases, providing alternative content on the homepage over showing the customer an error page is always preferable: graceful degradation in practice.

3.3 Takeaways

Classifying dependencies as either hard or soft is critical to how microservice application developers deal with partial failure in microservice applications. In short, developers prefer soft dependencies, where failures can be mitigated at runtime by ignoring or compensating for the failure with data from a different, responsive service. However, soft dependencies can often not be used; for example, if the customer's user information cannot be retrieved, it may be impossible to proceed. An example of this was presented in both the Audible and Netflix examples.

When soft dependencies can be used, this is typically called graceful degradation. In essence, it is always preferred to give the customer a slightly incorrect or incomplete answer over giving them an error, where they may turn to different vendors of the same consumer service (*i.e.*, when Uber, a ride-sharing service, is returning errors, using Lyft, an alternative, competing ride-sharing service as the

cost of switching services is relatively low.) Therefore, using soft dependencies enables applications to be highly resilient to failures that may affect customers.

Chapter 4

Microservice Application Corpus

"I really enjoy listening to stories. I remember them and keep them in my mind."

Abbas Kiarostami

The first challenge in approaching the design of a new fault injection technique for microservice applications is to understand the structure of industrial microservice applications and the type of faults that affect them. In other fields, such as software testing and distributed systems, this is typically performed using open-source applications and bug corpora. Unfortunately, this is unavailable for microservice applications.

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

As another example, most of the research into the testing of monolithic software uses bug databases assembled by the research community: collections of software projects with documented bugs harvested from open-source code repositories. How-

ever, these projects suffer from two issues that make them unsuitable for use in microservice testing:

1. These applications are monolithic in design, lacking communication over the network between different system components.

2. Many, if not all, of the bugs in the significant bug repositories contain bugs that could be identified through traditional software testing using regular unit or functional tests. Simply put, these bugs are not specific to resilience issues in microservice architectures.

Therefore, to perform this research, it was necessary to create an application corpus that could be used as the basis for research.

To create this corpus, conference talks at industry events such as Chaos Conf and AWS re:Invent were used, where it is common for industry practitioners to discuss and advocate for the use of chaos engineering. The following search terms were used to find these talks: "chaos engineering" or "resilence", "chaos", or "fault injection". Building upon this, companies that sold chaos engineering services were also identified, and their public client lists were then used to perform a reverse search for these companies to find a presentation or blog post discussing their use of chaos engineering.

In total, 50 presentations were systematically reviewed, which represented 32 different companies[1]) 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 the majority of these presentations, companies had at least one of two major concerns: concerns around the reliability of software under development and the reliability of the cloud infrastructure that the company was running its software on.

From there, the presentations were filtered using the following criteria:

1. Did the presentations detail a real bug they discovered using chaos engineering?

2. Did the presentation run a chaos engineering experiment that could have been performed locally?

[1]Provided at the following link is the list to a single representative presentation, each of the 32. Chaos engineering was used in one case, but no talk or blog post discussing its use was available: https://pastebin.com/qB7gdg45.

Finally, bugs where the bug did not occur in the application code but instead were related to incorrect cloud configuration were filtered out. These examples ranged from the incorrect configuration of authorization policies (*c.f.*, AWS IAM) to missing auto-scaling rules (*c.f.*, AWS EC2.)

In the end, four presentations were selected, which represented the following companies: Audible, Expedia, Mailchimp, and Netflix.

- **Audible** is a company that provides an audiobook streaming mobile application. In their presentation, they describe 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 to the mobile client with a generic error message. They discovered this bug using chaos engineering.

- **Expedia** 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, they will fall back to another service that provides chronologically sorted reviews.

- **Mailchimp** is a product for e-mail communication management. In their presentation, they discuss two bugs:

  1. legacy code that does not handle the case where their database server returns an error code to indicate that it is read-only; and
  2. one service becomes unavailable and returns an unhandled error to the application.

  Both of these bugs were discovered using chaos engineering.

- **Netflix** is a media streaming product. In two of their reviewed presentations, Netflix discusses the services involved in loading a Netflix customer's homepage. Netflix doesn't disclose the actual fallback behavior for each service in these presentations but instead alludes to possible fallback behavior. During corpus construction, liberties were taken as to what this behavior was.

  In one of these presentations, Netflix discusses several bugs they discovered using their chaos engineering infrastructure. These are:

  1. *Misconfigured timeouts:*
     Nested service calls aren't configured correctly to allow requests that take longer than expected, but remain within the timeout interval;

2. *Fallbacks to the same server:*
   Services are configured with fallbacks that point back to the failed service; and
3. *Critical services with no fallbacks:*
   Critical services do not have fallbacks configured.

All three of these bugs were re-created in the corpus.

In addition to these four industry examples in the corpus, the corpus also contains 8 small microservice applications, the **cinema examples**, each demonstrating a particular pattern observed in microservice applications during the survey. The cinema examples demonstrate the various implementation strategies for an application that books tickets for movie viewings may have.

Each example contains unit tests as well as functional tests that verify the functional behavior of the application. Since the functional tests were not discussed in most of the talks, a functional test was written that reflects the application's "happy path" behavior. For the cinema examples, the "happy path" test attempts to retrieve the movie bookings for a particular user.

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

4.1 Cinema Examples

A tutorial microservice application was used as the starting point for each of the cinema examples. 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 all of this behavior.

A diagram of the structure of this application and the request path for this functional test in Figure 4.1. In this diagram, the Showtimes service is not contacted but was included by the original tutorial, because that service is not involved in this functional test.

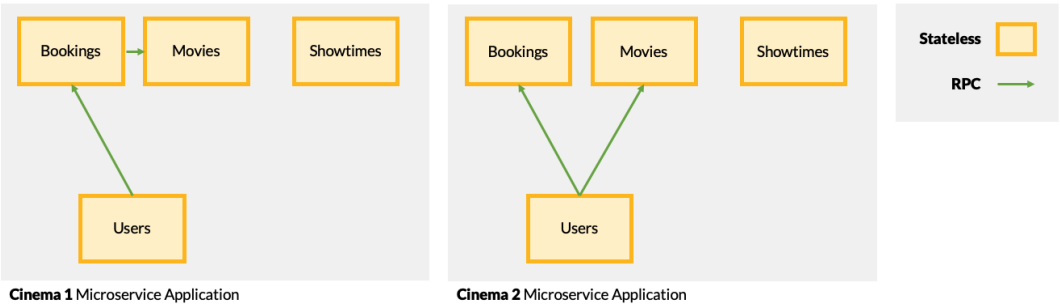

Figure 4.1: Cinema examples in the microservice application corpus.

There are 8 cinema examples. Each demonstrates a different pattern observed in microservice applications. Here, the details of each cinema example are provided, all examples are modifications to `cinema-1`, unless otherwise specified.

- **cinema-2**:
  Modifications: The Bookings service talks directly to the Movies service.
- **cinema-3**, derived from **cinema-2**:
  Modifications: The 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 requests IMDB; the Bookings service requests 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 the Bookings service, that is contacted in the event of failure of the primary replica.
- **cinema-7**, derived from **cinema-6**:
  Modifications: the Users service calls a health check endpoint on the primary Bookings service replica before issuing the actual request.
- **cinema-8**, derived from **cinema-1**::
  Modifications: The example is collapsed into monolith where an API server makes requests to it with a retry loop.

4.2 Industry Examples

This section provides a description of the four industrial examples contained in the corpus: Audible, Expedia, Mailchimp, and Netflix. These examples are not meant to reproduce the entire microservice architecture of these companies. Instead, they focus only on the services involved in a particular chaos experiment that they performed.

4.2.1 Audible

The Audible example was presented in Section 3.1. Compared to Audible's actual deployment, some of the components represented here as services are cloud services:

1. the Asset Metadata and Audio Assets services are AWS S3 buckets; To simulate this, HTTP services were created that either return a `200 OK` containing the asset if available or a `404 Not Found` if the asset isn't present.
2. The Ownership and Activation services are AWS RDS instances. To simulate this, HTTP services were created that implements 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`.
3. The Stats service is an AWS DynamoDB instance. To simulate this, I created an HTTP service that returns a `200 OK`. It was not necessary to implement a negative response as it was not required for the reproduction of the bug.

A simple test was written for the functional test that attempts to download (or stream) a user's audiobook. 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

precise bug discussed in the Audible presentation and causes a generic error to be presented to the user in the mobile application.

4.2.2 Expedia

The Expedia example, presented in Figure 3.2, has 3 services:

- **Review ML**:
  Returns reviews in relevance order;
- **Review Time**:
  Returns reviews in chronological order; and
- **API Gateway**:
  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 run.

4.2.3 Mailchimp

The Mailchimp example, presented in Figure 3.2, has 5 services:

- **Requestmapper**:
  Maps pretty URLs in e-mail campaigns to actual resource URLs;
- **DB Primary:**
  The primary replica of their database;
- **DB Secondary**:
  The secondary replica of their database;
- **App Server**:
  Requests the Requestmapper service to resolve a URL and then perform a read-then-write request to the database, with fallback to secondary database replica if the primary replica is unavailable; and
- **Load Balancer**:
  Load balances requests.

Compared to Mailchimp's actual deployment, some of the components represented as services are non-HTTP services:

1. the DB Primary and Secondary services are MySQL instances. To simulate this, an HTTP service was created that either returns a `200 OK` on a successful read or write or a `403 Forbidden` if the database is read-only.

2. the Load Balancer service is an HAProxy instance. To simulate this, an HTTP proxy service was created in Python.

For the functional test, a URL is resolved using the service. 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 unhandled in one area of the code. Since Mailchimp uses PHP, this error is rendered directly into the page output, which is simulated by turning the `403 Forbidden` response into output 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.

4.2.4 Netflix

The Netflix example was presented in Section 3.2. However, in their presentations, their fallback behavior is just provided as an example. Therefore, in the corpus, arbitrary but realistic choices are made on what the fallbacks should be seemed to reflect possible fallback behavior. In short, the specific fallback does not matter when it comes to fault injection testing to identify bugs, but rather, a reasonable fallresent.

Two examples of fallback behavior implemented in the corpus:

1. when the Bookmarks are unavailable, load Trending content instead and log error to Telemetry; and

2. when User Recommendations are unavailable, load Global Recommendations.

For the functional test, there is 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.

4.3 Takeaways

The construction of a microservice application corpus is critical in the ability to enable practical microservice research into building resilient applications through testing. As discussed in Chapter 2, the lack of an industrial corpus is believed to be a limiting factor in existing academic research on microservices. Towards this goal, this newly created corpus serves as an essential foundation in this dissertation. Additionally, it has also already been applied by other researchers towards this goal [RPA22].

Chapter 5

Distributed Execution Indexing

"Sometimes, the songs that we hear are just songs of our own."

Garcia & Hunter, *"Eyes of the World"*

At the core of any automated *exhaustive* fault injection technique for microservice applications imaginable is the need to uniquely and deterministically identify every RPC a microservice application issues. For example, in the Audible example that is presented in Figure 3.1, one must identify the RPC issued between the Content Delivery Service and the Audible Download Service both uniquely and deterministically *across all executions where an audiobook is retrieved.* This is necessary to know when the exhaustive search is complete and all possible faults have been injected.

While this may seem relatively 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 execution. The programming patterns that cause this behavior are relatively commonplace: loops, branching, function indirection, and concurrency. In this section, these programming patterns are described and used to motivate the design of a new algorithm.

To solve this problem, an algorithm called *Distributed Execution Indexing* (DEI) is presented. DEI assigns identifiers to RPCs while guaranteeing that identifiers are both unique within an execution and deterministic across multiple executions of the same code.

DEI differs from the state of the art in both industrial libraries (*e.g.,* Zipkin[1], Dapper [Sig+10], Jaeger[2], OpenTelemetry[3]) and academic (*e.g.,* Pivot Tracing [MRF18],

[1] https://zipkin.io
[2] https://www.jaegertracing.io
[3] https://opentelemetry.io

3MileBeach [Zha+21a]) proposals, as it is designed specifically for use during testing as it is higher cost, but when used, provides guarantees not available in previous systems. These differences are useful for testing microservice applications exhaustively using fault injection.

Distributed Execution Indexing (DEI) both builds on and inherits its namesake from execution indexing [XSZ08], a technique for identification of function invocations in monolithic programs. However, DEI differs from execution indexing in that rather than identifying the series of method invocations leading to a program execution point, it aims to identify the remote method invocations that lead to a particular program execution point.

5.1 Algorithm Requirements

As with all dynamic program analyses, of which an exhaustive microservice fault injection testing approach is, they must ideally be both **sound** and **complete**. For the following definitions, a fault injection analysis is assumed which systematically performs fault injection for all identified RPCs issued by a microservice application.

Therefore, we can define soundness and completeness as follows:

Definition 5 (Soundness)**.** *RPCs selected for fault injection must have faults injected on only those RPCs. Soundness is violated when a fault is injected on an RPC invocation where it was not intended.*

If not, either because multiple RPCs share the same identifier or nondeterministic assignment on repeated executions, such an approach will exhibit **unsound** behavior, which may prevent both deterministic replay (*i.e.*, debugging) and exhaustive search as *faults may be injected on the incorrect RPCs.*

Definition 6 (Completeness)**.** *RPC identifiers must be unique within a given execution and deterministically assigned across all executions.*

If not, the analysis will exhibit **incomplete** behavior, where exhaustive search may *fail to explore the entire fault space properly.*

Definition 7 (Correctness)**.** *An exhaustive fault injection analysis can only be correct if the assignment of identifiers to each RPC, used for targeting the injection of faults, is both sound and complete.*

If not, the analysis may fail to inject faults where faults should be injected (*i.e.*, false negative) and may inject faults where not intended (*i.e.*, false positive.)

5.2 Synchronous *Distributed Execution Indexing*

To motivate *Distributed Execution Indexing* (DEI), the simplest possible RPC identification method is used. From there, complexity is added to address programming patterns common in industrial microservice applications.

5.2.1 Signatures Are Too Coarse-Grained

Consider a straightforward way of identifying RPCs, used by other fault injection approaches, the RPC's signature. A RPC signature is defined as follows:

Definition 8 (Signature)**.** *A signature is a triple* (m, f, a) *where*

- 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 can be easily used to represent both of the 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 identify an RPC uniquely and deterministically.

Consider the example in Figure 5.1. In this example, a microservice application comprised of two services is presented in pseudocode. Service A exposes a single RPC endpoint, `helloworld`, which issues two RPCs to Service B's RPC endpoint, `echo`, before combining the responses and returning a response. If Service B is down, a default response is returned by the function wrapping the RPC 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))`. Therefore, an analysis using signature alone would not be able to distinguish between the first and second RPCs for fault injection; that is, the RPC signature alone is too *coarse-grained* for identifying a particular RPC.

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

5.2.2 Increasing Granularity: Invocation Count or Call Stack

Increasing the assigned identifiers' granularity is one solution for resolving the issue where identical identifiers are assigned to different RPCs. Here, the two ways this could be accomplished are examined to demonstrate that they must be used together.

In the following discussion, since the presentation goes beyond just signature-based identifiers, it is assumed (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, only the *invoking service name* (*e.g.*, `A`) is used 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 the method's formal parameters.

1. *Invocation count.*
   3MileBeach keeps track of the number of invocations for each RPC call site to distinguish multiple calls to the same call site. In Figure 5.1, the same RPC

is invoked twice. The symbol "|" indicates 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 line 10.

2. *Call stack.*
   Another approach is to increase the granularity of the identifier with some representation of the call stack. In Figure 5.1, the RPC is invoked twice at line 10, however, with different calling contexts for the `echo` function (lines 3 and 4). A *superscript* indicates the line number(s) corresponding to the call stack at the time of invocation. For example, the two RPC invocations in Figure 5.1 can be distinguished by identifiers $A^{3,10}$ and $A^{4,10}$.

For the example in Figure 5.1, 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 5.2. In Figure 5.2, the reader is presented with a different implementation for A; it is assumed the exact implementation of B from Figure 5.1. In this example, A's RPC endpoint `helloworld` takes, as parameters, a list of `String`. For each `String` that is provided, an RPC is invoked to B's `echo` endpoint. If 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 is 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, an exhaustive analysis must run five different executions.

First, consider the execution where both loop iterations execute and all RPCs are successful, denoted 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, it is now demonstrated 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.

```
1  @service_a.method("helloworld")
2  def service_a_helloworld(ss : List[String]):
3      rs = []
4      failure = False
5
6      for s in ss:
7          try:
8              r = rpc(service_b, "echo", s)
9              rs.append(r)
10         except Exception as e:
11             failure = True
12             break
13
14     if failure:
15         s = "Hello World"
16         r = rpc(service_b, "echo", s)
17         return r
18     else:
19         return rs.join(" ")
```

Figure 5.2: 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 Count Alone is Insufficient.***
  Consider executions e_1 and e_4. Using *invocation count alone* 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 correctly 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 correctly assign identifiers to these RPCs, the granularity must be increased to include the number of times each RPC invocation statement is reached.

Combining the RPC signature and the calling context creates a dynamic *invocation signature*. This allows for handling both looping constructs and conditional control

flow, as presented in Figure 5.2. This is defined as follows:

Definition 9 (Synchronous Invocation Signature)**.** *The synchronous invocation signature for an RPC invocation is a triple* (s, t)*, usually denoted as* s^t*, where:*

- *s is the signature of the RPC;*
- *t represents 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 8) can be statically determined, the invocation signatures (Definition 9) are determined only based on observed executions.

5.2.3 Increasing Granularity: Path to Invoking RPC

Figure 5.3 is another variation of the `helloworld` microservice application. Similar to Figure 5.2, 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 5.2, 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, the parameter name `s` in the invocation signatures is omitted for readability.

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, including 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^{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^{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 2^{nd} 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.

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

    for s in ss:
        try:
            r = rpc(service_b, "echo", s)
            rs.append(r)
        except Exception as e:
            failure = True
            failures.append(len(rs) - 1, s)
            rs.append("")

    if failure:
        for (i, s) in failures:
            try:
                r = rpc(service_b, "echo", s)
                rs[i] = r
            except Exception as e:
                pass

    return rs.join(" ")

@service_b.method("echo")
def service_b_decorate_echo(s : String):
    try:
        r = rpc(service_c, "echo", s)
        return r
    except Exception as e:
        return s

@service_c.method("echo")
def service_c_echo(s : String):
    return s
```

Figure 5.3: RPC signature, when extended with invocation count and call stack, is insufficient when different incoming RPC requests trigger RPC invocation.

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 different, 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 an exhaustive search.

To resolve this issue, the path of RPC invocations that resulted in the current RPC must be included, as the call stack does not capture this information. 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.

Test executions e_1 and e_2 can be reformulated 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 10 (Synchronous Distributed Execution Index)**.** *The synchronous distributed execution index 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 synchronous distributed execution index 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 DEI is sufficient for applications that issue *synchronous* RPCs, but what about applications that use *asynchronous* RPCs to improve performance?

5.3 Asynchronous *Distributed Execution Indexing*

Recall from the previous discussion the definition of invocation signatures (Definition 9). With that definition and the associated discussion, the 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 5.4, a modified version of Figure 5.2, 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 their execution orders are 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 1^{st} iteration of the loop, where $A^7|_2$ is the RPC invoked in the 2^{nd} iteration of the loop. However, on repeated execution of this test through deterministic replay, or when performing an exhaustive search, scheduling nondeterminism may result in the 2^{nd} iteration of the loop being assigned $A^7|_1$, if the 2^{nd} block happens to execute first.

```
1  @service_a.method("helloworld")
2  def service_a_helloworld(ss : List[String]):
3      rs = []
4
5      for s in ss:
6          r = async {
7              return rpc(service_b, "echo", s)
8          }
9          rs.append(r)
10
11     awaitAll rs
12     return rs.join(" ")
```

Figure 5.4: Scheduling nondeterminism can permute the assignment of identifiers. In this case, $A^7|_1$, can refer to the RPC invocation from either the 1^{st} or 2^{nd} loop iteration.

Model checkers for distributed systems also face the problem of scheduling nondeterminism. However, these model checkers were primarily 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 unrealistic for large microservice applications where (A) they may not run all services on a single machine during testing and where (B) services are implemented in several languages. Therefore, ideally, a solution to this problem will not require control of the thread scheduler.

There are three ways this could be accomplished; unfortunately, none are sufficient.

1. *Cloning per block.*
   One approach is to *clone* the state used to generate identifiers for each asynchronous block. This would ensure that each block would independently count invocations for each RPC signature and associated call stack. However, this approach does not work. In Figure 5.4, 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 [Jos+09], 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 for 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 one follows this line of thinking, they could also *clone* the state used to generate the identifiers for each thread. This does not work either. In Figure 5.4, 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 *should* 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 state 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 11 (Invocation Payload)**.** *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 5.4, 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 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 the RPC signature, the calling context, and the invocation payload a *asynchronous invocation signature* for an RPC can be defined as follows:

Definition 12 (Asynchronous Invocation Signature)**.** *The asynchronous 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 *represents the call stack of the RPC.*

Thus, the notation $s(p)^t|_k$ refers to the k-th invocation of an RPC with asynchronous 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.

From here, we can define a *asynchronous distributed execution index*, or what we will refer to from here on as a *distributed execution index* (DEI).

Definition 13 (Distributed Execution Index)**.** *The distributed execution index 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 asynchronous 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 distributed execution index is thus recursive, with the base case being the top-level entry point to the application, whose path is the empty sequence [ ].

This design assumes that a key observation holds: microservice applications do not issue concurrent RPCs from the same call site, with the same calling context, to the same service, containing the same payload: what is believed to be valid from the survey of microservice applications performed during the construction of the microservice application corpus.

5.4 Implementation

DEI's can be represented in several formats in an implementation, depending on the required use case. In the case of the implementation for this dissertation, three different formats were realized with an option to reconfigure their use at runtime. However, other representations may be helpful when applying DEI's in other dynamic analyses.

5.4.1 Debugging Representation

For the **debugging representation**, each element in a DEI sequence is composed of five components:

1. A *metadata* component, where application developers can add debugging annotations into their code to populate the metadata section with additional debugging tags. This metadata section defaults to an empty value when no tags or annotations are used.

2. The *invoking service name*, a textual string of the invoking service's name. This is not necessary for correctness but may be helpful to the developer in identifying the caller of a particular RPC.

3. A *signature*, containing only the DEI's signature: module or class, method or function name, and parameter names and types.

4. A *synchronous* component, containing the representation of the call stack, as used in both the asynchronous (*i.e.*, Definition 12) and synchronous (*i.e.*, Definition 9) invocation signatures.

5. An *asynchronous* component, containing the representation of the invocation payload, as used in the asynchronous invocation signature (*i.e.*, Definition 12.)

6. The *invocation count* for the invocation signature.

The invoking service name, signature, synchronous, and asynchronous component form a five-tuple identifier. Identifiers and invocation count are combined into a tuple to represent an RPC executed by the application. Therefore, a sequence of tuples forms a DEI, encoded as an array in RPC invocation order, and captures the RPC invocation path.

5.4.1.1 Projection and Partial Orders

This representation helps debug and analyze microservice applications by providing several applicable *partial orders* using a DEI projection algorithm. For example, several projections have proven useful during this dissertation:

- ***Identifying all RPCs executed in an application trace.***
  By projecting the signature and synchronous components of the last sequence element in a DEI for each element in an application trace containing several DEIs, an application developer can identify all of the RPCs executed, *from a particular call site*, in that specific trace.

- ***Identifying possibly redundant RPCs from the same service.***
  By projecting the invoking service, signature, and asynchronous components in a DEI, developers can identify possibly redundant RPCs: when a service executes the same RPC, with the same arguments, from the same service twice as part of the same trace, regardless of the call site. For example, when a service retrieves the user information twice. This is only an over-approximation, as the RPC may perform a side-effect twice on purpose instead of reading values using an RPC.

- ***Identifying possibly redundant RPCs from different services.***
  By projecting the signature and asynchronous component of the last sequence in a DEI, developers can identify possibly redundant RPCs: when a service executes the same RPC, with the same arguments, from the same service twice as part of the same trace, regardless of the call site. For example, a service retrieves the user, while some dependent service retrieves the user. This may indicate that retrieved data should be passed between services instead of reinvoking the dependency. As before, this is only an over-approximation, as the RPC may perform a side-effect twice on purpose instead of reading values using an RPC.

Similarly, these projections have also been used to identify situations where RPC APIs could be constructed to be more efficient: for example, where two RPCs are issued to the same downstream service from the same service in direct sequence, where the data from the first RPC is fed into the arguments of the second RPC. This can indicate that the downstream service is not providing the correct API for its callers and may arise naturally from the decentralized nature of microservice application development.

These microservice application "smells" and their detection using DEI were used to construct a code linter that is available as open source as part of this dissertation.

5.4.2 Verbose and Compact Representations

The **verbose representation** is the same as the debugging representation. However, each element in the five-tuple is run through a cryptographically insecure hash function (*e.g.*, SHA-1) This representation is more space efficient than the debug representation but preserves the ability to identify signatures and call sites using hash comparison uniquely.

The **compact representation** further compacts the DEI representation, necessary for tracing systems that place a size restriction on metadata forwarded with RPC invocations, as each a DEI grows linearly with the depth of the microservice graph.

5.4.3 Assignment

DEI's are assigned by applying decorators (or, in the case of an RPC framework that supports interception of RPC invocations such as Google's gRPC, interceptors) to the RPC libraries in use. These decorators (or interceptors) interpose on invocations to each RPC and generate a DEI for the invocation before forwarding the call to the underlying RPC framework. Once generated, they must store the generated DEI in outgoing metadata (*e.g.*, HTTP headers, gRPC metadata) to be forwarded to the next service.

Request State. As DEI's are compositional in their sequence construction, the *incoming* DEI, the DEI assigned to the RPC invocation that resulted in a subsequent RPC execution, must be parsed from incoming metadata and stored in an interim location (*e.g.*, thread-local state) such that it can then be used in the generation of any subsequent *outgoing* DEI's (associated with downstream RPCs) originating from that service as a result of the incoming RPC.

This location must be accessible to any decorator attached to any RPC framework used on further downstream executions. For example, when an incoming gRPC invocation triggers downstream invocations of HTTP RPCs, the gRPC handler must place the incoming DEI into a location accessible by the HTTP library, performing the subsequent downstream RPC.

Service Re-Entrancy. As services may also be re-entrant, such that Service A calls Service B that then re-calls Service A using possibly a different RPC method, the state required for generating an outgoing DEI must also be discriminated by the entry. In short, a re-entry must generate outgoing DEI's using the corresponding incoming DEI.

To achieve this, incoming RPCs are assigned a random UUID called a *request identifier* associated with the incoming DEI. Therefore, when a service is re-entrant,

a new request identifier is generated for the re-entry, and the associated incoming DEI is related to that request identifier. This ensures that any outgoing DEI's are generated using the proper incoming DEI. As expected, instead of the incoming DEI, the request identifier must be placed in the same interim state.

Finally, services should expose a mechanism for resetting any accumulated DEI state. This reset mechanism is responsible for clearing any state, including the request identifier to DEI mapping and any other state accumulated to track DEI during execution.

Concurrency. While straightforward in theory, mechanizing this is significantly more complicated in practice. In Python, a programming language that contains concurrency with no true parallelism, the global state can track the active request identifier for the current thread and map request identifiers to DEI's. However, where concurrency and true parallelism are present in Java, DEI maintenance is more challenging.

For example, when receiving an incoming DEI, a request identifier must first be assigned and stored in a global mapping of request identifiers to DEI. This request identifier can then be assigned to thread local storage. Any subsequent RPCs executed by the invoked service can use the associated DEI to construct DEI for any downstream RPC invocations. But what happens if the invoked code spawns two threads to issue two downstream RPC concurrently?

One solution is to use an inherited thread-local state, such that any created threads inherit the request identifier and use the incoming DEI: this only works if access to the request identifier to DEI mapping is synchronized. However, if threads from a preexisting thread pool are used instead of creating new threads, the state will not be present in the local state of the used thread. Therefore, these threads must be bootstrapped with state, including the correct request identifier (in addition to synchronizing access to the request identifier to DEI mapping in the global state). This state must be migrated between threads upon context switches. This is further complicated by task parallel libraries that are built on top of threads in Java (*i.e.*, Kotlin, Scala, Clojure) or asynchronous RPC libraries with thread migration (*e.g.*, Armeria) where execution of RPCs can occur across multiple threads.

To solve this problem, it is necessary to implement the underlying concurrency mechanisms in Java (*e.g.*, `Thread`, `ForkJoinPool`, etc.) and any task parallel libraries written for the JVM (*e.g.*, Kotlin's `CoroutineContext`) to migrate any thread local state upon context switches to ensure that the request identifier is always present when RPC invocations occur. This is not a new problem; the authors of OpenTelemetry have already dedicated significant effort to solving this problem for their tracing

library. Therefore, their existing solution[4] is leveraged to ensure that DEI's are propagated accordingly in the presence of concurrency.

5.5 Takeaways

DEI's form one of the foundational contributions of this dissertation: the ability to identify the location of RPC invocations in a microservice application in a manner where they are consistent across executions of the same code. This ensures that any fault injection approach that aims to be exhaustive can guarantee that all RPC invocations are susceptible to fault injection and that no necessary fault injections are missed.

As presented, DEI's can also have applicability outside of just fault injection and can be used in other types of dynamic analyses: for example, by understanding application behavior, identifying where RPC APIs may be incorrectly designed, or where redundant RPC invocations are performed and should be optimized for performance.

[4]This solution relies on using a special Java instrumentation API (*e.g.,* `-javaagent INSTRUMENTATION.jar`) at runtime to be able to override the implementations of standard library classes (*e.g.,* `java.lang.Thread`).

Chapter 6

Service-level Fault Injection Testing

> *"A story should have a beginning, a middle, and an end, but not necessarily in that order."*
>
> Jean-Luc Godard

In this chapter, one of the core contributions of this dissertation is presented: *Service-level Fault Injection Testing* (SFIT). SFIT is a *software-supported methodology* for identifying a microservice application's behavior *under fault* through the use of automated fault injection testing.

In spirit, SFIT is not unlike the previously mentioned experimental approaches to fault injection (*i.e.,* chaos engineering) used by industry: SFIT injects faults at all of the inter-service communication points (*e.g.,* RPC) to see what the application's behavior is under that specific fault. However, SFIT improves on several of the deficiencies in the existing academic research in this area by taking a *developer-centric* approach:

- ***Exhaustive.***
  SFIT is exhaustive in that SFIT will automatically explore the impact of faults (and combinations thereof) of every RPC executed by a microservice application automatically, without requiring that developers specifically derive and manually execute fault injection scenarios.

- ***Specification-Free.***
  Instead of requiring that developers require specifications, in a specification language specific for fault injection testing, SFIT uses the functional test suite that developers of microservice applications are already writing to establish what the "correct" behavior of their application is when faults are not present.

- ***Behavior-Focused.***
  When failures occur due to fault injection, application developers will then be asked whether or not the failure is *legitimate*: did the application do the "correct" thing under failure? If not, SFIT has discovered a bug. If so, the developers are asked to update their tests accordingly with *derivations* from the expected behavior when a fault is injected: for example, by stating something should not occur when a fault is injected.

6.1 Overview

Service-level Fault Injection Testing (SFIT) is a developer-first approach for finding resilience bugs in microservice applications using fault injection. SFIT integrates fault injection testing as early as possible, *in development, before deployment of application code* and avoids the need for specifications, written in a specification language, by leveraging a developer's existing integration of functional tests of their application. This decision is critical, making it so SFIT can seamlessly integrate with developers' existing development environments and tools.

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

- ***Microservices developed in isolation.***
  Microservice architectures are typically adopted when teams need to facilitate rapid growth, breaking the team into smaller groups that develop individual services that adhere to a contract. This contract typically requires two or more teams to meet and agree to an API between the services they manage. Therefore, individual team members typically do not understand the state or internals of services outside their control well enough to write a detailed application specification to verify it automatically with a model checker.

- ***Mocking could prevent failures.***
  Many bugs in microservice applications could have been detected earlier if the developers had written mocks that simulated the failure of the remote service in a testing environment.

- ***Functional tests are the gold standard.***
  Instead of writing specifications, developers write multiple end-to-end functional tests that verify application behavior.

SFIT starts by identifying the faults that should be injected at each RPC invocation site in a microservice application to construct the fault space that will be exhaustively

searched. To do this, SFIT uses information from the underlying RPC frameworks to issue those RPCs: this accounts for both return values that indicate an error and thrown exceptions at the invocation site. SFIT also relies on instrumented versions of each RPC framework to identify RPC invocations and perform fault injection. This instrumentation communicates with a test server responsible for driving the exhaustive search. This is shown in Figure **??**.

To start, SFIT uses the existing functional test suite of the microservice application. This captures the desired behavior of the application when faults are not present.

However, SFIT places several requirements on these functional tests:

1. All downstream dependencies invoked by a service return successful responses when faults are not injected.

2. Each functional test contains a comprehensive set of assertions to detect application correctness bugs. For example, SFIT requires assertions that capture a non-customer-visible side effect that later prevents the customer from performing some subsequent action in the application. From SFIT's point of view, the onus of guaranteeing this is placed on the test's author.

With these requirements met, SFIT runs each test in sequence to perform the fault injection analysis. For each test, SFIT first runs a reference execution where no faults are injected. This is the same as running the test without SFIT. Then, SFIT will repeatedly re-execute it, injecting different (combinations of) faults until the fault space is explored.

Using the microservice application in Figure 6.1 to demonstrate, the RPC between the pricing adjustment service and the shipping service would first be tested for all possible faults. Then, the RPCs between the order service and its direct dependencies on the user, cart, and pricing adjustment services would be tested for those faults, including all possible combinations. As faults are injected, tests will fail. However, this is expected, as most existing tests only encode the application behavior when no faults exist. Therefore, developers will be prompted to use *fault injection predicates* provided by SFIT to assert the desired application behavior under failure.

SFIT sits at the opposite side of the design space of fault injection testing techniques for microservice applications like chaos engineering [Bas+16; RJ20a; Tuc+18; Bas+19b] by focusing on exhaustive correctness testing, **in development, before deployment of application code to production.**

In contrast, techniques like chaos engineering typically rely on coarse-grained[1] fault injection experiments designed by humans where high-level Key Performance Indicators (KPI) are used to detect resilience issues in the *production* environment:

[1] crashing or DNS black-holing an EC2 instance.

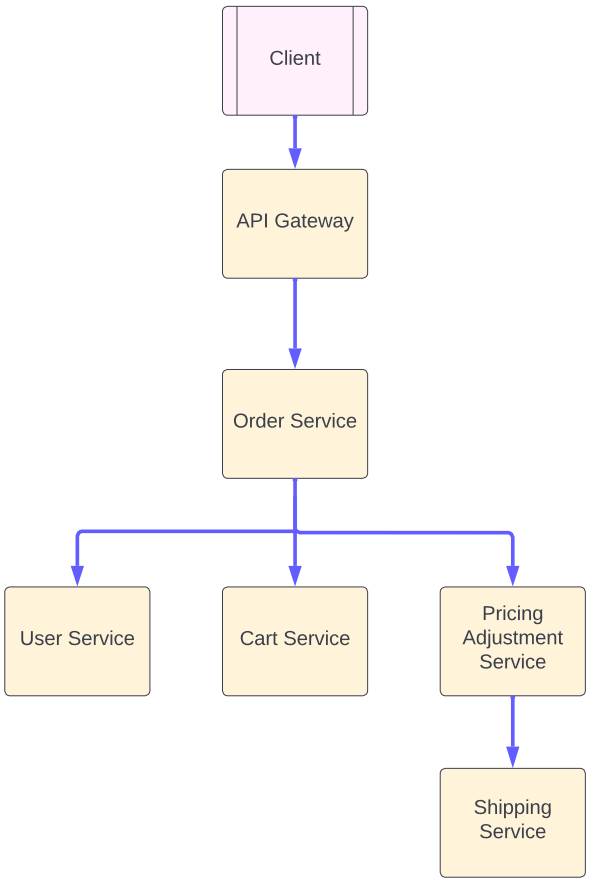

Figure 6.1: Purchase Application Structure

for example, Netflix's stream-starts-per-second KPI that indicates only if customers cannot watch video streams in their application. While chaos engineering is used by companies of all sizes, in all sectors — large tech firms (*e.g.*, Microsoft, Amazon, Google), big box retailers (*e.g.*, Walmart, Target), travel companies (*e.g.*, Expedia), financial institutions (*e.g.*, JPMC, HSBC), and media and telecommunications companies (*e.g.*, Conde Nast, media dpg, Netflix) [Mei+21b] — the cost of running experiments in production is not the same for all these companies. For example, a chaos experiment that prevents a user from reading an article in the New Yorker or streaming a movie with Netflix is not the same as preventing a user from purchasing a travel package with Expedia or performing a financial transaction with HSBC. This industrial partner Foodly, one of the largest food delivery services in

the United States, chose SFIT specifically for this reason: to verify the correctness of their application under fault during development, before deployment, without impacting customers.

Academically, SFIT builds on the Directed Automatic Random Testing [GKS05] technique, which combines symbolic execution with concrete execution to explore all possible paths of a monolithic application by synthesizing inputs that provoke the application into new application paths. In contrast, SFIT provokes an application into error-handling code paths by identifying the locations of method calls that may fail and injecting faults.

6.2 Algorithm

First, some definitions will be presented to start describing the *Service-level Fault Injection Testing* algorithm. This algorithm assumes a single functional test that results in the microservice application issuing a number of RPCs when executed.

First, the definition of a *fault configuration*, provided to SFIT to inform it on which faults should be explored, is presented. Then, definitions of how to test executions are represented internally to SFIT.

Definition 14 (Fault Configuration)**.** *A fault configuration is a partial function from a pair* (s, p) *to a set of fault specifications* $fs = \{f_1, \ldots, f_n\}$*, where:*

- s *is the signature of the RPC;*
- p *is the invocation payload of the RPC, which may be empty and*
- f *is a fault specification: a client-specific representation of the fault that should be injected into the application.*

In reality, f_i can be an arbitrary object indicating an exception to be injected with a set of properties set on the exception (*e.g.*, Google's gRPC) or may alternatively be the description of an HTTP error response containing the status code, body, and (optional) response headers. It is up to the instrumentation to correctly interpret and inject this fault, as the precise exception types that may be raised differ by implementation language of the invoking RPC client (*e.g.*, Java vs. Python gRPC.)

Definition 15 (Concrete Test Execution)**.** *A concrete test execution is a four-tuple* (t, d, f, r) *that represents a concrete execution where:*

- t *is the functional test that leads to the invocations of those RPCs;*

- *d is a set, such that $\{d_1, \ldots, d_n\}$ is the set of distributed execution indexes representing all of the RPCs executed by that functional test, t.*
- *f is a partial function from a distributed execution index d_i to a fault injection specifications f_i, for elements in d that indicates a fault was injected for the RPC represented by d_i;*
- *r is a partial function from a distributed execution index d_i to an RPC result for elements in d that indicates the client-encoded response of an RPC invocation under d.*

Concrete test executions are sufficient for a replay of a test execution, given a deterministic functional test. In short, as long as the application issues the same RPCs, identified by the same DEIs, the same faults will be injected on those RPCs, resulting in deterministic replay given a deterministic execution.

Definition 16 (Abstract Test Execution)**.** *An abstract test execution is a triple (t, d, f) that represents a partial concrete execution, where:*

- *t is the functional test that leads to the invocation of at least the RPCs in d;*
- *d is a set, such that $\{d_1, \ldots, d_i\}$ is the set of distributed execution indexes representing some of the RPCs executed by that functional test, t. This set is always a prefix of a valid, well-formed execution of t: for example, the first 5 RPCs executed by a test that executes 10 RPCs.*
- *f is a partial function from a distributed execution index d_i to an fault injection specification f_i, for elements in d that indicates a fault has or should be injected for the RPC represented by d_i.*

Abstract test executions are executions that *may or may not result in more RPCs*: for example, by injecting a fault, the application may respond by executing subsequent RPCs on failure. When this does not occur, the abstract test execution and its associated concrete test execution are *equivalent*.

Next, the search algorithm of SFIT is presented in Algorithm 1. This algorithm repeatedly re-executes the test of the microservice application, recording the concrete test executions associated with each test execution while deriving new abstract test executions that represent new failure scenarios until the failure space is exhausted based on the provided fault configuration.

First, an initial reference execution is executed. This is a *fault-free* execution that is equivalent to executing the test without injecting any faults. It executes as usual, unaffected by SFIT. During that execution, a concrete test execution contains all

of the DEI's associated with every executed RPC by the microservice application. During that execution, a set of abstract test executions are scheduled for execution. These executions are only valid prefixes of the RPCs executed by the application, as when injecting a fault on these executions may result in the subsequent execution of additional RPCs in failure handling code. These abstract test executions are generated for each RPC and each fault that should be injected for that RPC. By default, these are explored in *depth-first* search order: first analyzing a failure on the deepest failure and then exploring failures on ancestor RPCs. However, this is configurable to provide a more friendly developer experience where the first RPC executed by the application is the first where faults are injected (*i.e.*, breath-first.) Upon exploring each abstract test execution, a concrete test execution is generated for deterministic replay and reporting on the test executions explored by SFIT. We depict this algorithm formally in Algorithm 1.

Algorithm 1: SFIT Algorithm: Search

```
1  Executed by SFIT at start of the search
2  Function Search(t : Test, c: FaultConfiguration):
3      Executes the initial, fault-free reference execution
4      CurrentFaultConfiguration = c
5      CurrentConcreteTestExecution = ConcreteTestExecution()
6      CurrentAbstractTestExecution = AbstractTestExecution()
7      ExecTest(t)
8      ExploredAbstractTestExecutions.add(CurrentAbstractTestExecution)
9      ExploredConcreteTestExecutions.add(CurrentConcreteTestExecution)
10     Explore the fault space while increasing it as new failure dimensions exist
11     while te = ToExploreAbstractTestExecutions.pop():
12         CurrentAbstractTestExecution = te
13         CurrentConcreteTestExecution = ConcreteTestExecution()
14         ExecTest(t)
15         ExploredAbstractTestExecutions.add(CurrentAbstractTestExecution)
16         ExploredConcreteTestExecutions.add(CurrentConcreteTestExecution)
```

When executing a test with SFIT, each invocation of an RPC results in two messages being sent to SFIT. First, a `invocation` message when the RPC client issues the RPC; second, a `invocation_received` when the server receives the RPC. This is necessary for protocol-specific reasons for both Google's gRPC and HTTP. First, when issuing calls with gRPC, multiple services may implement the same gRPC

specification and respond to the same method; therefore, to properly inject faults on the appropriate service it is necessary to determine precisely the service that terminates the call. For example, even thought a gRPC service might indicate that the method `NotificationService.sendEmail` is being invoked, that method might be implemented by both a `OldNotificationService` and a `NewNotificationService` where one might only want to perform fault injection on a single service. Second, with HTTP, often requests are made using a hostname or IP only: therefore, to properly determine the service SFIT must defer fault scheduling until it knows the actual reported service name of the invoked host. In this case, SFIT defers fault injection until the actual service name is known.

In this presentation, it is assumed that the same `OnRPCInvoked` method is used in either case, as the logic remains the same and is tolerant to whenever clients report RPCs (*i.e.*, client invocation or server invocation) as long as a service name is present. However, it is important to note that when testing third-party services, the server invocation is not controlled and fault injection *must* be performed and configured based on client configuration alone. Whenever an RPC completes, it invokes `onRPCFinished`. This algorithm is provided in Algorithm 2.

Algorithm 2: SFIT Algorithm: Scheduling New Tests

```
1  Executed on SFIT any time the microservice application executes an RPC
2  Function OnRPCInvoked(d : DEI, s : Signature, p : InvocationPayload):
3      if d ∈ CurrentAbstractTestExecution.d:
4          Add to the current concrete execution
5          CurrentConcreteTestExecution.d.add(d)
6          This RPC appears in abstract execution
7          if f = CurrentAbstractTestExecution.f(d):
8              Add to the current concrete execution with fault
9              CurrentConcreteTestExecution.f.add(d, f)
10             return d, f
11         else:
12             Allow the RPC to execute as normal
13             return Noop()
14     else:
15         If we should inject some faults...
16         if fs = CurrentFaultConfiguration (s, p):
17             Schedule for testing in subsequent executions...
18             for f ∈ fs:
19                 NewATE = CurrentAbstractTestExecution
20                 NewATE.d.add(f)
21                 NewATE.f.add(f)
22                 ...only if we have not already scheduled for exploration
23                 if NewATE ∉ ToExploreAbstractTestExecutions:
24                     ToExploreAbstractTestExecutions.push(NewATE)
25         Allow to execute normally
26         return Noop()
27 Executed on SFIT any time the microservice application finishes an RPC
28 Function OnRPCFinished(d : DEI, r : Response):
29     CurrentConcreteTestExecution.r.add(d, r)
```

```
var response = rpc(service\_a, method, args);
if wasFaultInjectedOn(service_b, method, args):
    assertFalse(database.contains(record));
    assertTrue(response.isFailure()));
else:
    assertTrue(database.contains(record));
    assertTrue(response.isSuccess()));
```

Figure 6.2: SFIT's fault injection predicates are used to update a test to capture an application's behavior under fault. Yellow indicates added lines during SFIT process.

6.3 Fault Injection Predicates

As application developers apply SFIT to their functional tests, written to assume a happy path execution of their application, their tests will fail as faults are injected. Therefore, application developers will need to *adapt* these tests to account for the behavior of their application under fault. The SFIT mechanism is called *fault injection predicates*.

Fault injection predicates can be used in conditional statements placed in functional test code to make assertions on application behavior dependent on whether faults were injected.

For example, by wrapping assertions in statements such as the following:

- *"Was a fault injected during this test execution?"*
- *"Was a fault injected on a particular downstream service during this test execution?"*
- *"Was a fault injected on a particular RPC method during this test execution?"*
- *"Was a fault injected on a particular RPC method, with certain arguments, during this test execution?"*

In Figure 6.2, we provide an example usage of fault injection predicates to make a test capture both the application's behavior in and without the presence of faults. We indicate that under a fault injection to a downstream RPC, the response returned from the tested RPC should either succeed or fail, with writes performed to the database accordingly. In yellow, we highlight the lines that would be added during the SFIT process. This process is not unlike the methods used by software model checking [JM09] or interactive theorem proving [BC13], where developers must encode all possible behaviors for completion.

Fault injection predicates use the current concrete execution to determine if that RPC was subjected to a fault and returns a response to the test. SFIT can be configured to generate all fault scenarios regardless of whether a previous fault injection scenario passed or stopped at the first failure. As always, upon each test execution with SFIT, all fault injection scenarios are re-executed and checked against the test's assertions.

6.4 Testing Process

Outlined below is the SFIT testing process:

1. Execute the developer's functional test without fault injection.
   Once the test is passing, proceed to fault injection.

2. Execute fault injection scenario from SFIT.
   If the test fails, display failure to the developer with a log of failure and fault injection details.

   a) *If a bug, resolve.*

   b) *If not bug, encode application's behavior under fault using recommendations.*

   c) **Go to 1.**

3. If more fault injection scenarios remain, go to 2 **only if the current test execution passed based on the developer specifications of application behavior under fault.**

6.5 *Encapsulated Service Reduction*

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 service requests requires executing tests in a magnitude that is *exponential in the number of service requests*. However, it is possible to leverage the decomposition of an application into independent microservices to reduce the search space without loss of completeness dramatically.

6.5.1 Service Encapsulation

Let us revisit the Audible example that is presented in Figure 3.1. Excluding the complete failure space for readability, let us 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 three dependencies: the Ownership, the Activation, and the Stats services. The entire request is failed if either the Ownership or Activation service calls fail. However, if the call to the Stats service fails, that failure has no impact on the result of the request. After testing, we 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.

With the CDS, at a minimum, SFIT has 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, to fully explore the failure space using our approach, 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 of and should not have to be tested in combination. For example:

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

It is critical, then, to identify a way to leverage our knowledge of service failures and their impact on the services that take them as dependencies. To do this, SFIT can take advantage of the following three key observations:

- First, SFIT must fully explore how a service's dependencies can fail. This ensures that SFIT understands the behavior of a single service when one or more of its dependencies fail and what the resulting failures returned by that service are. Referring to the Audible example in Figure 3.1, it is clear that 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.)
- Second, when SFIT is about to inject faults on at least *one* dependency of *two or more* different services, SFIT already know the impact that those failures

will have on the services which takes them as dependencies. Referring to the Audible example in Figure 3.1, SFIT already knows what the ADS will return when its dependencies fail in any possible combination, as we've already run that test. SFIT also already knows what the CDS will return when its dependencies fail in any possible combination for the same reason. Therefore, SFIT does not have to inject the fault at the dependencies. SFIT can inject the appropriate response directly at the ADS or CDS.

- Third, SFIT has already injected that fault at that service, then the test is redundant, as it has already observed that application behavior.
  Referring to the Audible example in Figure 3.1, SFIT do not need to test the Stats service failing in combination with failures of the Audio Assets or Audio Metadata services, as SFIT already knows the outcome of those failures on the services that take them as dependencies; we have also already observed those outcomes.

Algorithm 3 presents the encapsulated service reduction algorithm (ESR). 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 3.1, 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.

The property of microservice applications that enables this style of dynamic test case reduction is referred to as *service encapsulation* and is enabled by DEI's indexing algorithm.

Definition 17 (Service Encapsulation)**.** *Service encapsulation states that invoked downstream dependencies of a service can only indicate failure or success to their direct caller, and as long as they do not expose their internal state to the invoker, the invoker can only assert on the responses that are returned to the caller by those dependencies.*

Therefore, an invoking Service A and a Service B exhibiting this property is said to be encapsulated by this service: in short, Service A is said to *encapsulate* Service B; conversely, Service B is encapsulated by Service A. When this is true, any service invoking an RPC to Service A can only infer the behavior of Service B through Service A's behavior.

Encapsulated service reduction is automatic and requires no additional information from the application developer. It is important to note that encapsulated service reduction is not sound in general and refers to our assumptions above on the behavior of a single functional test: service responses are deterministic for a single

functional test, and service code does not contain data dependencies on previous failures.

It is important to note that *Encapsulated Service Reduction* is not sound in general and refers to our assumptions above on the behavior of a single functional test: service responses are deterministic for a single functional test, and service code does not contain data dependencies on previous failures.

6.5.2 Algorithm

Algorithm 3: SFIT Algorithm: Encapsulated Service Reduction

```
1  Function ShouldSkip(a: AbstractTestExecution):
2      Start by searching every concrete execution we have already run...
3      for ConcreteTestExecution ∈ ExploredConcreteTestExecutions:
4          found = True
5          For every RPC executed in the abstract test execution provided...
6          for d ∈ a.d:
7              If we want to inject a fault in our execution...
8              if f = a.f(d):
9                  ...but the fault we want to inject already happened organically...
10                 if f ∈ ConcreteTestExecution.r:
11                     ...then, we might be able to skip this execution
12                     found = found && True
13                 else:
14                     ...but if the injected fault doesn't match our response...
15                     ...we have to run it.
16                     found = found && False
17         If every fault we want to inject occurred organically (or synthetically)
18         in another test execution we already ran, we can definitely skip it.
19         if found:
20             return True
21     Otherwise, we have to run it because it's different...
22     return False
```

This algorithm is executed each time an abstract test execution is about to be executed by the SFIT search. First, every concrete test execution that has already

been executed is searched to determine if there is a matching concrete execution where, for each fault, the following criteria are met: either a fault was injected on an encapsulat*ed* service that resulted in the encapsulat*ing* service returning a failure response that we intend to inject through fault injection, or a fault is (or should be) directly injected on that request. If that criterion is true, ESR determines that the test case is redundant and does not execute it.

This results in the following *augmented* search algorithm for SFIT, presented in Algorithm 4: SFIT with ESR.

Algorithm 4: SFIT Algorithm: Search with ESR

```
1  Executed by SFIT at start of the search
2  Function SearchESR(t : Test, c: FaultConfiguration):
3      Executes the initial, fault-free reference execution
4      CurrentFaultConfiguration = c
5      CurrentConcreteTestExecution = ConcreteTestExecution()
6      CurrentAbstractTestExecution = AbstractTestExecution()
7      ExecTest(t)
8      ExploredAbstractTestExecutions.add(CurrentAbstractTestExecution)
9      ExploredConcreteTestExecutions.add(CurrentConcreteTestExecution)
10     Explore the fault space while increasing it as new failure dimensions exist
11     while te = ToExploreAbstractTestExecutions.pop():
12         CurrentAbstractTestExecution = te
13         CurrentConcreteTestExecution = ConcreteTestExecution()
14         if ! ShouldSkip(CurrentAbstractTestExecution):
15             ExecTest(t)
16         ExploredAbstractTestExecutions.add(CurrentAbstractTestExecution)
17         ExploredConcreteTestExecutions.add(CurrentConcreteTestExecution)
```

6.6 Takeaways

Service-level Fault Injection Testing (SFIT) is one of the core contributions of this dissertation: a fault injection technique for identifying resilience issues in microservice applications.

SFIT achieves this by leveraging DEI, a technique for uniquely identifying the RPCs executed by a microservice application, to systematically inject a list of faults

on each of (and all combinations thereof) those RPCs. This forces application developers to think about failure scenarios they may or may not have been aware of and ensure their application does what they expect when those failures inevitably occur, *in development* and before application code is deployed to production.

To account for these failures, application developers are prompted to use *fault injection predicates*: a mechanism for encoding the failure behavior of their microservice application in the existing happy path tests of their application.

When possible, SFIT can avoid executing *redundant* test cases where faults are injected, and the outcome of those faults is already known because of service encapsulation. Service encapsulation is a core property of microservice applications and the way they are developed: failures of downstream services are often hidden behind, or within, the responses of their ancestors in the microservice graph. This optimization can reduce the number of test cases needed for full coverage of the faults exhibited in application code upon dependency failure of a downstream service.

In the next chapter, an evaluation of SFIT is presented, using the other foundational contribution of this thesis: the constructed microservice application corpus taken from industry examples.

Chapter 7

Evaluating SFIT: Corpus

"If you can talk brilliantly about a problem, it can create the consoling illusion that it has been mastered."

Stanley Kubrick

In this chapter, a synthetic evaluation of both *Distributed Execution Indexing* (DEI) *Service-level Fault Injection Testing* (SFIT) is presented.

First, the experimental configuration is presented. This section describes the prototype implementation of SFIT, called Filibuster. This initial implementation is written in the Python programming language to match the implementation language of the synthetic microservice corpus and is herein referred to as Python Filibuster. This is done to distinguish it from the second prototype implemented in Java, which will be discussed in a subsequent chapter.

Next, an evaluation of DEI is presented using the microservice corpus. This evaluation shows that DEI encodes the minimal amount of information required to correctly identify RPCs in a microservice application both uniquely and deterministically. Specifically, this is demonstrated by the systematic removal of each component in the DEI highlighting the violations in either soundness or completeness (in SFIT) that result from their removal.

Next, SFIT is evaluated on the corpus using the DEI's evaluated in the previous section. First, it is shown that SFIT can increase code coverage by automatically generating the necessary tests to exercise the code paths associated with RPC failure. This shows that in addition to the increased coverage, the SFIT can identify all of the synthetic, industry-inspired bugs seeded into the corpus applications. Then ESR is evaluated, and it is shown that ESR can significantly reduce the testing overhead by removing redundant test cases when microservice application graphs are structured accordingly: depth over breadth. From there, it is shown that SFIT

can avoid expensive mock creation that would typically be manually created by automatically generating those test cases. Finally, it is shown that the execution of SFIT is reasonable enough to be integrated directly into either the local development or continuous integration environments to identify application resilience issues *in development, and before deployment* of application code to production.

7.1 Experimental Configuration

All experiments in this section were performed using the prototype implementation of Filibuster in Python. This implementation is called Python Filibuster and is available as open source.

As all the services in the constructed corpus communicate using HTTP, the popular HTTP *requests* library was used. To enable fault injection, the opentelemetry instrumentation library for Python was extended to support fault injection and enable metadata assignment to requests. To identify the target of requests made to services via URL, the opentelemetry instrumentation library was used for Flask to record the service where instrumented requests were terminated.

This instrumentation assigns distributed execution indexes and vector clocks [Fid91; Mat88] to each request to uniquely identify each request. This information is also forwarded to each service through the instrumentation. Distributed execution indexes correlate calls between different test executions for dynamic reduction; vector clocks are used to identify the dependencies of a particular service.

This instrumentation communicates with the Filibuster server, as shown in Figure 7.1. The Filibuster is responsible for starting all the services associated with an application in either local processes, Docker Compose, or Kubernetes. The Filibuster server then runs the functional test, records and maintains the stack of test executions to execute, performs functional test assertions, reports test failures, and aggregates test coverage. The Filibuster client library, written in Python which communicates with the Filibuster server provides an API that functional tests can use to write conditional assertions and allows for test replay using a counterexample file. Test coverage is aggregated from each service by the server.

Filibuster requires a static analysis to determine the types of failures each service can return. In this prototype implementation, a purely lexical static analysis that over-approximates these errors by performing abstract syntax tree traversals on the source code of each service is used. This analysis is highly tailored to how Flask applications are written by identifying `raise` statements that throw exceptions converted to HTTP responses containing status codes indicating error by Flask. (*e.g.*, `ServiceUnavailable`, `NotFound`.) If this type of analysis is impossible, developer's

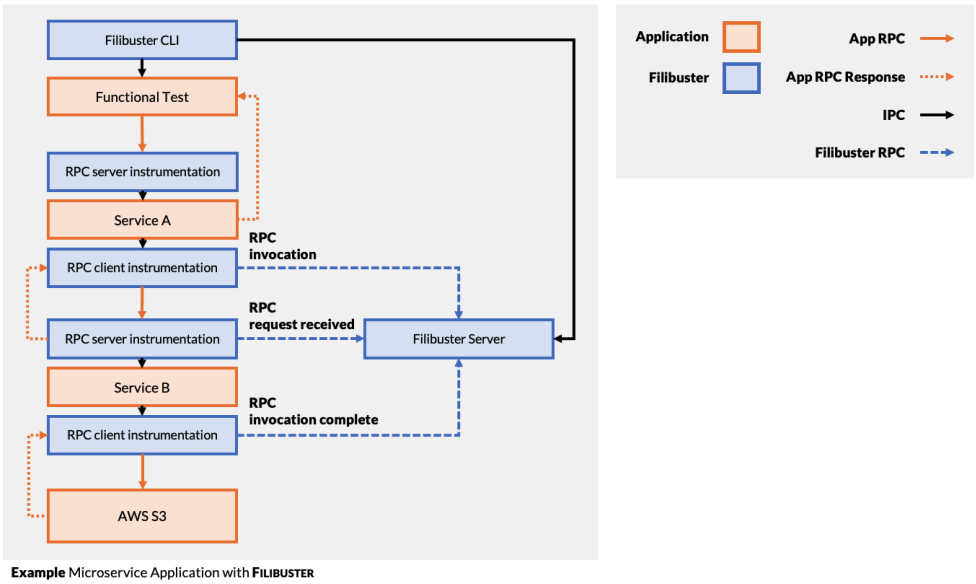

Figure 7.1: Architecture of Python FILIBUSTER

can opt to test all potential failures, as a finite number of HTTP status codes indicate the error.

This implementation of FILIBUSTER can inject the following faults (or failures):

- *Callsite exceptions.*
  Callsite exception thrown by the *requests* library that indicates conditions like connection error or timeout. For all exception types, FILIBUSTER can conditionally contact the other service before throwing the exception. For timeouts, FILIBUSTER can conditionally wait the timeout period before throwing a timeout exception.

- *Error responses.*
  Error responses returned from a remote service using standard HTTP error codes that indicate conditions like Internal Server Error or Service Unavailable. For each error code, FILIBUSTER can return an associated body conditionally.

Each is configurable when running FILIBUSTER. This enables developers to run a subset of these errors during local development, a more significant subset on push to a feature branch, and the complete subset as a nightly job. The prototype implementation of FILIBUSTER was configured to run in AWS CodeBuild following this design, allowing for rapid feedback to developers and complete coverage nightly.

As Filibuster has been written using a client-server architecture, cross-language support is possible and currently exists for the JUnit testing library in Java for HTTP and Google's gRPC: this implementation is discussed in a subsequent chapter. All communication between instrumentation and the Filibuster server is through a language-independent protocol; anything language-specific is done in the instrumentation library.

7.2 *Distributed Execution Indexing*

In this section, Python Filibuster is used as an evaluation of DEI to demonstrate that the innovations in DEI are required for a microservice application fault injection approach to be both sound and complete. In short, without the inclusion of the invocation count, call stack, and invocation path, a fault injection analysis such as SFIT can violate either soundness (where faults are injected on the incorrect RPCs), completeness (where faults are not injected on RPCs they should be), or both.

Next, an example application in Java is written to demonstrate that nondeterminism is a problem that any mechanism that identifies RPCs for the target of fault injection must deal with. This application is tested using Filibuster by extending the Filibuster prototype with support for Google's gRPC in Java. Then, it is shown that through the inclusion of the invocation payload, DEI can deal with nondeterminism while avoiding the expensive instrumentation required for control of the Java scheduler; however, only when applications do not issue concurrent RPCs to the same RPC service, method, from the same call site, with the same RPC arguments.

While these innovations are evaluated in the context of the prototype implementation of SFIT, Filibuster, any request-level fault injection technique (*e.g.*, LDFI, 3MileBeach, etc.) are susceptible to these issues. Therefore, DEI is a general solution for any request-level fault injection technique for addressing the identification of RPCs for fault injection in a microservice application.

7.2.1 Required: Invocation Count, Stack, and Path

The constructed corpus is used to demonstrate the need for invocation count, call stacks, and RPC path. As none of the industrial examples in this corpus contained the coding patterns discussed in Section 5, the cinema examples from the corpus are used for this evaluation instead.

With regard to the cinema examples, one cinema example in particular, *cinema-3*, exhibits a pattern that requires the inclusion of the call stack or invocation count. To demonstrate the need for both, combining the structure of *cinema-6* with retries on

Cinema Example	All	No IC	No Stack	No IC & Stack	No Path, IC & Stack
cinema-3	7	**4**	7	-	-
cinema-9	5	5	5	**3**	-
cinema-10	6	6	6	6	**5**

Table 7.1: Results demonstrating all techniques must be combined for correct identification.

failure from *cinema-1 is necessary.* It is also necessary to extract the RPC invocation into a helper function.

To demonstrate the need for inclusion of the execution path, it was required to combine the structure of *cinema-2* with the use of default responses on failure from *cinema-5*. These *new* examples are included in the corpus and referred to as *cinema-9* and *cinema-10*, respectively. These are not unrealistic, as the changes were taken from other examples in the corpus and combined with existing examples.

To demonstrate, recall that cinema is comprised of four services, as depicted in Figure 4.1. The useres service retrieves the bookings for a user: this involves an RPC to the Bookings service and then to Movies serivce for each retrieved booking. In the two variations, either:

1. the Users service issues an RPC to the movies service after the response from the Bookings service, or

2. the Bookings service issues an RPC to the Movies service directly.

For testing, a functional test that retrieves a user's movie bookings is used. In terms of fault injection specifically, a single connection error exception is injected for each RPC.

The evaluation results are presented in Table 7.1. For clarity, redundant results are omitted: for example, if a test execution produces either unsound or incomplete behavior without including an invocation count. No results are presented where invocation count and stack are excluded.

Invocation Count. For this experiment, *cinema-3* is used.

In *cinema-3*, the RPC from the Users service to the Bookings service is performed in a loop and re-executed once on failure.

Exhaustive search requires seven executions. Without the invocation count, only four executions are performed.

✓ **SFIT is incorrect without *invocation counting:***

- *Unsound.*
  Each RPC in the loop will be assigned the same identifier, SFIT will inject a fault on zero or all iterations.
- *Incomplete.*
  Requests that occur due to any iteration, not the 1st, will not have faults injected.

Call Stack. For this experiment, *cinema-9* is used.

In the event of failure of the 1st RPC from the Users service to the Bookings service, it will mark the request as failed and try that request later from a different call site. This differs from the loop where the same call site is used. Each call site uses a helper function to issue the RPC to ensure the stacks differ.

Exhaustive search requires five executions. Without call stack, only three executions are ran.

✓ **SFIT is incorrect without *call stack inclusion:***

- *Unsound.*
  As each call invocation of the helper's RPC will be assigned the same identifier, SFIT will either inject a fault on both or neither.
- *Incomplete.*
  Requests that occur as a result of the 1st invocation will not have faults injected.

RPC Path. For this experiment, *cinema-10* is used.

In the event of a failure of the Bookings service, a default response is used in place of the failure and the Movies service is contacted directly by the Users service.

Exhaustive search requires 6 test executions. Without the RPC path, only 5 executions are performed.

✓ **SFIT is incorrect without *RPC path inclusion:***

- *Unsound.*
  Since the bookings RPC to movies and the users RPC to movies share the same identifier, SFIT will either inject a fault on both or neither.
- *Incomplete.*
  As SFIT will always fail the 2nd RPC to movies, the successful case is not explored.

7.2.2 Nondeterminism is a Problem

In order to understand the impact of scheduling nondeterminism within the JVM on correct identifier assignment, a small example was constructed with the Armeria library for Java that contained two services: Hello and World.

In this example, the World service exposed a single endpoint that returned a `String` constant when it received an RPC. The Hello service exposed an endpoint that, when it received an RPC from the test harness, would launch a configurable number of threads, each that issued an RPC to the World service, and then wait for them to complete. Each thread was defined as a class in Java, where the Hello service would create instances of this class of in a loop. This ensured that the call site of the RPC was the same and the stack trace of the call site were identical. All RPCs were made the same service and differed only in the payload, which contained the identifier of the thread determined by thread creation order.

For this experiment, Python Filibuster was extended with support for Java, Google's gRPC library. Then, a server stub at the World service was installed to record the execution indexes generated by the Hello service. Then, the DEI algorithm was reconfigured to include the thread creation order; therefore, the payload differed only by this identifier.

This test application was executed for varying numbers of RPCs (2, 4, 8, 16, 32, 64) for 100 iterations each. The thread pool size was also fixed at the Hello service at size 2. For each iteration, it was recorded whether or not the execution index assignment matched the thread creation ed by examining the execution index payload values. The results are presented in Figure 7.2.

With only 2 RPCs, 44% of the tests exhibited an RPC execution order that did not match the creation order; by 64, 100% of the RPCs did not match the creation order. With 16 RPCs, this rose to 80%; by 32 RPCs, this was at 96%, and by 64, 98% of the RPCs did not match the creation order.

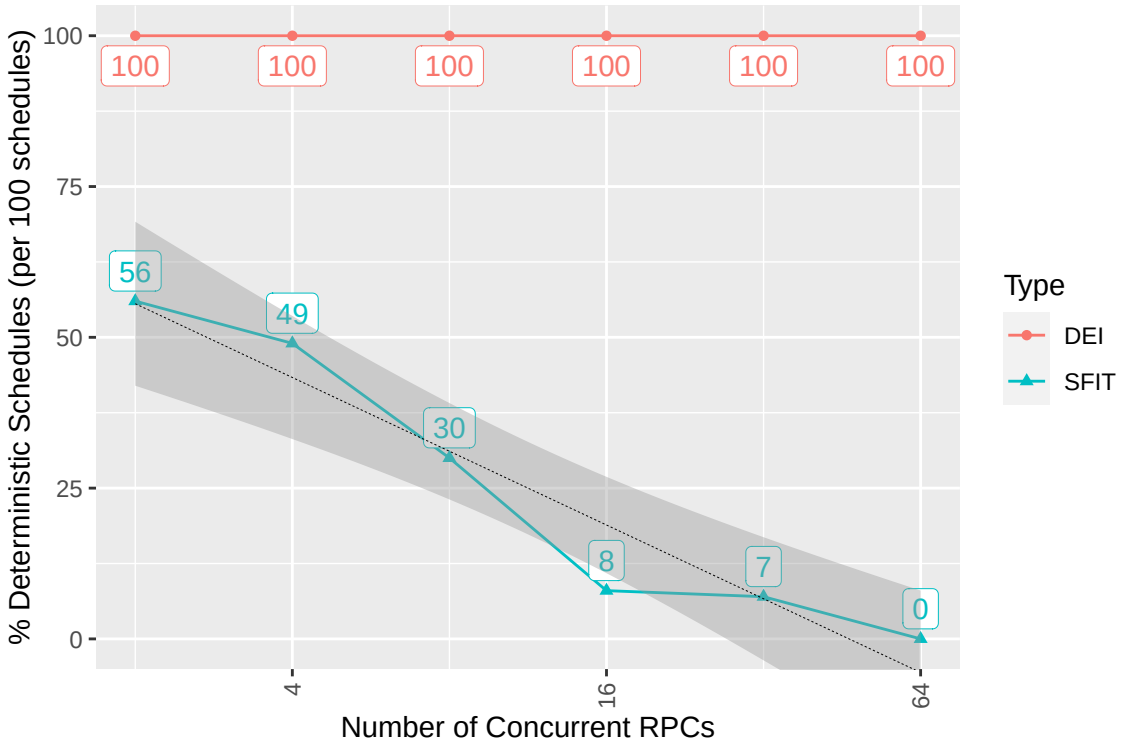

Figure 7.2: Percentage of executions with deterministic assignment for two threads.

✓ **Even in the presence of relatively low amounts of concurrency, scheduling nondeterminism is a problem for SFIT.**

7.2.3 Payload Inclusion Distinguishes

Using the same example, the *key insight* of asynchronous DEI could be verified: including the payload into the identifier for each RPC invocation was sufficient for distinguishing these concurrent, inter-service RPCs. With asynchronous DEI, all RPC identifiers were unique and deterministic, as shown in Figure 7.2. Therefore, scheduling nondeterminism was not an issue.

✓ **Payload is sufficient for distinguishing concurrent, inter-service RPCs in microservice applications, assuming that these RPCs will not share the same payload when issued to the same service and method with the same parameters.**

7.3 *Service-level Fault Injection Testing*

Table 7.2 presents results from running Python Filibuster on the corpus; the table encapsulated service reduction is shortened to ESR. It is assumed that all remote calls can return a connection error for faults. When a timeout is specified, timeout exceptions are considered. Any service-specific failures are also included, as determined by the Python Filibuster static analysis.

All examples were run on an AWS CodeBuild instance with 15 GB of memory and eight vCPUs. At the start of the Python Filibuster run, all of the services were started for each example; Python Filibuster waited for those services to come online,e, 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 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 system's performance with and without encapsulated service reduction.

7.3.1 Tests Generated and Increased Coverage

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

The "Test Gen/ESR Gen" column presents the number of tests generated and executed by Python Filibuster. Since each example only has a functional test, these numbers include that test in the total. Python 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 could be identified using Python Filibuster.

The "Coverage After" column shows the increase in statement coverage. By generating the tests that cover possible failures, Python Python Filibuster can increase the 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.

✓ **SFIT 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 many tests to ensure coverage of the failure space properly. For

Example	**Test/ ESR Gen**	**Coverage After** (%)	**Time w/ESR** (s)	**ESR 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)	18653/46 70 (-13983)	97.38 (+15.67)	2363.84 (+2353.8)	744052 (0.07)	60002.91 (3.31)

a

Table 7.2: Python FILIBUSTER evaluated on the corpus. Includes the number of generated tests with and without encapsulated service reduction; coverage before and after using Python FILIBUSTER, overhead of encapsulated service reduction algorithm, and overhead of test generation.

most organizations, manually writing this many tests without a system to generate these tests automatically would be expensive in terms of development time. Similarly, the cost of test adaptation is also low. In the Netflix example, Python Filibuster executed 1,606 tests but required only nine fault injection predicates to capture all behavior. SFIT also found all of the bugs in a *development setting* without running chaos experiments in a live production environment.

As discussed in our section on corpus creation, all of these bugs were discovered using chaos engineering. They were used as use cases to advocate for the adoption of chaos engineering. Using Python Filibuster, chaos engineering can be avoided.

7.3.2 Encapsulated Service Reduction

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

Encapsulated service reduction excels when graphs have more depth and less breadth. In the Audible example, there are deep paths containing nested requests that can allow Python Filibuster to avoid running redundant test executions. However, in the Netflix example (without bugs), the graph has an enormous breadth with no depth. In this case, all combinations of failures must be tested, as the 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 encapsulated service reduction become valuable–only 25% of the tests have to be executed to reach the same failure coverage.

✓ **When applications are structured with depth over breadth to the service graph, they can significantly benefit from encapsulated service 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 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 the reuse of results across test executions. This reduces the overall test time required to exhaust the space of possible failures.

7.3.3 Mocks

During the initial corpus implementation, unit 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 3.1 and consider the Audible Download Service. In this example, unit tests were only written, each containing a mock for the failures of the three dependencies: Ownership, Activation, and Stats. The list of service-specific failures is omitted 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, but it was also a significant amount of additional code. These failures also under-approximate the actual failures that could occur in the application: mocks that verified all possible combinations of failures were not written. For example, the failure of 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. An additional 743 LOC (+79.3%) of test code was written to verify failure behavior.

> ✓ **SFIT 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.**

Python 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.

7.3.4 Execution Time

The "Time w/ESR" column shows the execution time with encapsulated service reduction enabled. This column shows the total execution time for all tests, excluding setup and tear-down time. The difference between running the initial single functional test and running all of the tests generated by Python Filibuster is presented in parentheses.

Comparing this difference to the number of tests generated and executed with encapsulated service reduction, it is clear. It is expected that the execution time scales linearly with the number of tests that must be performed. This per-test execution

time accounts for starting a Python interpreter, performing whatever setup and tear-down 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 Python FILIBUSTER server, schedules new test executions each time a new request is reached, and the Python FILIBUSTER server learns about this call through the instrumentation call from the service. As is apparent, this overhead is minimal. The overhead for each generated test is presented in parentheses, which in the worst case is 3.2 milliseconds.

In the "ESR Overhead" column, the total overhead (in milliseconds) introduced by the encapsulated service 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 apparent, this overhead is minimal. The overhead per test is presented in parentheses: in the most complicated examples, it is 90 microseconds.

✓ **SFIT's execution time scales linearly with the number of generated tests.**

However, the test generation overhead is significantly less than the cost of the development time required in manually writing these tests using mocks. Additionally, Python FILIBUSTER provides higher coverage by automatically writing mocks for combinations of failures across service boundaries.

7.3.5 Misconfigured Timeouts

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 one additional millisecond before returning a `Timeout` exception. This ensures that Python FILIBUSTER waits at least long enough to account for the timeout interval.

In Figure 7.2, the difference in execution time when testing timeouts is highlighted in red. To identify Netflix bug #1, Python 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.

✓ **SFIT can detect incorrectly configured timeouts at the cost of additional execution time, equivalent to the injected timeout duration.**

7.4 Takeaways

In this chapter, an evaluation of *Service-level Fault Injection Testing* (SFIT) was presented using the initial prototype implementation of SFIT in Python, FILIBUSTER.

Using the synthetic microservice application corpus constructed in Chapter 4, it was first demonstrated that *Distributed Execution Indexing* (DEI) is a necessary foundational technique for the identification of RPCs executed, for fault injection, in a microservice application.

DEI's provide both a sound and complete method for enabling request-level (*e.g.*, RPCs) fault injection techniques while ensuring that repeated execution of the same application path yields the same identifiers enabling both exhaustive search and deterministic replay. DEI also address RPC execution nondeterminism in microservice applications, under certain design restrictions, while avoiding the necessity of scheduler control in managed VMs: common in distributed system testing but impractical in microservice applications composed of many different services that are implemented in other implementation languages.

SFIT was then evaluated against the synthetic microservice application corpus. It was shown that SFIT can identify all application bugs seeded into the synthetic microservice corpus at lower cost, and *in development* compared to existing fault injection techniques typically performed in production. By leveraging an emergent property of microservice applications and the way they are developed, *service encapsulation*, DEI's can be used in SFIT to further reduce the overhead in testing microservice applications for resilience by avoiding redundant test cases when testing the application wholesale, end-to-end. This test case reduction technique is referred to as *Encapsulated Service Reduction*.

In short, SFIT provides an efficient mechanism for detecting resilience bugs, *in the development environment* and before deploying application code, when the application can be tested end-to-end in the development environment.

Chapter 8

Industrial Microservices: Foodly

"I learned a long time ago that reality was much weirder than anyone's imagination."

Hunter S. Thompson

As discussed in the background and related work, designing a fault injection technique for microservice applications is only half of the problem: many fault injection techniques have been created in academia but have failed to find adoption with industry practitioners as their designed was mismatched with how application developers design, develop, deploy, and maintain their microservice applications.

To address this deficiency, it was first necessary to identify an industry partner interested in using fault injection testing to improve the resilience of their microservice application. Then, it was necessary to determine if the designed approach was compatible with how they develop and test their services. Finally, it was necessary to determine if the approach could identify application bugs related to resilience in their application. Therefore, and towards that end, a partnership was established with Foodly. Foodly operates one of the largest food delivery service in the United States using a microservice application comprised of over 500 different services.

This chapter presents background material on Foodly: how they build their microservice application, how they remain resilient to faults, and why, from their perspective, existing industrial fault injection techniques are not sufficient or appropriate for their consumer product.

8.1 Foodly

Foodly is a food ordering and delivery consumer product built using a microservice architecture comprised of over 500 services, primarily written in the Kotlin programming language. Foodly's decision to migrate from a monolithic to a microservice architecture to improve developer productivity at scale. Each service in Foodly's microservice application platform provides a specific set of functionality that is grouped by business logic (*e.g.*, ordering, delivery), and an independent team manages the software life cycle for those services. Services at Foodly primarily communicate using Google's gRPC, with some legacy communication occurring over HTTP. Services are deployed when new code is ready for release, with around 100 unique deployment events occurring daily.

Each service's functionality is controlled at runtime using an experimentation framework that allows features to be deployed and then gradually enabled for customers. This is done to minimize the impact of application bugs and allow for experiments that adversely impact the business to be disabled, all without redeploying. These experiments are tested independently but run concurrently in production. There can be hundreds of active experiments running across the deployed application simultaneously. The combination of frequent deployments and active experiments means that the system is constantly in flux and, as a result, no specification of the end-to-end system behavior exists.

8.2 How Foodly is Resilient To Faults

Foodly's microservice application must remain resilient to faults. For example, in the case of bad deployments, where application code that contain bugs or is prone to crashing and spurious failures due to underlying infrastructure issues such as container failures and network anomalies. At Foodly, and many other companies that operate microservice applications, this reliability is ensured through several different techniques, working in concert and with the primary goal of minimizing the impact of failures in production.

Several techniques are employed to mitigate the impact of bad deployments: container orchestration, canary deployments, load balancing, probes, and autoscaling rules. Container orchestration is used to support rolling deployments of services where possible. Canary deployments and bulkheads detect bad releases as early in the deployment process as possible while minimizing the impact of these bad deployments. Services are deployed into replica sets, and load balancing distributes the load across all service instances within a replica set. Services within a replica set are automatically checked using startup, liveness, and readiness probes

to detect malfunctioning services for automatic removal from the replica set. Autoscaling rules and restart policies ensure that a minimum set of nodes in a replica set is operational at a given time. These techniques ensure that services are online and responding as much as possible by minimizing the impact of bad deployments through detection and automatic rollback.

Retries, load shedding, and circuit breaking are employed for spurious failures. Foodly's custom RPC client automatically retries RPCs in the event of failure, except timeout failures, which may indicate a failure or slowdown of a transitive dependency. Load shedding is employed by each service to ensure that services refuse (*i.e.*, short-circuit with an error response) incoming RPCs when the service is either already overloaded or at risk of becoming overloaded should they process the incoming RPC. Circuit breakers are used at each service to prevent repeated invocation of a downstream dependency that may be malfunctioning or otherwise failing by dynamically re-configuring the application to short-circuit the RPC with an error until the remote service begins functioning correctly again. These techniques ensure that spurious failures are mitigated through retries until the application can be reconfigured to avoid invocation to these failing or failed dependencies until the failures are resolved.

While these techniques for minimizing the impact of spurious failures provide reliability at scale, they are primarily reactive approaches, specifically, responding to failures as quickly as possible to either isolate the failure's effect or degrade the application gracefully. However, they do not account for failures observed by the application until the application is reconfigured or failures propagated under that configuration. Therefore, developers are left to answer the following questions:

- What does a service return to its callers when one of its downstream dependencies returns an error due to active load shedding?
- What does a service return to its callers when one of its downstream dependencies is returning an error because the circuit breaker is open?
- What does a service return to its callers when one of its downstream dependencies is slow or returning an error before load shedding or circuit breaking has been activated?

8.3 Why Not Chaos Engineering?

Frequently, to answer questions of resilience, the operators of large, industrial microservice applications turn to chaos engineering. Chaos engineering is a coarse-grained fault injection technique typically performed in an environment where the

entire end-to-end application can be tested — for example, staging or production — pioneered by Netflix when moving to the cloud. While chaos engineering is excellent at identifying infrastructure issues that can adversely affect microservice applications, it is somewhat lacking in its ability to identify application bugs related to resilience. This is due to the risks, cost, and granularity involved in this style of large-scale experimentation.

First, concerning the risks, when an experiment is not well designed or is scoped too broadly, it might adversely affect the customer experience, for example, by preventing a consumer from making a purchase. For Foodly, this is not an option. Second, the cost of running and monitoring chaos experiments can be prohibitive. Therefore, the number of experiments that can be run and the frequency of those experiments are pretty limited. It is impossible to run a set of experiments of this style per-commit or even daily. Even after fully automating chaos experiments and removing any manual overhead in monitoring and rollback, Netflix concluded that they could not run experiments fast enough to cover all the experiments they wanted. Third, if the analysis of these experiments is performed too coarsely — for example, by examining the volume of purchases relative to a control group — minor variations, such as a single small group of customers being unable to purchase due to a latent application bug that is activated by fault injection, may go unnoticed. As active experimentation of this style is prohibitive for Foodly for these very reasons, Foodly chose SFIT to identify application bugs related to resilience, *in development, before deployment* of application code to production.

8.4 How Changes at Foodly are Tested

To set the context of an evaluation of SFIT at Foodly, the testing and deployment practices of Foodly are described. Specifically, one might ask how is an application, like Foodly, comprised of over 500 services with 100 unique deployment events per day tested before a change is deployed?

First, an application of this size cannot be run on a single developer's machine for local testing. This is due to the sheer number of services and database dependencies required to set up an end-to-end *functioning* version of the application. Even using a unique cloud environment for testing is cost-prohibitive in terms of both required effort and time: inherent to the decentralized nature of service development in a microservice architecture, each service may end up with slightly different processes for deployment and configuration. Combined with churn from frequent deployments, keeping a separate staging environment functioning and reflective of the production environment to facilitate realistic testing is complex and, at Foodly specifically, is not done. As a result, Foodly primarily tests changes to services in two ways: in

production with sandboxes or *locally in isolation*.

For production testing with sandboxes, the service code under test is first deployed to a "sandbox" instance running in production, where a developer can attach a remote debugger to the sandbox's JVM instance. At this point, the developer can then issue end-to-end requests from Foodly's mobile or web application with a test account to test the service's features interactively. When issuing RPCs to the service under test, these requests will traverse production dependencies and have those RPCs routed to the sandbox instance. From there, any RPCs issued to downstream dependencies by the service in the sandbox are then routed back to production instances. To prevent data corruption when writing to data stores, multi-tenancy routes writes for *only* those requests to separate data stores or namespaces.

With local, isolated testing of services, automated tests that exercise the service's API are used. These tests use stubs for downstream service dependencies, where the hard-coded responses required for each test are encoded in each stub. It is important to note that these stubs are in-process gRPC services that return these hard-coded responses, not stubs that short-circuit the method invocation with the hard-coded response. Therefore, the test code exercises the RPC framework code that will be executed in production. For downstream data store dependencies, Docker containers are used where they are seeded with the required data for each test: again, these are the same databases that are used in production, just running locally. The only difference from production code paths is that dependency injection changes the IP addresses/port numbers used to connect to dependencies, maximizing the coverage of production code paths. Foodly refers to this style of testing as *component testing*, which resembles functional testing of individual services as a single functional component; the use of a different name was needed to distinguish these tests from the functional tests that exercise end-to-end application behavior, to minimize confusion when discussing different testing approaches.

8.5 Takeaways

Conceptually, Foodly is a prime candidate for evaluating SFIT for several reasons:

- *Minimized Blast Radius.*
  First, Foodly wants to ideally avoid any fault injection testing performed *in production*, as any injected fault in the production environment may adversely affect the customer experience when ordering food. Uniquely, SFIT is designed to identify application resilience bugs by testing *in development* before the application code is deployed into production.

- *Automated Test Generation.*
  Second, Foodly has a large microservice application graph, where individual services can issue upwards of a hundred RPCs to downstream services to process a single customer request, prime for fault injection testing where a single RPC failure can alter application behavior. SFIT avoids the overhead of application developers manually writing these fault scenarios using mocks by automatically generating these scenarios based on functional tests of the microservice application. Similarly, SFIT also ensures that no important fault injection scenarios are missed during testing, providing that each RPC is testing for the set of faults it is susceptible to.

- *Exhaustiveness.*
  Third, and finally, while Foodly has implemented a large number of application resilience mechanisms (*e.g.*, circuit breakers, load shedding, retries) that are all independently tested and provided as libraries, their application remains vulnerable to failures before these resilience mechanisms activate and after failures start occurring. This space is where SFIT shines: identifying all possible faults the application can and will observe.

In the following chapter, SFIT is evaluated on Foodly's application.

Chapter 9

Evaluating SFIT: In Practice

> *"They wanna know how many rhymes have I ripped in rep, but researchers never found all the pieces yet, scientists try to solve the context, philosophers are wondering what's next."*
>
> Eric B and Rakim, *Don't Sweat The Technique*

As demonstrated in the previous chapter, Foodly provides an exciting and very relevant industrial microservice application for evaluating *Service-level Fault Injection Testing* (SFIT) in practice. In this chapter, the challenges with applying the SFIT technique, using both the existing research prototype Python Filibuster and a new re-implementation Java Filibuster, are presented and discussed.

The challenges faced during this evaluation range all across the software engineering spectrum: technical difficulties with the prototype implementation, challenges faced with integration of the technique into existing processes within the organization, and the high cost involved in testing services.

Here, these challenges are presented in the context of testing a single service at Foodly using Filibuster, a continuous process performed over two years (2) as both an independent contractor and full-time employee of Foodly.

9.1 Philosophical Challenges

As discussed in Chapter 8, Foodly primarily tests their microservice application in two ways: manual testing with their mobile application in production using sandboxes and locally, *in development*, using component tests.

Manual testing does not suit SFIT, in that it is too difficult to orchestrate an efficient, systematic exhaustive testing approach that involves manual, interactive

developer actions: for example, by clicking in the UI of a mobile application to perform operations that result in requests to the application. Therefore, it was decided by Foodly to explore the use of SFIT on a service that used component tests, testing services independently and pair-wise against any possible failures that any of their direct downstream dependencies might return. Rather obviously, pair-wise testing of services in a microservice application obviates the need for any test case reduction strategy such as *Encapsulated Service Reduction* (ESR) that is based on how services encapsulate the the failures of their downstream services when responding to their caller(s).

As the predominant communication framework used by Foodly was Google's gRPC framework, which can only return a minimal (16) set of possible error codes, each RPC issued by a service could be tested using SFIT for all error codes that any downstream dependencies of that service might return: this includes error codes that are overloaded for indicating active load shedding or open circuit breakers, allowing SFIT to cover those failure cases as well. This contrasts with the regular application of SFIT, where an application is tested entirely by using a reverse topological sort of the dependency graph to only test upstream services for the errors that their direct downstream dependencies can produce: in essence, by preferring raw compute power during testing to address the restrictions of not being able to test the microservice application wholesale.

Instead, a single service at Foodly was selected, and its existing functional test suite was tested using SFIT. This resulted in a significant number of test failures, as expected, as none of these tests contained assertions regarding application behavior under fault. One test class was selected that contained 39 functional tests, which represented the service's core functionality and accounted for the majority of that service's test coverage, and using the SFIT methodology, iterated on each test as faults were injected. The test failed, and the desired behavior under failure was encoded into the test using SFIT's fault injection predicates. This process was completed when all failure behavior in the application, for the functionality covered by the test suite, was captured by the test, and the test suite passed ultimately under SFIT execution.

9.2 Results Overview

The results were somewhat surprising in applying SFIT to this service at Foodly. First, fewer test failures occurred than expected, due to the use of *soft* dependencies. It was only upon manual inspection of the individual tests that *should have failed with fault injection*, that the following was discovered:

1. First, many functional tests invoked downstream dependencies that were not stubbed. This resulted in RPCs to these unstubbed dependencies returning the error code indicating the RPC method was unimplemented. Not only is fault injection redundant in these cases, but the functional test does not capture application behavior when all downstream dependencies are working correctly.

2. Second, many tests did not assert that invoked downstream dependencies *were* invoked (or stated that the dependencies could be invoked any number of times.) This was independent of whether they were stubbed.

This might indicate one of two things: developers only write tests for the pessimistic case to ensure that applications work correctly when all soft dependencies fail, or the latter, where developers only iterate on tests until they pass. If the latter, it is safe to assume that if developers do not investigate tests sufficiently to stub all dependencies when tests are passing, any tests that continue to pass under fault injection will remain unquestioned, resulting in possible latent bugs in production.

When it came to *hard* dependencies, it was a cumbersome task to encode the same failure behavior in every functional test for all combinations of failures. This is because many of the tests shared a common code path responsible for loading context information related to the customer with several RPCs, each throwing a fatal error under RPC failure. This was then exacerbated by tests that performed setup for the test inside of the test itself, which resulted in fatal faults being injected during test setup and not test execution, requiring that both the test and test setup be updated with fault injection predicates, as described by SFIT.

Finally, the process of updating tests to encode the behavior of the application under fault was generally difficult:

1. It was challenging to understand where an injected fault occurred in a test execution due to the large number of RPCs executed asynchronously by a single test (with some identical RPCs being performed multiple times in a single execution at different points in that execution.)

2. It was challenging to know precisely the correct SFIT API to encode fault behavior.

3. The process of updating tests with the appropriate SFIT API was cumbersome due to duplication of fault behavior across multiple tests and when faults were injected in test setup and tear down.

In the following sections, both the challenges faced when applying SFIT at Foodly and the preliminary results of using SFIT at Foodly are presented. These results

are then used to derive a new design of SFIT, *Principled Service-level Fault Injection Testing* (*p*-SFIT) that is the result of a co-evolution of the *Service-level Fault Injection Testing* design by the author of this dissertation in collaboration with Foodly.

9.3 Experimental Configuration

To evaluate SFIT, one service at Foodly was selected to be tested with FILIBUSTER. In a graph of hundreds of different services, this service has the primary responsibility of subscribing consumers to the premium plan offered by Foodly. This service takes dependencies on several different downstream services over gRPC and several direct dependencies on databases and queues: CockroachDB and Kafka.

This service is automatically tested for each new commit using 3,209 tests, where 1,419 of those 3,209 tests are component tests. There are 936 authored component tests, which result in 1,419 tests: 862 are standard tests, whereas 74 are parameterized tests with a set of provided inputs. These tests have been written and maintained by 30 different developers.

For our evaluation, only the 862 non-parameterized tests were used to understand the broad types of failures exhibited by the system under fault injection and then focus on one specific test file containing 39 specific tests that cover the essential functionality of the service: subscribing users.

9.3.1 Component Tests

Component tests are implemented in JUnit as standard tests with supporting code for starting and stopping the service under test, stubbing dependencies, and starting required database dependencies. Stubbing of RPCs is performed using an in-process gRPC server where fixtures are installed for an RPC method; therefore, RPCs are fully executed instead of directly stubbing the method invocations. Stub usage in component tests is a proxy for (potential) side effects performed by other services. For example, instead of running a dependency and asserting that the state was mutated in that service's database, a stub is used to mimic that service's behavior.

Test authors mostly use a "given, when, then" style when writing tests. However, this style is not enforced by any underlying framework code that implements the separation of these different concerns but is merely a coding style used inside of a single test method with the different sections delineated by code comments.

In the "given" section, developers install required fixtures and stub downstream dependencies. These stubbed dependencies are then automatically checked for the correct number of invocations by the stubbing framework only *after* the test passes successfully. In the "when" section, developers issue an RPC to one of the

APIs exposed by the service. In the "then" section, developers assert the expected response from the RPC and any side-effects that should have been performed. Developers leave this section blank to indicate that an RPC invocation does not throw an exception without any other explicit assertions.

Based on this structure, component tests fail only in three possible ways:

1. *Thrown Exceptions*:
   when the RPC invocation unexpected exception;

2. *Assertion Failures*:
   when the test assertions do not hold; and

3. *Stub Invocation Failures*:
   when stubs are not invoked the number of times specified. While stub invocation failures do surface as assertion failures at the end of execution, these are distinguished frotion failures. as in contrast to assertion failures that can check any arbitrary expression that reduces to a boolean value, stub invocation failures take a specific form, corresponding to a particular type of test failure, and can be automatically checked against the observed behavior of the system, as in *p*-SFIT.

9.3.2 Re-implementing FILIBUSTER

The original goal of this evaluation was to adapt the prototype implementation of SFIT, FILIBUSTER, as necessary to perform testing of services at Foodly.

Architecture and Implementation. As discussed, the original FILIBUSTER prototype is implemented using a language-agnostic design comprised of different components that operate in concert:

- Several **instrumented RPC clients**, implemented in Python to support the Filibuster test corpus but designed to be language-agnostic. These instrumented RPC clients, implemented in the language of choice used by the microservice, communicate with the FILIBUSTER server to register RPCs and inject faults when necessary in a language-agnostic JSON format.

- A **CLI** for executing tests, implemented in Python. The CLI executes the developer-authored functional test in any language that exercises the application behavior.

- A **server**, implemented in Python. The Filibuster server tracks how many test executions are necessary and what faults must be injected for each test execution.

Initially, the thinking was to create instrumented RPC clients for Java simply, the language runtime used by Foodly, and reuse the existing components. However, in the course of iterating on the integration of Filibuster with Foodly's code over two years, the entirety of Filibuster was rewritten in Java.

First, the instrumentation needed for RPC clients was written in Java for Google's gRPC library. Next, an external CLI tool was replaced with integration into the JUnit test framework used by Foodly's functional tests using a custom test annotation and factory. This was done for several reasons.

1. As Filibuster needs to run the functional test multiple times, it was necessary to minimize the cost of starting a new JVM each execution and waiting for Kotlin test code generation to execute. Even with sharing a `gradle` daemon, the build tool used by Foodly, across executions, this startup time was prohibitive, with a single test execution taking as long as 1 minute.

2. Not restarting the JVM and integrating it into JUnit would avoid the need to write a custom tool for aggregation of coverage data across different executions, as this would happen automatically.

3. With JUnit integration, an external Filibuster test harness that lived outside the application code was no longer required. Similarly, this integration avoided the need to pause Filibuster execution to allow an external debugger to be attached when debugging failures from fault injection were no longer required.

Finally, the Filibuster server that communicated with the instrumented clients using RPCs was replaced with an in-process Java implementation of that server. This reduced several challenges we faced with using an external Python process in Foodly's continuous integration environment, reduced execution time by avoiding expensive RPCs between the application and Filibuster server and allowed for more detailed reporting by removing language-agnostic error reporting with the server itself.

From there, it was determined that the style of reporting used by Filibuster's server, duplicated in the Java re-implementation of the server and presented as log entries, made debugging and understanding of what went wrong during a fault injection test too difficult. Therefore, an IntelliJ plugin was designed that allowed application developers to visualize the results recorded by the Filibuster server to make debugging significantly more accessible. This included, but was not limited to,

the test executions generated by FILIBUSTER, the faults that were injected, the request and responses for all issued RPCs, and the test failures that occurred when faults were injected.

Nondeterminism: Data and Scheduling. Most of the functional tests that Foodly writes are free from observable nondeterminism. However, these tests contain significant nondeterminism in their implementation in both data and the underlying scheduling of threads. This complicates the SFIT approach, as the systematic, exhaustive exploration performed by SFIT assumes that RPC invocations can be uniquely and deterministically identified using their request and call site, as well as that the responses to those RPC invocations across different test executions are deterministic. In practice, all of these assumptions were violated.

As discussed, SFIT uses an algorithm called DEI to identify RPC invocations uniquely and deterministically: this enables SFIT to know when the fault space has been fully explored. DEI uses the invoked RPC service, RPC method, call site, and invocation count to determine the precise location of an RPC in the application code regardless of conditional control flow, looping constructs, and function abstraction. This is encoded as a unique and deterministic identifier for that stable RPC across multiple executions of the same test. When concurrency primitives are used, like threads, DEI also encodes the RPC's arguments, called the invocation payload, into this identifier to avoid the use of a specialized research version of the JVM in testing that can control thread scheduling. This is impractical for large microservice applications that use different language runtime versions and support libraries across other services.

At Foodly, component tests are nondeterminism in data and scheduling. Regarding data, tests generate random user identifiers often derived from auto-incrementing fields in databases. These identifiers are included in many RPCs executed by the application, and therefore, RPC arguments and responses are not deterministic across executions. Regarding scheduling, every RPC executed uses a Kotlin coroutine, where a single RPC can begin running on one thread and complete execution on a different thread. Furthering the complexity, Kotlin's coroutines are limited in that stack traces, used to determine the RPCs call site, are not available under coroutine resumption in application code but only available asynchronously through a reassembly process performed by, and available only in, the debugger, when attached. These complicate the generation of stable identifiers for each RPC necessary for exhaustive search by SFIT.

To work around this deficiency, a modified version of DEI was created that is "best-effort." This modified version encodes as much information as is available at the time of the RPC, omitting any information known to be nondeterministic

ban annotation provided to the test itself. In practice, almost all tests contain data nondeterminism in RPCs. However, most of the code tested in this evaluation executes downstream RPCs in sequence, regardless of coroutine usage. When RPCs occur concurrently, they almost always differ by RPC service or method.

9.3.3 Enabling Filibuster

To enable Filibuster for the subscription service, Filibuster was included as a test dependency of the application. This ensured that any code introduced by Filibuster did not impact any of the production code or alter any production dependencies of the subscription service.

A minor refactoring was then performed of the subscription service to allow the Filibuster GRPC interceptor to be added (used for tracking RPC during the reference execution and injecting fault during the generated fault injection executions) only at runtime when running as a test using the dependency injection library used by Foodly: Google's Guice.

This interceptor was specifically installed between the application code and the underlying RPC framework used by Foodly: Foodly's Hermes library. This was done to ensure faults could be injected without being adversely affected by Hermes built-in resilience mechanisms: specifically, circuit breaking and load shedding under failure as well as automatic retry on certain RPC failures. Therefore, this allowed Filibuster to inject faults without those RPCs being automatically retried under failure or risk repeated fault injection interfering with Filibuster's exhaustive exploration algorithm: the goal was to find application resilience issues and not test the already well-tested built-in resilience mechanisms of the underlying RPC framework.

9.3.4 Configuring Filibuster

Filibuster was configured using the following configuration options. First, to ensure termination of the search algorithm within a reasonable time:

- *Data Nondeterminism.*
  Filibuster was configured to adapt the exhaustive search algorithm to deal with the data nondeterminism that is present in component tests.

  As presented, Filibuster uses an algorithm that encodes the RPC's arguments into a unique RPC identifier used during an exhaustive search. This allows Filibuster to search the fault space when scheduling nondeterminism properly is present without control of the underlying language VM scheduler.

Unfortunately, this algorithm is incompatible with tests containing data nondeterminism, as these arguments will vary from execution to execution as all of the tests written by Foodly have data nondeterminism. For example, each test generates a unique user identifier for each re-execution of the same test. Therefore, each invocation of the same test will not execute the same RPCs, and the exhaustive search algorithm has to be adapted accordingly to correlate RPCs between different executions as the same RPC for exhaustive search to terminate.

- *Maximum Fault Injections* (*Upper Bound.*)
  Filibuster was configured to generate, at maximum, 100 fault injection tests per component test. Because of restrictions deep in the design of JUnit, fault injection tests that were not necessary were still generated but were effectively a no-op.

  Specifically, if a reference execution of a given test executed 30 RPCs, Filibuster would generate 30 additional tests (*i.e.*, 1 reference execution + 30 fault executions) to cover all single fault, no permutation scenarios required, with the remaining 70 executions resulting in no-ops. If a reference execution of a test executed 101 RPCs, Filibuster would generate 100 additional tests, skipping the final required fault injection (*i.e.*, 1 reference + 100 faults, one non-executed scenario.)

  No-op executions incurred an execution penalty of 0.15ms per execution. In the worst case, a 15-second penalty per test that executed no RPCs. This was done to place an upper bound on execution time under the assumption that no test would run more than 100 RPCs to downstream services.

Second, to reduce the number of Filibuster-generated tests per component test, two optimizations were done:

- *Fault Permutations.*
  Filibuster was configured to not inject multiple simultaneous faults and only to inject a single fault in each generated test. This was done to keep the execution time reasonable, given that each test issues many RPCs. Specifically, reducing the running time from lying within a polynomial factor, factorial in the number of RPCs to exponential in the number of faults. This optimization was performed by examination of a subset of tests that determined that RPC execution was performed in a sequential workflow with minimal concurrency.

- *Exhaustiveness.*
  Foregoing static analysis, Filibuster was configured only to inject a single

fault: the gRPC `StatusRuntimeException` exception parameterized by code `UNAVAILABLE`, which is most often used to indicate that the remote service is unreachable or returning a connection error.

Again, this was done to keep the execution time reasonable for this naive evaluation given that there was a significant amount of tests to execute using Filibuster and that each test issues a large number of RPCs.

Specifically, reducing the running time to lie within linear execution time based on the number of RPCs executed by a given test.

This optimization was performed by examining a subset of tests that determined that RPC execution rarely contained error handling code conditional on the error code that was returned.

Third, an application-specific optimization was done:

- *Cache Assumption.*
  Using a cache for RPC invocations and responses for a given RPC was assumed. At the time of writing, the service tested was converting from a Redis-based cache to a multi-tier cache in the application. Both caches were turned off by default when executing the test suite. Therefore, to avoid encoding cache semantics into the test harness and to avoid having to restart both Redis and the JVM in between each test invocation, the Filibuster search algorithm was configured to prevent fault injections on RPCs for the same service, method, and arguments if it has already succeeded in the test execution previously, simulating the behavior of this cache. This under-approximation reduces the completeness of the Filibuster algorithm by avoiding scenarios of cache failure mid-request.

It is noted that all optimizations were validated against the subscription process, which accounts for the majority of code coverage of this service. While this is not definitive, it accounts for the programming style widely used in this repository and across the Foodly microservice application.

9.4 Socio-Technical Challenges

In terms of socio-technical challenges, it was found that any modifications made to application code, build processes, or test code to support SFIT needed to be justified for those changes to be integrated into the code base. This involved writing technical documentation, minimizing the impact on existing processes, and producing visual results for understanding the approach's benefits.

9.4.1 Education and Documentation

When it came to integration into the service that was tested with SFIT at Foodly, the author was immediately asked several questions by different developers that all took the general form of "What is this, and how will it impact me?" In short, developers wanted to know what this tool was doing, how it would impact them, and how it would impact their daily workflow. This had to be addressed in several steps.

As modifications were being made directly to test code that application developers of the service were working on as part of their daily work, documentation had to be written on how actually to work with the SFIT tool, what errors they might see, what those errors mean, and what to do about debugging those errors. This involved extensive Javadoc work within the code of the Filibuster Java implementation itself and using Foodly's external documentation tool for writing several interactive tutorials on adapting a tool for use with Filibuster and debugging the resulting failures. In addition, how the Filibuster integration was performed for that service, how it integrated into the IDE, and how it integrated into the build and deployment processes used for feature development all had to be documented.

During this process, several iterations of the Filibuster API were made based on feedback from the team that worked on the service we were testing. From there, a "Request for Comments" (RFC) document was produced that was distributed to the broader team with what we assumed would be the finalized API, which resulted in several additional iterations of the API. As part of this, several detailed discussions were had with team members to justify why this approach was better than the alternative approaches for fault injection testing and why seemingly "cleaner" approaches would not scale within Foodly.

9.4.2 Development Processes

Regarding the local development experience, several different IDE integrations were built based on developer feedback. It was discovered that developers wanted a mostly opt-in experience, where they did not run these tests automatically when running the test suite of their application. In short, they did not want any process changes made that may impact their local development experience. When the author inadvertently made these changes early during our initial integration, they were immediately asked to revert them. A design was ultimately settled where a single test could be run with fault injection optionally in IntelliJ but defaulted to running that test without fault injection. It is believed that this reflects the methodology of this style of test development: iterate quickly to get the test passing and then run with SFIT to identify resilience bugs. In practice, there is no evidence to indicate if

this integration style will ever result in a developer running the tests with SFIT, but instead defer to resilience bugs identified in continuous integration.

As observed when integrating SFIT with continuous integration, developers are hesitant to adopt any process that will increase a perceived "slow" build time for their application in continuous integration. Therefore, any proposal where a new technique would increase code coverage, for example, at the cost of 16 additional tests for every single RPC executed by a given test of several thousands of tests, was met with pushback. Therefore, it was necessary to develop a strategy where SFIT could still be used but without impacting the daily workflow of developers. The strategy ultimately implemented involved performing nightly SFIT runs on the most recent commit to the 'main' branch of the repository and reporting failures using a Slack notification on failure. While this appeased the developers of the service where this testing was being performed by avoiding an expensive SFIT run for each commit during feature development, it was found that both caused errors to be ignored by most developers and made correlation between SFIT failures with the change set that introduced the regression difficult. At this point, it remains unclear what the correct balance is here, as traditional approaches for regression identification (*e.g.*, `git bisect`) can be prohibitively expensive due to the overhead involved in SFIT-style testing.

It was also discovered that integration with code coverage was helpful in convincing developers of the value of SFIT testing. For example, it is possible to visually show developers the increase in code coverage provided by SFIT through fault injection. To do this, statistics were reported using Java CodeCov using a specific tag that could be used to differentiate the non-SFIT test suite execution against the SFIT test execution visually and see the increase in coverage.

9.5 Results

The application of Java Filibuster resulted in many failing tests. This was expected as none of the existing tests were adapted to encode the application's behavior under failure.

Of the 862 component tests, Java Filibuster identified 546 fault injection scenarios that resulted in test failures, with two of those failures also appearing in the reference execution but not in any fault injection execution, indicating that these tests were flaky. These failures were consistent with our analysis of how component tests can fail: *thrown exceptions* for the failure of hard dependencies *assertion failures*, and *stub invocation failures* for the failure of soft dependencies.

- 367 of 546 (67.22%) failures were because the test expected a response, but

a gRPC exception was thrown. This indicates a *hard dependency* failure as an exception was thrown, indicating that the request could not be processed.

> ✓ **Therefore, *p*-SFIT should provide a mechanism for directly identifying RPCs that are hard dependencies and the thrown exceptions that their failure produces.**

- In 85 of the 367 (23.16%), the same exception was thrown on hard dependency failure. However, it was parameterized with four different status codes depending on the failed dependency. Similarly, in 304 of the 367 (82.83%), the same exception was thrown on hard dependency failure. However, it was parameterized with the same status code with different exception messages, each to indicate which hard dependency failed. This indicates that different hard dependency failures may arbitrarily return the same or different status codes.

  > ✓ **Therefore, *p*-SFIT should require that developers indicate the assertions that should hold under a thrown exception *by status code* to ensure that status codes have consistent meaning and can be treated uniformly by the upstream, calling service.**

  In contrast, *p*-SFIT should not allow the developer to encode behavior based on the contents of the exception message, as that is meant only for debugging.

- 110 of the 546 (20.15%) failures were because the test failed an assertion after returning a successful response to the tested RPC method. This *may* indicates that a *soft dependency* failed and the service gracefully degraded. However, as assertions may be written arbitrarily, it may also indicate the failure of a hard dependency. A random sampling of the 110 failures revealed that many of these errors were directly related to graceful degradation: for example, by assuming a customer's location is the USA when it cannot be determined due to RPC failure or by not displaying five advertisements on a landing page if only 4 of the RPCs that load the advertisements succeed. This indicates that both *hard and soft dependency* failures can be identified by assertion failures.

> ✓ **Therefore, *p*-SFIT should require that on both hard and soft dependency failure, developers explicitly indicate the assertions that should hold.** This ensures that when soft dependencies fail and fallbacks are used, they are captured in tests; when hard dependencies fail and throw exceptions, the system remains consistent based on the thrown exception type.

- 61 of the 546 (11.17%) failures were because the test received a successful response from the tested RPC method, then passed all assertions, but failed when RPC stubs were verified for the correct number of invocations. This indicates a *soft dependency* failure only detectable through missing stub invocations.

> ✓ **Therefore, *p*-SFIT should require that developers indicate precisely how many times every stubbed RPC should be invoked.**

A single file that exercised the core functionality of the service (39 tests) was then selected, and the tests were subsequently updated to contain the application's behavior under fault using SFIT's fault injection predicates.

> ✓ ***p*-SFIT should provide recommendations to the developer on how to update their tests to contain the application's behavior under fault.**

In every test, it was necessary to encode the failure of two *hard dependencies* in shared code paths for thrown exceptions: if the RPC failed to look up the customer or their location. Then, for *soft dependencies*, the invocation count had to be conditionalized based on the fault injections on either the stubbed RPC or any RPC failure that would prevent the stubbed RPC from being invoked due to control flow that was conditional on previous RPCs succeeding.

Out of the 39 tests that were encoded with the application's behavior under fault, the following was identified:

- 33 of the 39 tests (84.62%) had to be adapted for the exception thrown when the location lookup failed. 34 of 39 tests (87.18%) had to be adapted for the exception thrown when the customer lookup failed. This indicates that *hard dependencies* are often used in shared code paths executed by multiple tests.

> ✓ **Therefore, *p*-SFIT should provide a method for sharing the encoded failure behavior of the application easily across different tests.**

- As expected, 7 of 39 tests (17.95%) had to be modified in the execution block to account for downstream dependencies where faults were injected. Still, they had no impact on test outcomes other than asserting they were invoked. However, 9 of the 39 tests (23.08%) had to be adapted to verify that, on a thrown exception, *other* downstream dependencies were not invoked by modification of their stub invocations using conditional code on fault injection. This indicates that *hard dependency* failures that result in thrown exceptions prevent the invocation of subsequent, stubbed RPCs.

  > ✓ **Therefore, *p*-SFIT should require that on thrown exception, developers re-assert how many times stubs should be invoked.**

  This ensures that side-effects performed by RPCs are *not* completed when an exception is thrown, potentially leaving the system in an inconsistent state.

Finally, out of the 39 tests encoded with the application's behavior under failure, not a single test stubbed *all of the soft dependencies it invoked*. Instead, developers duplicated the same test repeatedly only to, in each derivative test, only stub the RPC and assert that the stub was invoked.

> ✓ **Therefore, *p*-SFIT should require that happy path functional tests stub all invoked RPCs and require those stubs to return successful responses.**

Using SFIT, it was clear that soft dependency failure can be the root cause of latent application bugs. Of the 39 tests, 3 of these tests (7.69%) revealed the presence of 3 unique application bugs. Two of these bugs resembled the bug in our motivating example, with the third bug initially appearing as a duplicate of the second. Still, a third independent bug was discovered in the existing test suite when fixing the second bug, indicating that SFIT can help surface interesting, unexplored failure scenarios.

With the first failing test case, as in our motivating example, a failure of RPC A inhibits the invocation of some RPC B. In this specific case, RPC B's failure is logged and reported. However, control flow prohibits the invocation of RPC B when RPC A fails. When RPC A fails, its failure is swallowed, and nothing is logged: RPC A

can fail by returning both a response indicating that a record is not present and a communication failure. When investigating the possible bug, it was determined that an offline process is used to *compensate* for the failure of both RPC through a manual reconciliation process using Foodly's data warehouse.

With both the second and the third failing test cases, the same shape appeared in the same code path: some RPC X fails and inhibits the invocation of some (different) RPC Y on failure. However, in the case of the third failing test, the service developers had written a test explicitly for the specific negative case of RPC Y failing. Therefore, during the investigation, it was determined that the X-failure-inhibits-Y bug should be repaired like the strategy employed in the test case that specifically modeled the failure of Y. What was identified was surprising: not only did the test case that developers had written to test the failure of RPC Y mishandle the failure explicitly, but it was an active bug that had adversely affected customers as recently as five days before the bug investigation. SFIT had not only identified a potential failure case, *but had alerted us to how the existing test suite was written incorrectly.*

In all of the identified cases, the root cause was clear: developers had explicitly mapped the failed RPC into either a `null` or default response of the appropriate type. This variable was passed throughout the application, causing the failure to change the control flow and bypass subsequent RPC invocations.

✓ **Therefore, *p*-SFIT should require that test code indicate how many times downstream dependencies, which do not affect the RPC's response (as in both cases a *successful* response was returned) should be invoked.**

9.6 Takeaways

As demonstrated in this section, SFIT, as designed, suffers from several interesting deficiencies that would have remained undiscovered without evaluation in an industrial context on a real-world industrial microservice application.

First, and most notably, industrial microservice applications are much too large and complicated to be tested wholesale. Therefore, any fault injection approach must test services in isolation, exhaustively for the faults they are susceptible to based on their downstream dependencies.

Second, without enforcing that application developers address each injected fault, it is much too easy for developers to write tests of services that ignore injected faults that do not result in test failures. For example, where an application developer fails to write an assertion on behavior conditional on whether a fault is injected.

Third, the overhead in determining how to encode failure behavior into application test code is high. Therefore, it is necessary to provide a mechanism for developers to understand where a fault is injected quickly, what the impact of the injected fault was, and how to, if desired, encode that behavior into the tests of their microservice application.

Chapter 10

Microservices: Dependency Type Evolution

"We all exist in our own personal reality of craziness."

Alejandro Jodorowsky

Armed with the results from the industrial evaluation at Foodly, presented in Chapter 9, the context can be switched back to research to address the deficiencies in the existing *Service-level Fault Injection Testing* (SFIT) approach.

More specifically, with the knowledge of how microservices are tested in industrial settings, the challenges with adapting industrial microservice application tests for resilience, and how both *hard* and *soft* dependencies are employed, and tested in industrial microservice applications, a new process for testing microservice applications can be designed.

However, first, it is necessary to design a new application for the microservice application corpus that reflects the ground truth of industrial microservice applications. In this chapter, application is presented using strictly hard dependencies: this application is inspired by application design and behavior observed at Foodly. Then this application is evolved, in the same manner as done at Foodly, to change *hard* dependencies to *soft* dependencies with the goal of improving application resilience. This evolution introduces latent application bugs, prime for detection with SFIT.

10.1 Application Structure

Consider the microservice application, depicted in Figure 10.1 comprised 6 services, each with its own associated data store:

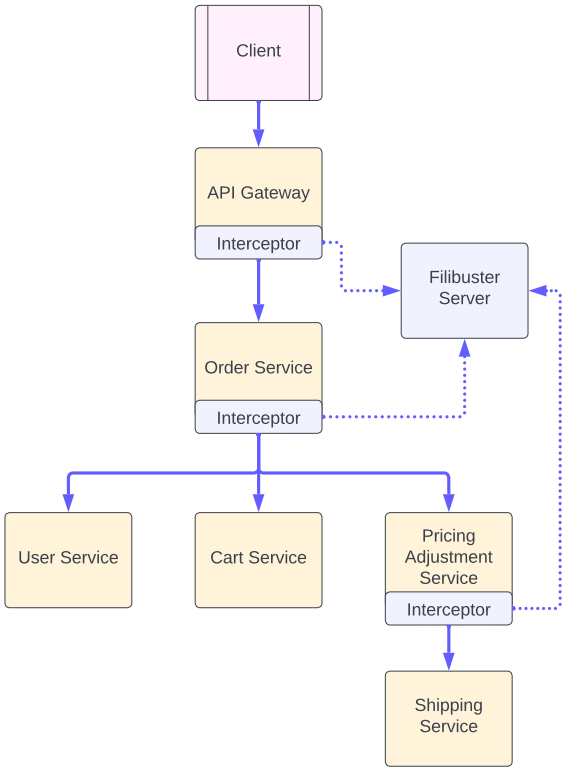

Figure 10.1: Purchase Application Structure with Filibuster Instrumentation

- **API Gateway:**
  For handling consumer requests;
- **Order:**
  For handling purchases;
- **User:**
  Mapping active sessions to users;
- **Cart:**
  Mapping sessions to shopping carts;
- **Pricing Adjustment:**
  Determining if a cart is eligible for a pricing adjustment, in terms of additional fees or discounts; and

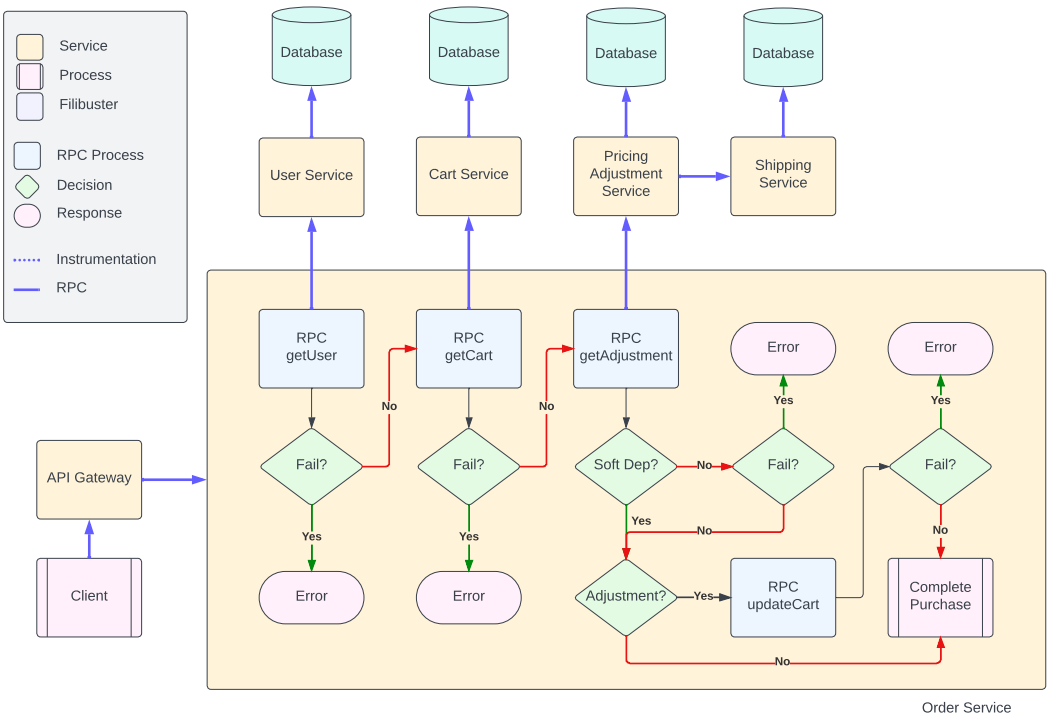

Figure 10.2: Purchase Application Workflow with Graceful Degradation

- **Shipping:**
  For shipping.

While this example reflects behavior at Foodly, the example is fictional: it is considerably simplified in this paper for both presentation and confidentiality. Specifically, services have been renamed, and additional services, RPCs, and business logic that is unnecessary for exploring RPC failure's impact on microservice resilience have been removed.

When the order service receives an RPC from the API gateway, it first issues an RPC to the user service to retrieve the user's information. Once that is complete, the order service then issues an RPC to the cart service to retrieve the user's shopping cart. Then, the cart and user information is sent in an RPC to the pricing adjustment service from the order service. If and only if an adjustment is returned, a subsequent RPC is issued to the cart service to adjust the pricing for that shopping cart. When all of that is complete, the purchase is confirmed through additional RPCs and database writes that have been omitted. This workflow is depicted in Figure 10.2.

10.2 Hard Dependencies

In the first iteration of the purchase application, all of the services that RPCs are made to are *hard dependencies*: in that if any of the RPCs fail, an error is returned to the API gateway which is propagated back to the customer and asks them to try their request again.

Let us explore what happens in a bad deployment of the pricing adjustment service. For example, consider the case where the pricing adjustment service is deployed with a bug that causes null pointer exceptions whenever an RPC is received to adjust the user's cart. This bug may not be caught in testing because it is only triggered by a mismatched dependency in production.

When this code is deployed to production, it is incrementally rolled out: upgrading a single service instance at a time, in bulkheads [Mic23b] that partition traffic to isolate the effects of a bad deployment to a single partition. As each partition is upgraded, it passes standard health checks and probes (*e.g.*, readiness, liveness) [Kub23] indicating that the service can respond to requests. However, these probes may only be shallow checks that verify the service is answering requests but do not exercise each code path. As soon as the code is live and begins receiving user traffic, developers will be alerted to failures as Service-Level Indicators (SLIs) and Service-Level Objectives [JC18] will begin to degrade, indicating both increased error rates to the pricing adjustment service, the order service, and the API gateway.

At this point, the deployment will be aborted, and the deployment version will be rolled back. But, before this can occur, circuit breakers [aut22] on the order service will fire and reconfigure the service to avoid calling the failing dependency. This allows developers to provide special-case error messages to upstream callers and removes load on the failing service, as the root cause may be unclear. Similarly, had this been related to a feature rollout, this same process would occur when enabling the feature using a feature flag. In this case, the resilience mechanisms work as expected: bugs are mitigated quickly with minimal impact on customers. Requests were aborted with errors, and clean-up of the incorrect state is minimal.

10.3 Soft Dependencies

With many industrial microservice applications issuing over 100 RPCs to handle a single customer request, developers of these applications tend to classify services into *tiers* [Atc19]:

- *Tier 1*,
  Representing the critical services;

- *Tier 2*,
  Representing essential business functions, and
- *Tier 3+*,
  Representing services that, if unavailable, have a hard-to-notice impact.

For example, in our purchase application, one might classify the user and cart services as Tier 1: the application cannot complete a customer's purchase without these services. However, one might classify the pricing adjustment service and its dependencies as Tier 2: important, but its failure should not prevent a user from making a purchase, as the purchase's payment amount could always be adjusted later. Therefore, one can think of the pricing adjustment service as a *soft dependency* of the order service. Specifically, should it fail, the request should proceed without the failure impacting the purchase.

Soft dependencies may prevent the mitigation of bad deployments by gracefully degrading features or using reasonable default values upon failure. To demonstrate, consider the case where the pricing adjustment service is deployed with a bug that causes null pointer exceptions whenever the pricing adjustment service receives an RPC, and *only when the user is eligible for a pricing adjustment*. When the null pointer exceptions *do* occur, the customer can still make a purchase, however will not receive a pricing adjustment.

As expected, soft dependency failures may be more difficult to identify after deployment for several reasons. First, the failure does not cause the SLIs of the order service and API gateway to change, as failures are swallowed, and customer requests are processed successfully, albeit without an adjusted price. Second, while the SLIs for the pricing adjustment service will be affected, given that only a subset of users will be eligible for a pricing adjustment, this failure may go unnoticed, unless someone is actively observing these metrics during deployment. Still, circuit breakers will activate with a sufficient error rate, and once the SLI is investigated as part of the deployment, the bad deployment will be rolled back. Cleanup is a bit more complex, as some customers will have placed purchases and paid the incorrect price. However, if a metric was logged for each error, manual remediation can be performed for each logged error. Every failed RPC reflected a single customer who should have received an adjustment.

10.4 Latent Resilience Bugs

In contrast to active application bugs, *latent* resilience bugs – bugs that are introduced in a bad deployment but that are later activated by the failure of a dependency – can be more challenging to mitigate as they will not be detected during deployment but

at some later time. When combined with soft dependencies, they can be extremely difficult to detect in production if failures are either infrequent or otherwise hidden by a service's error budget.

For example, consider the case where the pricing adjustment service contains a latent bug involving a downstream dependency. With this latent bug, a timeout is misconfigured with respect to the order service's timeout, and therefore when the pricing adjustment service's dependency slows down, a timeout error is propagated back to the order service. This could happen for any number of reasons related to the service or underlying infrastructure. Given that the pricing adjustment service is not a hard dependency, the customer's purchase is completed without the proper pricing adjustment and must be remediated. However, identifying which purchases require remediation may be difficult, as the remediation process must now discriminate between legitimate and spurious failures.

10.5 Takeaways

As described in the background and related work material, one of the limiting factors in research on microservice applications is access to industrial applications.

In fact, as demonstrated in our evaluation of SFIT in practice, this resulted in several mismatches between the SFIT design and how industrial applications are tested, resulting in difficulties in applying the approach. Even when those difficulties were overcome, challenges were still faced related to gaps in the SFIT approach: SFIT assumes that tests will fail when injecting faults and non-failing tests may result in a lack of investigation of the impact of faults.

To address this, the learnings from this industrial evaluation are used in the co-evolution of the SFIT approach, with Foodly, to derive a new synthetic example that can be used to further microservice application resilience testing. This resulted in the design of a new approach, *Principled Service-level Fault Injection Testing*, presented in the next chapter.

Chapter 11

Principled Service-level Fault Injection Testing

> *"Everybody's a mad scientist, and life is their lab. We're all trying to experiment to find a way to live, to solve problems, to fend off madness and chaos."*
>
> David Cronenberg

Principled Service-level Fault Injection Testing (*p*-SFIT) is a software development process that extends the SFIT approach for microservice applications that are designed to degrade gracefully under failure. It provides a structured method for *writing structured happy path functional tests* – test cases that properly capture application behavior when the application contains no faults – and then *updating those tests to also account for the application's behavior under failure* – once fault injection is applied.

It does this with a development IDE plugin that visualizes fault injection scenarios and provides recommendations on accounting for failures in test code. *p*-SFIT forces developers to make explicit their application's behavior under fault and alleviates the burden of having them think about how their application may be impacted by RPC failure by auto-generating the failure scenarios.

This innovation was necessary, as SFIT leaves several questions unanswered and in the hands of the developer to solve. *p*-SFIT answers these questions by proposing a testing process that extends SFIT. This process was designed alongside an industrial microservice application to remain grounded in how these applications behave and are tested.

Regarding *writing structured happy path functional tests*, two questions remain unanswered by SFIT.

1. How can one write these happy path functional tests in a way where, when applying fault injection, they avoid executing *uninteresting* fault scenarios: for example, where they inject failures in setup, assertion, or tear down code?

2. How does one know if these happy path functional tests, which they will use as the basis for the SFIT process, are written in enough detail to capture latent bugs activated by the failure of a downstream dependency?

Similarly, when it comes to *updating those tests to account for the application's behavior under failure*, three questions remain unanswered by SFIT.

3. With soft dependencies that may not provoke a failure with fault injection, how does one ensure that a developer has specified and considered its failure behavior?

4. How does one make updating existing functional tests to capture failure behavior easier?

5. How does one avoid requiring developers to encode every possible combination of failures in their test code?

To answer (1), the possibility of generating fault injection scenarios that are not interesting should be minimized when testing a service for its behavior under fault. This avoids tests where faults are injected in downstream RPCs executed as part of the setup, tear-down, or assertion code. For example, the test code that makes a purchase and the assertion code that verifies a purchase invoke the same downstream dependency to look up the user. In this case, only injecting faults on the lookup for the purchase.

To answer (2), tests should also start from a known good place. Tests should not invoke downstream dependencies that are either not stubbed or not started and return an error. Instead, tests should either start or stub all downstream dependencies reflecting the "happy path" case where all dependencies return non-failure responses. Fault injection should then be used to generate the required variations of the test where faults are injected for those RPCs.

Similarly, if downstream dependencies are stubbed, a precise expected invocation count for that stub should be supplied unless that dependency is specifically indicated as a read-only dependency without side-effects. For example, a method to retrieve the user may be invoked an arbitrary number of times if it only reads from a database and returns the response. In contrast, a dependency that performs a write operation on a database should have its necessary number of invocations specified precisely. This is necessary to prevent both under-invocation and over-invocation in the case of non-idempotent endpoints that do not de-duplicate requests.

To answer (3), developers should be forced to consider every injected fault and encode the behavior of the system when those faults are injected. This should be done lightweight, indicating that the fault has no impact, results in different assertions holding, or results in the upstream caller receiving an exception. This is necessary because existing assertions are often too weak or permissive of faults. In short, without explicitly encoding that a fault has no impact, the lack of a failure may indicate a *false negative*.

To answer (4), the test interface should allow for reuse across multiple tests that test either the same API with different arguments or system configurations (*i.e.*, feature flags.) This is necessary when the APIs share common code paths, where re-encoding of failure behavior would be redundant. Further, to ease the process of encoding this failure behavior, developers should be provided with "hints" on how to encode the failure. For example, suppose the failure of a hard dependency returns an exception. In that case, the developer should be provided with a recommendation or code snippet that they can easily use to convert their test to encode that behavior under fault.

To answer (5), compositional reasoning *within a service* should be used wherever possible to avoid redundant encoding of failure behavior under combinations of faults in the same test execution. For example, suppose the developer indicates, in isolation, that a failure of RPC 1 has no impact on the test outcome and that a failure of RPC 2 causes the system to throw an exception. In that case, the tool should attempt to check that the simultaneous failure of RPC 1 and 2 causes the system to throw an exception, without requiring that the developer explicitly encode this. Furthermore, the developer should only be required to encode behavior where the application behavior is ambiguous. For example, when the failure of RPC 1 and RPC 2 each have been independently specified with two different sets of assertions holding.

11.1 Overview of *p*-SFIT Approach

To realize these requirements, a testing process that integrates with the existing testing procedure of SFIT is presented. The testing process is composed of both developer actions (in *italics*), actions performed by *p*-SFIT (in **bold**), and actions performed by SFIT (in normal).

When developers are *writing structured happy paths functional tests*, the process is as follows:

1. Execute the test without fault injection.
   Once the test is passing:

a) **Fail the test if any invoked dependencies return a non-success response.**
   i. *Update test to stub RPC with response.*
   ii. **Go to Step 1.**

b) **Fail the test if any stubbed dependencies do not explicitly indicate a precise invocation count.**
   i. *Update stub to indicate expected invocation count. p-SFIT will not accept use of imprecise counts: "any # of times" or "at least once."*
   ii. **Go to Step 1.**

When developers are *updating their happy path functional test to account for the application's behavior under fault*, the process is as follows:

2. Execute fault injection scenario from SFIT.
   Did the developer encode application behavior under fault for the injected faults?

   a) **If not, fail the test for unspecified behavior and provide recommendations on updating the test to encode the application's behavior under fault based on the observed behavior.**
      i. *Update test to encode the application's behavior under fault. Figure 6.2 provides an example of this in SFIT.*
      ii. **Go to Step 1.**

   b) **If so, check that application behavior matches the developer-specified behavior.**
      If false, ask the developer to investigate a possible application bug and provide recommendations on encoding the observed application behavior under fault into the test.
      i. *If a bug, resolve.*
      ii. *If not bug, encode application's behavior under fault using recommendations.*
      iii. **Go to 1.**

3. If more fault injection scenarios remain, go to 2 **only if the current test execution passed based on the developer specifications of application behavior under fault.**

p-SFIT differs from SFIT in that, rather than running all possible fault injection scenarios immediately and showing all fault injection executions and how they failed, *p*-SFIT only executes fault injection scenarios until it encounters the first

test failure. The developer is then prompted to update the test to account for the application's failure behavior before proceeding, refining that behavior over time as new fault injection scenarios are introduced and avoiding the scenario where developers encode behavior that may be derived through compositional reasoning for example, if they were to start with the last fault injection scenario, which contains the most significant number of simultaneous faults.

The *p*-SFIT testing process, just like SFIT, always re-executes both the happy path execution and all fault scenario executions each time it is run. This is done because the developer may have inadvertently made a change to either the code (to resolve an identified bug) or the test (during the process of updating the test to capture the application's behavior under fault) that causes either a previously passing fault scenario (or the happy path execution) to fail.

11.2 Components of *p*-SFIT

Our design comprises four major components to meet these requirements and realize our testing procedure.

1. a *test interface* that improves on both the *writing of happy path functional tests* and *updating those tests to account for the application's behavior under fault:*
    a) avoiding the generation of "non-interesting" failure cases resulting from RPCs executed in test setup, assertion, and tear-down code;
    b) maximizing reuse of failure specifications; and
    c) warns on the under-encoding of dependency failure.
2. an API for *updating happy path functional tests to account for application behavior under fault* that supports precise failure specification for all possible failure scenarios of both hard and soft dependencies. This API is then mechanically checked by *p*-SFIT to ensure that all injected (single) faults have a corresponding covering specification;
3. an algorithm for *compositional reasoning* within a service. This algorithm allows for the encoding of failure specification under single faults and the automatic checking of failure scenarios under simultaneous faults when the desired behavior is unambiguous and
4. an IntelliJ *IDE plugin* that visually depicts:
    a) test executions and the RPCs that they execute to downstream dependencies;

b) the specific RPCs where SFIT fault injections have been performed in each execution; and

c) code snippets and recommendations with integrated JavaDoc on how to encode failure behavior for a test failure, based on the precise injected fault, for use when that behavior is desired under fault.

These four components are described in the following sections.

Transitive Service Encapsulation Testing. However, before describing these four different components, it is important to note that a complimentary algorithm to ESR was designed during the development of *p*-SFIT that is compatible with the component testing style.

In this algorithm, the results from comprehensively testing a service for failures to its downstream services would produce a fault injection configuration that was based on the observed responses returned from the service-under-test (say Service *B*). This configuration could then be used for testing a different service that took the tested service as a dependency, in order to avoid exhaustive testing of that service (say Service *A*, which calls Service *B*) directly.

This obviates the need for a static analysis, however only under the assumption that testing all terminal nodes in a graph exhaustively yield all possible error responses that those services can return. Thus, any upstream service must only be tested for those faults (plus a standard set of timeout and networking failures) using a depth-first search of the microservice graph, recursively as the graph is traversed. In essence, this leverages the service encapsulation property, compositionally, by only synthesizing necessary tests; this is in direct contrast to ESR which performs the reduction of redundant test cases based on observation of service behavior.

As an example, consider the Audible example presented in Figure 3.1. By exhaustively testing Audible Download Service for failures of all of it's downstream dependencies, and then using the information learned from the Audible Download Service's responses to it's upstream service when faults are injected, the Content Delivery Service need only be tested for the failures *can be actually returned* returned by the Audible Download Service, it's downstream. This corresponds to the results presented in Chapter 10.

While it can be shown that this new algorithm is equivalent to the ESR algorithm in terms of the number of tests required for exhaustive search[1], its discussion omitted from this dissertation. This is because the decentralized nature of microservice application development, the independent development cadences of individual services, and required infrastructure for sharing this information across different

[1]Implemented in the Filibuster prototype in the open-source implementation.

repositories and different teams, all make it more operationally challenging (in theory, but not evaluated) than just running the additional fault injection experiments required for exhaustive search of all error code during testing. Similarly, it was observed that there was little interest in this feature from application developers, once built, while considered novel.

11.2.1 Structured Test Interface

To ensure structured happy path functional tests, an interface where explicit blocks are used to delineate the stages of the testing lifecycle is proposed.

To address 1(a), explicit blocks are proposed for each stage of testing (*e.g.*, setup, assertions, tear down) to know precisely when fault injection should be used to avoid injecting faults when setting up the test or performing assertions that happen to use the APIs that are under test.

To address 1(b), each block will support abstraction via inheritance, allowing developers to quickly create test variants that change the RPC method's arguments and slightly augment the assertions under fault-free or faulty executions.

This is necessary for two reasons: shared code paths and feature flag-based variations. With shared code paths many RPC methods share a common code path when handling requests that perform request validation and retrieve required information from downstream dependencies using RPCs, for example, the user's data. With variations based on feature flags, many tests are variants of the same test with either the same or slightly different assertions. For example, testing an experiment (*i.e.*, A/B test) that invokes the same RPC method with the same arguments and performs the same assertions under a different configuration.

To address 1(c), the developer will be required to stub any invoked RPC method and indicate precisely the number of times that the RPC method will be invoked if that dependency is not either marked as read-only or not started as an external service. This prevents the developer from writing tests where dependencies fail in the "happy path" without knowing how many times they will be invoked. These indications of desired invocations count will then automatically be adjusted (*i.e.*, required invocation calculate one automatically is converted to 0 when a fault is injected on that specific stubbed RPC.) when faults are injected to avoid the developer having to perform that adjustment manually as required in SFIT.

11.2.2 Behavior-Under-Fault Encoding API

p-SFIT requires that developer state the behavior of their application under all RPC failures to avoid assumptions around how an application's behavior changes under failure. Therefore, when updating tests to account for the application's behavior

under fault, *p*-SFITshould allow the developer to do this for both hard and soft dependencies, and at increasing levels of precision (*e.g.,* from any fault for a specific RPC to a particular fault for a specific RPC with specific arguments), as necessary by the *p*-SFIT testing process.

When it comes to *hard dependencies*, where a fault causes the API to return an error, the developer must indicate so. This ensures that any instances of error suppression, where errors are re-written before being returned, are explicitly encoded in the test. This forces the developer to enumerate the exceptions returned to the upstream on fault. Similarly, for any fault propagated directly back to the upstream without any suppression (*i.e.,* unhandled exception), the developer must indicate this behavior is desired. This is then verified by *p*-SFIT by ensuring that when a fault is injected, it surfaces as an error in the test, thus avoiding cases where refactored code inadvertently introduces suppression.

In these cases, the developer also will be forced to reassert how many times *any subsequent downstream dependencies* will be invoked by indicating whether they are *read-only*, where they may or may not happen when that error is returned, or *side-effecting*, where they must(not) happen under any error returned to the upstream. This explicit design is to automatically identify two different types of bugs: partially applied side effects or a lack of required side effects. For the former, identify the case where an error is returned to the user, but some side effects have been performed. For the latter, identifying the case where success is returned to the user but some side-effects have not been performed because of a fault.

For each returned error, developers must then encode the assertions that should hold regardless of the fault that resulted in that error. For example, two different errors result in a service throwing a gRPC exception with status code `NOT_FOUND`. This is done to ensure that the system state is consistent for any faults that cause the same exception to be thrown and to ensure that side-effects are explicitly indicated for each exception. This avoids situations where the same exception is thrown. Still, the system is left in different states, with side-effects partially applied (or worse, un-applied), making it easier for the upstream, who received the exception, to know whether it should retry the request.

Regarding *soft dependencies* — dependencies that, upon failure, do not cause the upstream to receive an exception — the developer must indicate whether the test's existing assertions continue to hold or provide different assertions. This ensures that developers think about the impact of failures by forcing a test failure.

11.2.3 Compositional Reasoning

While the developer should indicate the behavior of their application under fault, requiring that they encode the precise behavior under all possible combinations of

faults is impractical. Therefore, some ability for compositional reasoning is required to keep the testing process tractable and minimize the human cost.

The *key observation* that enables compositional reasoning is the following. First, as described, RPC failures can have either no impact if a soft dependency, result in different system assertions holding if a soft dependency, or thrown exceptions in the case of hard dependency. The combinatorial explosion inherent to this style of testing occurs when soft dependencies are failed simultaneously together. Therefore, our *key insight* in reducing overhead in encoding failure behavior is the following: *any SFIT-generated fault injection executions that contain multiple faults, where at least one of those failures has no impact, should be observationally equivalent to the SFIT-generated fault injection execution that does not contain the faults that have no impact.*

In short, if two RPCs fail in the same execution, that when tested individually in separate executions did not cause assertion failures, and their simultaneous failure should also not cause assertion failures. However, this may not be generally true for all microservice applications, as developers cite code conditional on whether or not two fail, itures occurred in the same execution, hence that they only *may be* observationally equivalent. Therefore, rather than *skipping* the test execution, *p*-SFIT executes the test assuming that the test outcome will be observationally equivalent; if this isn't true, the test will fail assertions normally and the developer will be prompted to encode the behavior for this combination of faults. In practice, this manifests itself in several different ways.

To demonstrate, consider the most straightforward case: RPCs that result in thrown exceptions or a particular set of assertions holding when a fault is injected combined with other RPCs that have no effect under failure in the same execution. In this case, *p*-SFIT will assume that the same exceptions will be thrown or the same set of assertions will hold under any of these combinations and explicitly test them for this behavior. In the case that the test fails, the developer will then be prompted by *p*-SFIT to explicitly encode the failure behavior for this combination of faults.

In more complicated cases, RPCs that result in thrown exceptions or a particular set of assertions holding when a fault is injected are combined with other RPCs that result in thrown exceptions or a particular set of assertions holding when a fault is injected. In these cases, the system cannot determine which exception will be thrown or what assertions will be true at the end of the execution. Therefore, the developer will then be prompted by *p*-SFIT to explicitly encode the failure behavior for this combination of faults.

In all cases, *p*-SFIT always executes the test and verifies the outcome: therefore, no test case reduction is performed, but instead it is preferred to avoid burdening the developer with writing the additional, redundant assertions if possible.

11.2.4 IDE Plugin

In order to assist application developers in debugging failed tests and encoding failure behavior, an IntelliJ plugin was developed where the *p*-SFIT process is visualized: every test generated, each RPC invoked by each test, and the location of every injected fault. This plugin also provides recommendations on how to resolve failing tests based on both the injected fault and the surfaced failure.

11.3 Implementation

The proposed design of *p*-SFIT is realized as a Java interface that each JUnit component test will implement: `FilibusterGrpcTest`. This interface requires that each test implements several abstract methods, each corresponding to one of the blocks representing a stage in the testing procedure run by *p*-SFIT.

Executed in this specific order, there is a block for each of the following stages of the testing procedure.

1. `setupBlock`
   Test setup;
2. `stubBlock`
   Stubbing of downstream dependencies;
3. `executeTestBlock`
   Execution of the service RPC method that is under test;
4. `assertTestBlock`
   Assertions on the response of the RPC method that is invoked in the test block;
5. `assertStubBlock`
   Assertions on the invocation count of installed stubs and
6. `teardownBlock`
   Test tear-down.

Finally, there is a failure block (`failureBlock`) that can be used for placing the assertions on failure behavior, which is executed before the test. The placement of assertions in this block is superficial: it is merely a location to encode failure behavior that is separate from the main test code.

As an alternative to using the FILIBUSTER supplied setup and tear-down blocks, developers may also use either the JUnit supported `@BeforeAll`, `@BeforeEach`, `@AfterEach`, and `@AfterAll` blocks: fault injection will automatically be inhibited for any RPCs

executed inside of these blocks. *p*-SFIT also provides a specific block type that can be used in any test that inhibits fault injection for code placed within it that can be placed in any block. Similarly, *p*-SFIT also provides an explicit block type that can be used for bracketing different sets of RPCs together. This is done only for ease of debugging tests using the associated IDE plugin, as it has no impact on fault injection.

When any of these blocks are used, *p*-SFIT automatically brackets any RPCs in its IDE plugin to indicate which block any RPCs executed in those blocks were executed from.

11.3.1 Failure Specification API

In Table 11.1, the API that can be used in the failure block for encoding behavior of the service under failure is presented. In the implementation, this API is designed specifically for the testing of gRPC endpoints. Therefore it is assumed that all thrown exceptions are of type `StatusRuntimeException` and are instantiated by both gRPC code and description. For the assertions on an injected fault, such as `assertFaultThrows`, `assertOnFault` and `assertFaultHasNoImpact`, four variants exist: by method, by method and error code; by method and request; and by method, request and error code. For the stub invocation methods, `readOnlyRPC` and `sideEffectingRPC`, they are syntactic sugar on top of the existing `atLeast(0)` and `times(N)` APIs, respectively. This is done to force the developers to *think* about the implications of stating that an RPC can be invoked an arbitrary number of times.

Should multiple faults be injected simultaneously in a test iteration, FILIBUSTER provides the builder `CompositeFaultSpecification`, which test authors can use to state the combination of the failing RPC requests. Test authors can use that `CompositeFaultSpecification` object as a parameter in `assertFaultThrows` and `assertOnFault` to assert their application's behavior when multiple RPCs simultaneously fail.

11.3.2 Fault Matching API

Developers should also be able to encode faults at whatever granularity they choose. For example, when injecting a fault on RPC 1 with status code `UNAVAILABLE`, the developer might use `assertFaultThrows` with arguments `(RPC1, UNAVAILABLE, NOT_FOUND)` to indicate that the service throws an exception with status code `NOT_FOUND` when RPC 1 throws an exception with status code `UNAVAILABLE`. The developer could alternatively encode `assertFaultThrows` with arguments `(RPC1, NOT_FOUND)` to indicate that the service throws an exception with code `NOT_FOUND` when RPC 1 throws an

API	**Description** (Usage Block)
`stubFor`	Stub downstream dependency with response (`stubBlock`)
`verityThat`	Check that stubbed dependency is invoked `N` times (`assertStubBlock`)
`assertOnException`	Assertions for thrown exception on hard dependency (`failureBlock`)
`assertOnFault`	Assertions on RPC failure for hard & soft dependency (`failureBlock`)
`assertFaultThrows`	RPC failure of hard dependency results in exception (`failureBlock`)
`assertFaultPropagates`	RPC failure of hard dependency as unhandled (`failureBlock`)
`assertFaultHasNoImpact`	RPC failure of soft dependency has no impact (`failureBlock`)
`readOnlyRPC`	Mark RPC to soft dependency as read-only May be called 0 or more times (`assertOnException`)
`sideEffectingRPC`	Mark RPC to soft dependency as side-effecting Needs explicit invocation counts (`assertOnException`)

Table 11.1: Table of *p*-SFIT-enabled Filibuster assertion API and usage blocks for each API method.

exception with *any* status code. Either is correct, but their knowledge of the service may influence the developer's choice.

When Filibuster generates a subsequent test that injects a fault on RPC 1 with status code `DEADLINE_EXCEEDED`, the service throws the same exception with the same status code as a result of that fault injection. At this point, the developer in the former case may either write an additional assertion for the new failure *or* refine their test to indicate that any status code causes the exception to be thrown, as first done in the latter case.

To facilitate this matching approach, a method was necessary to characterize each system fault. This method would enable SFIT to generate sortable projections of each fault, organized in descending order based on the specificity of their fault

representation. This ensured that SFIT could match the fault behavior specified by the user based on their intention. Inspired by the distributed execution indexing algorithm, each fault is represented by the RPC's signature where the fault was injected, the RPC's request arguments, and the description of the fault injected for that RPC. When matching, SFIT first look for a matching use of one of the API functions by signature, arguments, and injected fault. When no match is found, SFIT then decreases the precision, first to the signature and arguments, then to the signature and code, and then to the signature alone.

This seems to match the developer's intent with the API:

1. First, the most precise match should be searched: the signature, arguments, and code.
2. Next, any encodings that use an explicit request should be matched, as they indicate multiple RPCs to the same method exist in the same execution, and failures of each may be treated differently at different call sites.
3. Third, the signature and status code, as all failures of the same type are most likely treated uniformly for a single request.
4. Finally, the developer may have just treated all failures for an RPC method uniformly.

11.3.3 IDE Plugin

To facilitate application developers in the diagnostic assessment of failed tests and encoding failure behavior, FILIBUSTER is provided with an IntelliJ IDE plugin. This plugin visualizes each test run generated and executed by *p*-SFIT, every RPC to a downstream dependency within a specific test along with its arguments and response, and each injected fault in their respective invocation sequences.

Figure 11.2 shows a screenshot of the plugin. For each of the 4 RPCs issued in the reference execution, FILIBUSTER injects `StatusRuntimeException` with the status code `DEADLINE_EXCEEDED` and `UNAVAILABLE`. This results in 8 executions with injected faults (4 RPCs * 2 faults). In Figure 11.3 an iteration where `StatusRuntimeException` with the code `DEADLINE_EXCEEDED` was injected in the RPC to (`UpdateCart`) is shown.

The plugin provides recommendations to the developer based on the test failures that FILIBUSTER identifies. These recommendations guide the developer: for example, if a test fails because a fault was injected and the developer did not encode the behavior under fault. For every exception that `FilibusterGrpcTest` throws, FILIBUSTER has a defined diagnostic message that the plugin shows. That message describes

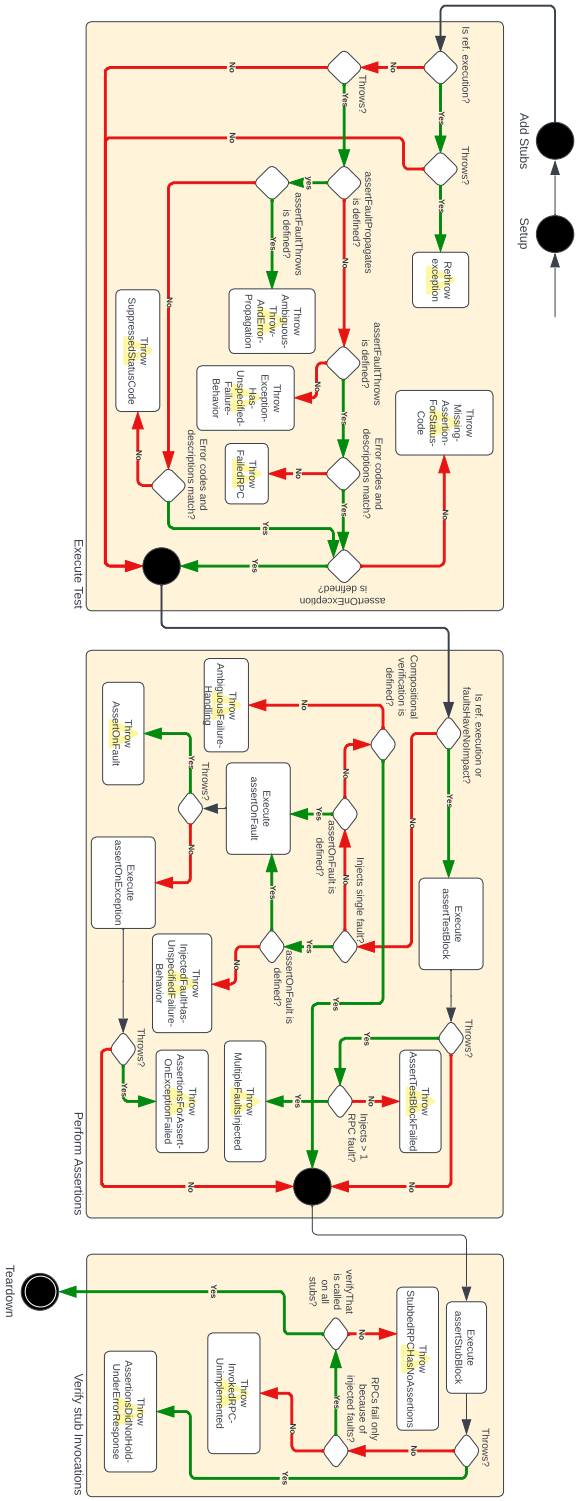

Figure 11.1: Testing procedure for p-SFIT.

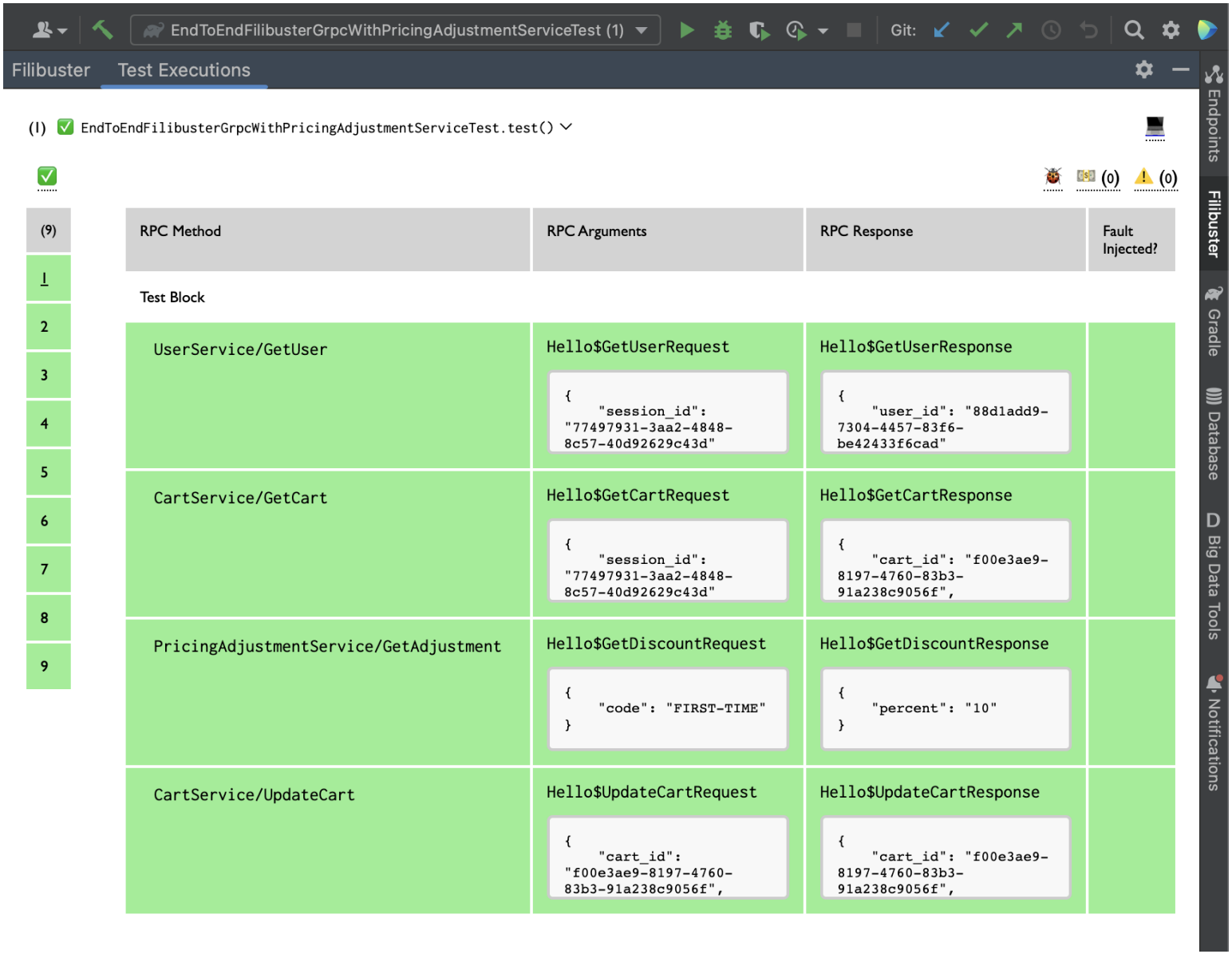

Figure 11.2: FILIBUSTER IntelliJ plugin visualizes the intercepted RPC invocations and their arguments and responses.

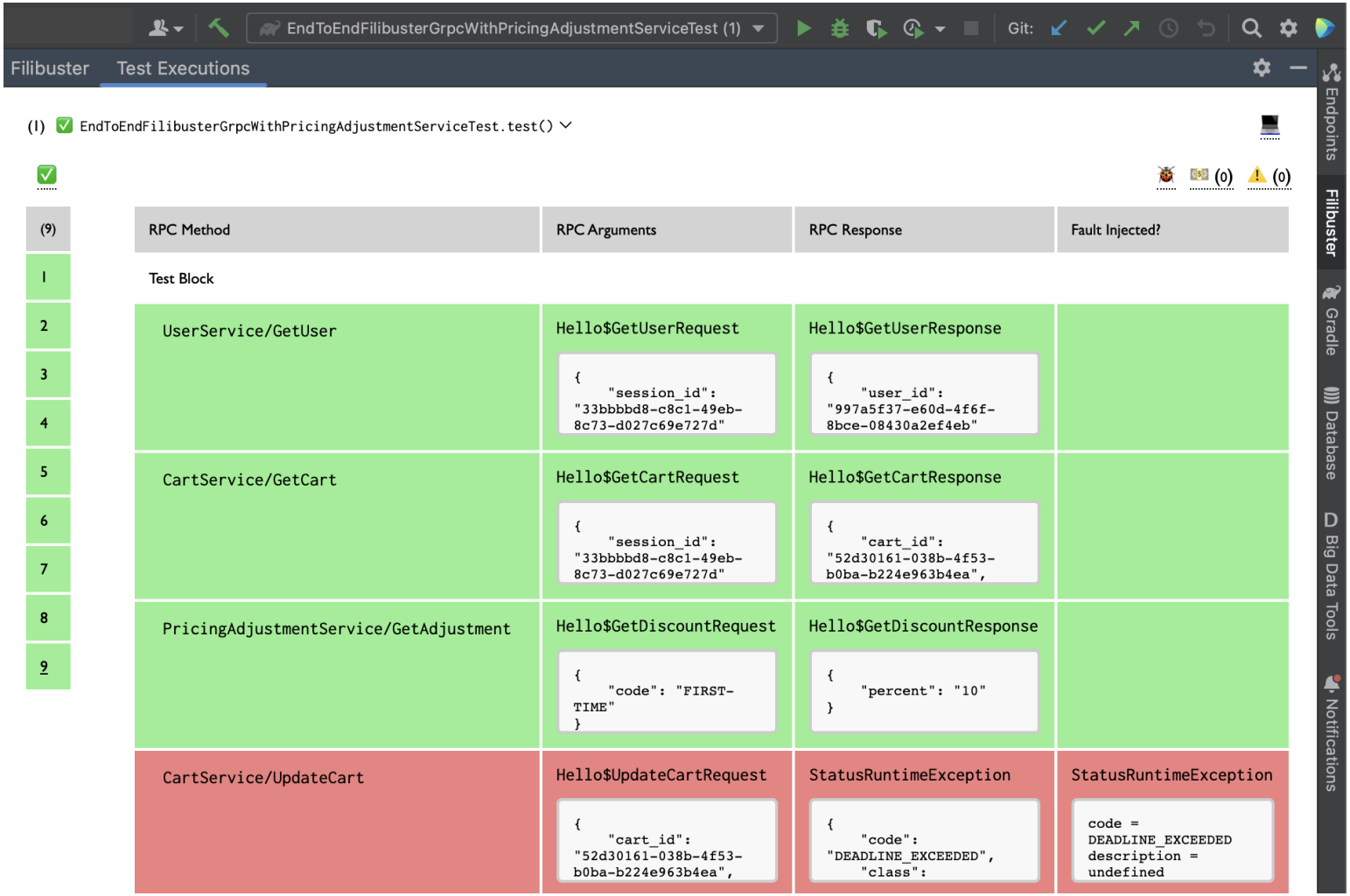

Figure 11.3: FILIBUSTER iteration where a `StatusRuntimeException` is injected in the RPC invocation to `UpdateCart`.

why the exception was thrown and provides links to the precise API methods that developers need to use to resolve the issue. That includes errors for hard dependencies, assertion failures for soft dependencies, and missing stub invocations for under-specified dependencies. The plugin also provides an overview of the API coverage of the service under test, prompting developers to increase coverage where necessary.

Anecdotally, it was identified that in the process of writing tests using this design ourselves, having a visual depiction of the failure with recommendations on how to fix was extremely valuable. An error and recommendations generated by the FILIBUSTER plugin is shown in Figure 11.4 and Figure 11.5.

For the implementation, the IntelliJ IDE plugin is a web view that renders information that is written to the file system, by the FILIBUSTER implementation. Therefore, no actual logic lives in the plugin code itself. This was done to enable the development of plugins for other IDE's, such as Visual Studio Code.

11.3.4 *p*-SFIT Testing Procedure

The *p*-SFIT testing procedure is depicted in Figure 11.1.

1. `setupBlock`
   Test authors should put code for the setup of the test in this block. The use of this block inhibits fault injection for any RPCs issued in it. For example, if using the service-under-test's API to stage state for running a test, any downstream dependencies that are invoked as part of the execution of the setup block will not be tested for faults. Thereby, this ensures that faults do not prevent proper staging before the execution of the actual test.

2. `stubBlock`
   This block contains code for stubbing any downstream dependencies that need to be stubbed for the test to pass. Stubbing of these dependencies should only be performed using the FILIBUSTER-provided `stubFor` method as FILIBUSTER needs to interpose on these mocks for fault injection testing.

3. `executeTestBlock`
   The test execution code should be placed in this block. Any downstream RPCs that are invoked as a result of a method invocation inside of this block will be subject to fault injection. Test authors should store any responses that are required for assertions, placed in the `assertTestBlock`, in instance variables. FILIBUSTER executes the test code written in this block. If the test is successful and no exceptions are thrown, FILIBUSTER proceeds to the next

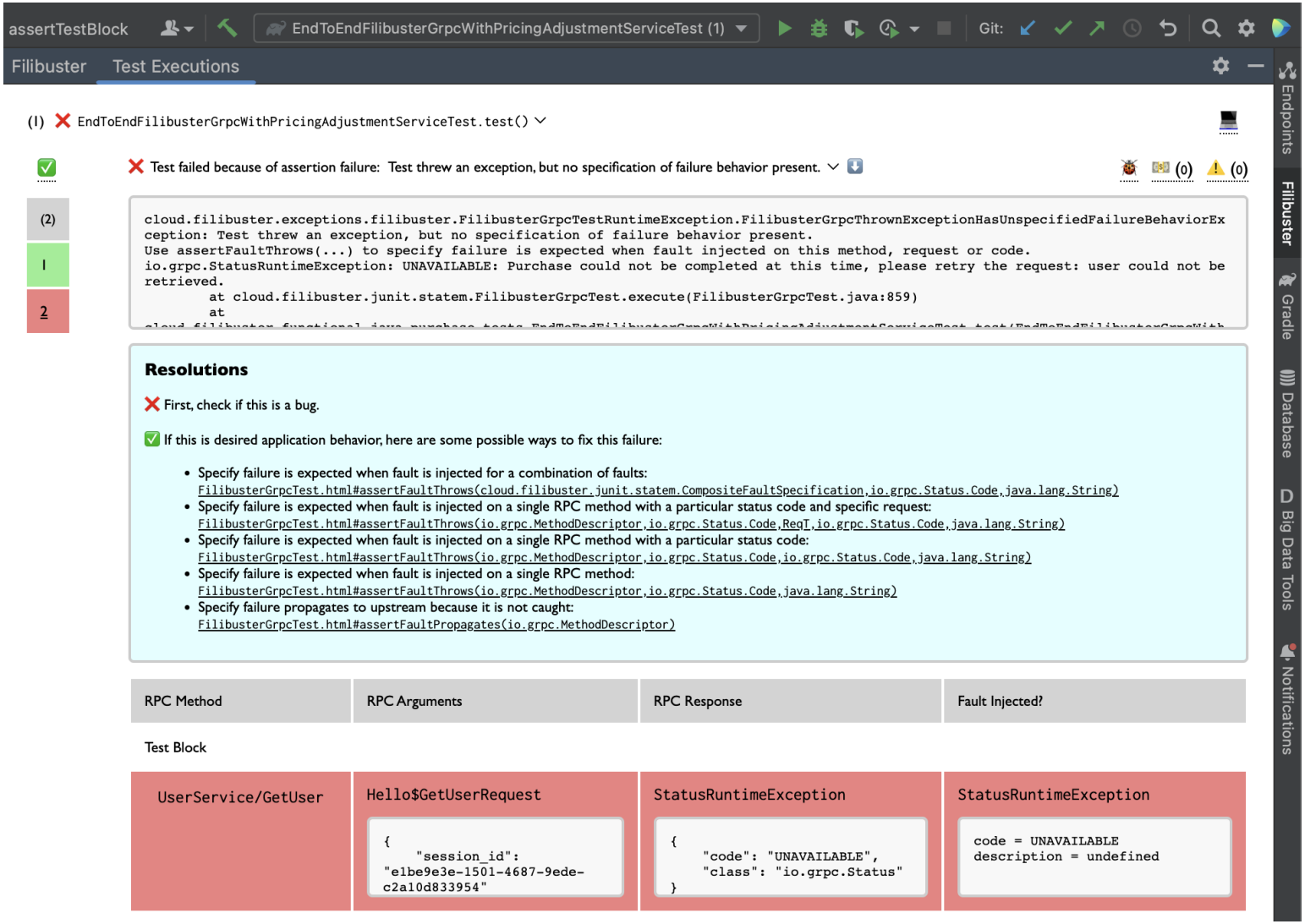

Figure 11.4: Iteration, where FILIBUSTER injected a `StatusRuntimeException` with the code `UNAVAILABLE` in `GetUser`. The test fails since the application behavior is not defined for this fault. The FILIBUSTER plugin shows recommendations regarding API methods that can be used to define the application behavior.

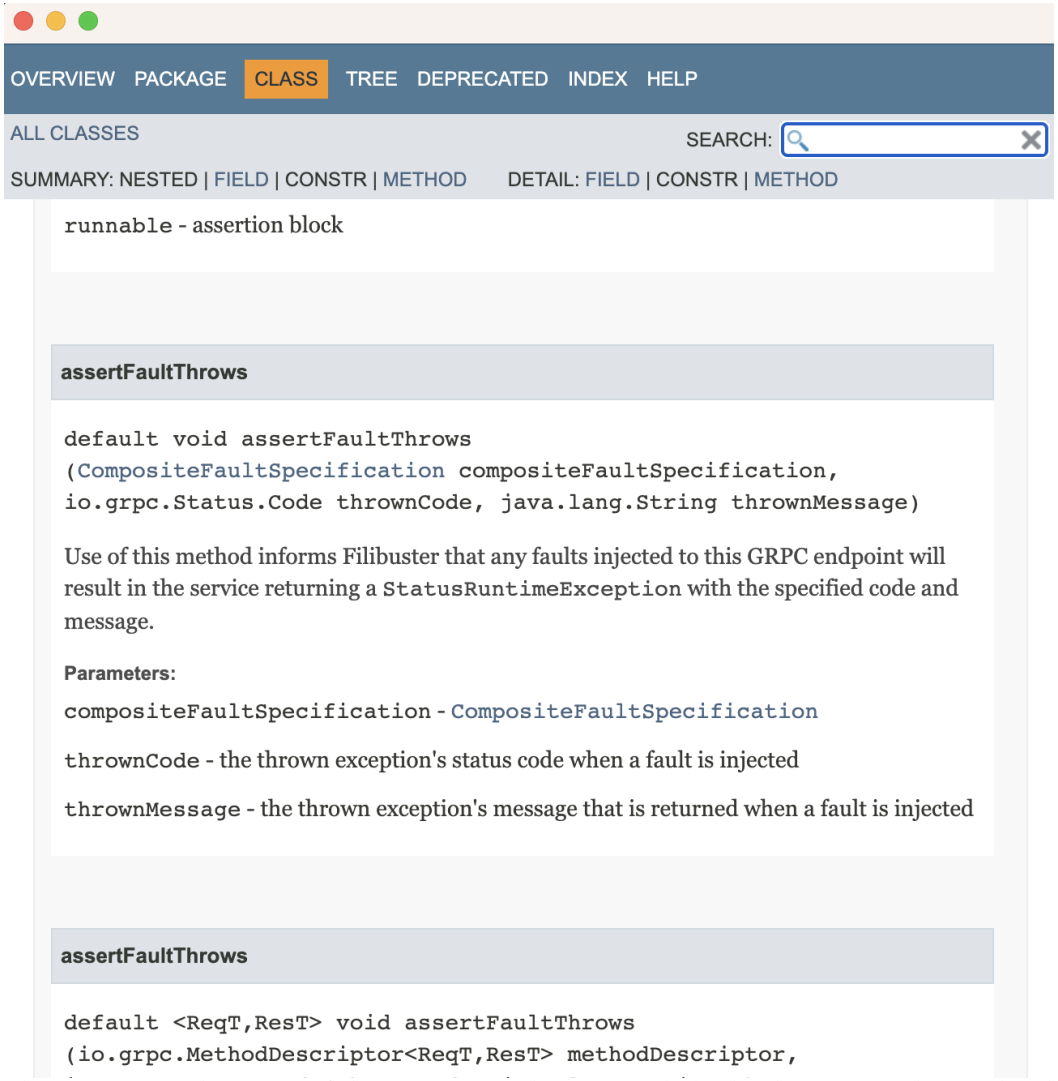

Figure 11.5: Popup that appears when one of the recommendations is clicked. The popup shows the documentation page of the respective method in Filibuster's API.

block. Otherwise, should the test throw an exception because of a failing RPC invocation (referred to here as R_f), FILIBUSTER distinguishes between two cases:

a) If the current execution is the initial reference execution where no faults are injected, FILIBUSTER rethrows the exception and fail the test iteration. FILIBUSTER shows the exception details in the IDE plugin.

b) Otherwise if a fault was injected in the current iteration, FILIBUSTER begins by verifying that the authors have properly specified the failure behavior of the application under fault in the test for a *hard* dependency. This process is described below in Section 11.3.4.1.

4. `assertTestBlock`
   Test authors should place test assertions here. The use of this block inhibits fault injection, which is necessary when using the very API under test to perform assertions. For example, if a test involves querying a user database to initiate a subscription and subsequently querying again to confirm the subscription, it is essential to introduce a fault solely during the subscription phase's user query and not during the verification phase.

   In this block, FILIBUSTER checks whether the current execution is the initial reference execution or whether the faults injected in the current execution have no impact. If either is the case, FILIBUSTER executes the assertions defined by the test author in the `assertTestBlock`. If they are successful, FILIBUSTER proceeds to the next block. Otherwise, if they fail, FILIBUSTER checks how many faults were injected.

   a) If more than 1 fault was injected, FILIBUSTER throws `FilibusterGrpc_」MultipleFaultsInjectedException`.

   b) Otherwise, FILIBUSTER throws `FilibusterGrpcAssertTestBlockFailed_」Exception`.

   Otherwise, if the current execution is not the reference execution and the injected faults have an impact, FILIBUSTER executes the subprocedure associated with *soft* dependency failure. This process is described below in Section 11.3.4.2.

5. `assertStubBlock`
   Use this block for performing assertions on stub invocations. This should be done using the FILIBUSTER-provided `verifyThat` method, as FILIBUSTER must interpose on these calls to automatically adjust the expected invocation count when faults are injected.

a) First, the assertions in the stub block are executed. Should they fail, `FilibusterGrpcAssertionsDidNotHoldUnder_ErrorResponseException` is thrown.

b) Second, FILIBUSTER checks whether all RPCs that failed had injected faults. If that is not the case, this indicates that some invoked RPCs were left unimplemented. Thus, FILIBUSTER throws `FilibusterGrpcInvokedRPC_UnimplementedException`.

c) Third, FILIBUSTER checks whether `verifyThat` was called on all stubbed methods. If that is not the case, FILIBUSTER fails the test and throws `FilibusterGrpcStubbedRPCHasNoAssertions_Exception`. Otherwise if successful, FILIBUSTER proceeds to the next block.

6. `teardownBlock`
   Test authors should use this block for performing test tear-down. Similar to the `setupBlock`, fault injection is inhibited for any downstream dependencies that are invoked as part of any method in this block.

11.3.4.1 Hard Dependency Subprocedure

This subprocedure is executed upon hard dependency failure, due to a fault injection, in the `executeTestBlock` block, when the test throws an unhandled exception received by the application.

1. First, FILIBUSTER starts by checking whether the test authors have indicated that the failure of R_f should propagate upstream (using `assertFaultPropagates`).

   a) If that is the case, FILIBUSTER perform a sanity check whether the test authors have simultaneously defined an exception-throwing behavior for R_f (using `assertFaultThrows`). Defining both `assertFaultThrows` and `assertFaultPropagates` for R_f indicates ambiguous failure behavior. If true, FILIBUSTER throws `FilibusterGrpcAmbiguousThrowAndErrorPropagationException`.

2. If the sanity check is successful and only `assertFaultPropagates` is defined for R_f, FILIBUSTER proceeds to check whether the status code and description of the propagated fault match the code and description of the injected fault.

   a) If they do not match, FILIBUSTER throws `FilibusterGrpcSuppressedStatus_CodeException`.

b) If the status code and description match, FILIBUSTER proceeds to check whether test authors have defined assertions for the exception-throwing behaviour of R_f (using `assertOnException`). If that is the case, FILIBUSTER proceeds to the next block. Otherwise, FILIBUSTER throws `Filibuster_GrpcMissingAssertionForStatusCodeException`.

3. If test authors have not indicated that failure of R_f propagates upstream, FILIBUSTER checks if they instead defined that failure of R_f results in the service returning an exception (using `assertFaultThrows`).

   a) In case `assertFaultThrows` was not defined, this indicates unspecified failure behavior for R_f and FILIBUSTER throws `FilibusterGrpcThrown_ExceptionHasUnspecifiedFailureBehaviorException`.

   b) If that is not the case, FILIBUSTER throws `FilibusterGrpcFailedRPCException`.

4. Otherwise, FILIBUSTER checks whether test authors have defined assertions for the exception-throwing behaviour of R_f (using `assertOnException`).

   a) If that is the case, FILIBUSTER proceeds to the next block.

   b) Otherwise, FILIBUSTER throws `FilibusterGrpcMissingAssertionForStatus_CodeException`.

11.3.4.2 Soft Dependency Subprocedure

This subprocedure is executed upon soft dependency failure, due to a fault injection, in the `assertTestBlock` block.

1. First, FILIBUSTER proceeds to check whether test authors have defined alternative assertions that apply when a fault is injected (using `assertOnFault`). If that is the case, FILIBUSTER executes these assertions.

   a) Should they fail, FILIBUSTER throws `FilibusterGrpcAssertOnFaultException`.

   b) Otherwise, if they are successful, FILIBUSTER executes the assertions defined in `assertOnException`. If the assertions fail, FILIBUSTER throws `FilibusterGrpcAssertionsForAssertOn_ExceptionFailedException`. Otherwise if successful, FILIBUSTER proceeds to the next block.

2. If test authors have not defined alternative assertions in `assertOnFault`, FILIBUSTER differentiates between two cases:

a) If Filibuster is injecting a single fault, the lack of `assertOnFault` indicates there is no specification of failure behavior. Thus, Filibuster throws `FilibusterGrpcInjectedFaultHasUnspecifiedFailureBehavior_Exception`.

b) If more than one fault is being injected, Filibuster checks if compositional reasoning potentially defines the fault behavior. If that is the case, Filibuster proceeds to the next block. Otherwise, Filibuster throws `FilibusterGrpcAmbiguousFailureHandlingException`.

11.4 Takeaways

Principled Service-level Fault Injection Testing (*p*-SFIT) improves on SFIT by providing a structure, developer-centric methodology for resilience testing.

More specifically, *p*-SFIT is a software development process that extends the SFIT approach for applications that are designed to gracefully degrade under failure, as observed in our industrial evaluation with Foodly. It improves the process of both *writing structured happy path functional tests* and then *updating those tests to also account for the application's behavior under failure*. Through the application of SFIT to an industrial application, requirements were derived based on observations from their test code, which led to the redesign of SFIT as *p*-SFIT, which has led to latent bugs found by engineers at Foodly.

In the next chapter, a tutorial on using *p*-SFIT is presented, demonstrating how the *p*-SFIT can help developers update tests to encode failure behavior in their functional tests based on fault injection.

Chapter 12

Using *p*-SFIT: A Tutorial

"Faced with mysteries dark and vast, statements just seem vain at last, some rise, some fall, some climb, to get to Terrapin."

Garcia & Hunter, *"Terrapin Station"*

In this chapter, the open-source prototype implementation of *p*-SFIT, Filibuster, is used to demonstrate the use of *p*-SFIT on the example application presented in Section 10 to capture its behavior under fault. For a review of this example application, the reader is referred to both Section **??** and Figure(s) 10.1 and 10.2.

First, a functional test of the application is written using the *p*-SFIT testing interface provided by Filibuster. This demonstrates how *p*-SFIT explicitly assists the developer in writing a well-formed happy path test of the application before the start of fault injection testing. Next, the authored functional test is used to perform fault injection testing of the application. From here, a developer must specify using *p*-SFIT *how the application's behavior changes under fault.*

Finally, the application's behavior is altered, and the *p*-SFIT testing process is re-employed to demonstrate how it forces that the original assertions be modified to account for the new behavior; the existing assertions are adjusted, and a specification of new application behavior under fault is added to the functional test.

12.1 Single Adjustment Example

Consider the case where only a single possible pricing adjustment might be performed by the example application presented in Chapter **??**.

12.1.1 `setupBlock`: Perform Test Setup

First, the `setupBlock` is used to set up the test state before each test execution. In this example, the cache and database states are reset and then primed by giving the customer some starting funds to use for their purchase.

This block functions similarly to the `@BeforeAll` and `@BeforeEach` blocks and can be combined. However, `setupBlock` allows subclasses to override test setup while inheriting any of the other blocks: not possible with `@BeforeAll` or `@BeforeEach`.

```
1  @Override
2  public void setupBlock() {
3      // Reset cache state.
4      PurchaseWorkflow.resetCacheObjectForUser(consumerId);
5
6      // Reset database state.
7      PurchaseWorkflow.depositFundsToAccount(consumerId, 20000);
8      assertEquals(
9          20000,
10             PurchaseWorkflow.getAccountBalance(consumerId));
11 }
```

12.1.2 `stubBlock`: Stub Downstream Dependencies

In the `stubBlock`, the Filibuster-provided `stubFor` method is used, which wraps the existing `stubFor` method provided by GrpcMock to be able to adjust stubs based on fault injections, to stub the responses of downstream dependencies.

Here, every downstream dependency invoked by this test with responses is stubbed. If one of these stubs is missed, Filibuster will provide a specific warning asking the stub to be written before proceeding.

Stubbing can be performed at a method level (as done with the `getUser`, `getCart`, and `updateCart` RPCs) or for an individual RPC with specific arguments, as done with the `getAdjustment` RPC.

```
1  @Override
2  public void stubBlock() {
3      stubFor(
4          UserServiceGrpc.getGetUserMethod(),
5          Hello.GetUserRequest.
6              newBuilder().
7                  setSessionId(sessionId).build(),
```

```
            Hello.GetUserResponse.
                newBuilder().
                    setUserId(consumerId).build());

        stubFor(
            CartServiceGrpc.getGetCartMethod(),
            Hello.GetCartRequest.newBuilder()
                .setSessionId(sessionId).build(),
            Hello.GetCartResponse.newBuilder()
                .setCartId(cartId)
                .setTotal("10000")
                .setMerchantId(merchantId)
                .build());

        stubFor(
            PricingAdjustmentServiceGrpc
                .getGetAdjustmentMethod(),
            Hello.GetDiscountRequest
                .newBuilder()
                    .setCode("FIRST-TIME").build(),
            Hello.GetDiscountResponse
                .newBuilder()
                    .setPercent("10").build());

        stubFor(
            CartServiceGrpc.getUpdateCartMethod(),
            Hello.UpdateCartRequest.newBuilder()
                .setCartId(cartId.toString())
                .setDiscountAmount("10")
                .build(),
            Hello.UpdateCartResponse.newBuilder()
                .setCartId(cartId.toString())
                .setTotal("9000")
                .build());
    }
```

12.1.3 executeTestBlock: Write the Functional Test

The test code is placed in the `executeTestBlock`. In this application, there is only a call to the API gateway. The response is then set to the `response` variable, which is provided by Filibuster and automatically fed into the assertion methods.

```
@Override
public void executeTestBlock() {
    APIServiceGrpc.APIServiceBlockingStub blockingStub =
        APIServiceGrpc.newBlockingStub(API_CHANNEL);
    Hello.PurchaseRequest request =
        Hello.PurchaseRequest.newBuilder()
            .setSessionId(sessionId)
            .build();
    response.set(blockingStub.makePurchase(request));
}
```

12.1.4 assertTestBlock: Perform Test Assertions

In this block, assertions on the response of the RPC method invoked in the test block are performed. As shown, it is asserted that the response is successful and that the database writes are correct.

In this example, a helper method is parameterized so that the correct total can be asserted in our failure case when the discount is not applied. Most likely, this refactoring would be performed when adjusting the functional test to account for the application's failure behavior; this presentation is for brevity.

```
@Override
public void assertTestBlock() {
    assertTestBlock(10000);
}

public void assertTestBlock(int total) {
    Hello.PurchaseResponse response =
        (Hello.PurchaseResponse) getResponse();

    // Verify response.
    assertNotNull(response);
    assertTrue(response.getSuccess());
    assertEquals(total, response.getTotal());
```

```
14
15      // Verify cache writes.
16      JSONObject cacheObject = PurchaseWorkflow
17          .getCacheObjectForUser(consumerId);
18      assertTrue(
19          generateExpectedCacheObject(
20              consumerId, cartId, total).similar(cacheObject));
21
22      // Verify database writes.
23      assertEquals(20000 - total,
24                   PurchaseWorkflow.getAccountBalance(consumerId));
25      assertEquals(total,
26                   PurchaseWorkflow.getAccountBalance(merchantId));
27  }
```

12.1.5 `assertStubBlock`: Verify Stub Invocations

In this block, assertions on the invocation counts of the stubbed RPC methods are written. As shown, the method `verifyThat` from Filibuster's API is used as a wrapper around the GrpcMock method of the same name to assert that the methods `getUser`, `getCart`, `getAdjustment`, and `updateCart` is invoked precisely once.

```
1   @Override
2   public void assertStubBlock() {
3       verifyThat(UserServiceGrpc.getGetUserMethod(), 1);
4       verifyThat(CartServiceGrpc.getGetCartMethod(), 1);
5
6       Hello.GetDiscountRequest request =
7           Hello.GetDiscountRequest.newBuilder()
8               .setCode("FIRST-TIME")
9               .build();
10
11      verifyThat(
12          PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
13          request,
14          1);
15
16      verifyThat(CartServiceGrpc.getUpdateCartMethod(), 1);
17  }
```

12.1.6 `teardownBlock`: Perform Test Teardown

Similar to `setupBlock`, the `teardownBlock` can be used for performing test teardown. Again, it doesn't replace use `@AfterEach` or `@AfterAll`, but operates in a similar way: they can be combined or overridden explicitly in sub-classes to change teardown while inheriting other behavior.

Here, the state of the database is reset in the teardown to be a proper test citizen.

```
1  @Override
2  public void teardownBlock() {
3      // Reset cache state.
4      PurchaseWorkflow.resetCacheObjectForUser(consumerId);
5
6      // Reset database state.
7      PurchaseWorkflow.deleteAccount(consumerId);
8      PurchaseWorkflow.deleteAccount(merchantId);
9  }
```

12.1.7 `failureBlock`: Application Failure Behavior

The `failureBlock` is executed at the start of the test, after setup and stubbing, and provides a convenient location for placing the assertions on the application's behavior under fault.

Here, the entirety of the complted `failureBlock` is presented. However, as part of the *p*-SFIT process, each injected fault would induce a test failure: from there, the developer would write a single assertion in this section to adapt the test's assertions to the application's behavior under fault. This would then trigger re-execution of the test where all assertions would be re-checked before injection of the next fault, which would require another addition to this section if that failure provoked a test failure. This process was described in Section 11.1.

First, it is stated that both the `getUser` and `getCart` methods are *hard dependencies*: they will throw an exception with a particular message on failure and abort the purchasing process. In both cases, they result in a thrown exception with GRPC status code `UNAVAILABLE`. This is a result of the first two fault injections: first, a fault injected on the `getUser` method; second, a fault injected on the `getCart` method.

Next, to ensure the state of the system is consistent whenever a GRPC exception with status code `UNAVAILABLE` is thrown, the state of the system is described. This avoids situations where hard dependencies fail and result in thrown exceptions, but the state of the system is different in both cases. Here, it is stated precisely that no transfer of funds should have occurred, and that the `updateCart` method should

not have been invoked. It is also stated that invocation of other downstream RPCs are OK because they are read-only: `getCart` and `getAdjustment`. This statement is required for both the first and second fault injections, as it must be written to proceed to the second fault injection scenario.

Next, it is stated that if the RPC to `getAdjustment` fails, the RPC will not be invoked, with side effects, `udpateCart` and that the customer will pay total price. Finally, it is stated that if the `updateCart` RPC fails, the customer will pay total price. These are examples of *soft dependencies.*

```
@Override
public void failureBlock() {
    // Any failure of the getUser call results in
    // upstream receiving UNAVAILABLE exception.
    assertFaultThrows(
        UserServiceGrpc.getGetUserMethod(),
        Status.Code.UNAVAILABLE,
        "Purchase could not be completed at this time,
            please retry the request:
            cart could not be retrieved."
    );

    // Any failure of the getCart call results in
    // upstream receiving UNAVAILABLE exception.
    assertFaultThrows(
        CartServiceGrpc.getGetCartMethod(),
        Status.Code.UNAVAILABLE,
        "Purchase could not be completed at this time,
            please retry the request:
            cart could not be retrieved."
    );

    // State what the state of the system on UNAVAILABLE.
    assertOnException(Status.Code.UNAVAILABLE, () -> {
        // Verify transaction did not occur.
        assertEquals(
            20000,
            PurchaseWorkflowWithPricingAdjustmentService
                .getAccountBalance(consumerId));
        assertEquals(
            0,
```

```
32              PurchaseWorkflowWithPricingAdjustmentService
33                  .getAccountBalance(merchantId));
34
35          // Notify the system some endpoints are read-only
36          // and therefore OK to skip
37          // when one returns a failure.
38          readOnlyRpc(CartServiceGrpc.getGetCartMethod());
39          readOnlyRpc(
40              PricingAdjustmentServiceGrpc
41                  .getGetAdjustmentMethod());
42
43          // Ensure one did not update the cart.
44          sideEffectingRpc(
45              CartServiceGrpc.getUpdateCartMethod(), 0);
46      });
47
48      // Failure of the getAdjustment call results
49      // in no discount (and no call to updateCart.)
50      assertOnFault(
51          PricingAdjustmentServiceGrpc
52              .getGetAdjustmentMethod(),
53          Hello.GetDiscountRequest.newBuilder()
54                  .setCode("FIRST-TIME")
55                  .build(),
56          () -> {
57              assertTestBlock(10000);
58              sideEffectingRpc(CartServiceGrpc
59                  .getUpdateCartMethod(), 0);
60          }
61      );
62
63      // Failure of the updateCart call results
64      // in no discount.
65      assertOnFault(
66          CartServiceGrpc.getUpdateCartMethod(),
67          () -> { assertTestBlock(10000); }
68      );
69  }
```

12.2 Multiple Adjustment Example

To demonstrate how Filibuster supports encoding the application's behavior under fault when multiple faults are injected at the same time, a few small modifications to our application. Instead of only seeing if the customer is eligible for a single pricing adjustment, the application should attempt to look up discounts using multiple discount codes, and depending on which RPCs succeed, apply the greatest discount value. Therefore, the amount the customer is charged is now dependent on which RPCs succeed.

12.2.1 Updating the Happy Path Test

First, it is necessary to add the new stubs for the additional calls to look up different discount codes in the `stubBlock`.

```
1  stubFor(PricingAdjustmentServiceGrpc
2              .getGetAdjustmentMethod(),
3          Hello.GetDiscountRequest.newBuilder()
4              .setCode("FIRST-TIME")
5              .build(),
6          Hello.GetDiscountResponse.newBuilder()
7              .setPercent("10")
8              .build());
9
10 stubFor(PricingAdjustmentServiceGrpc
11             .getGetAdjustmentMethod(),
12         Hello.GetDiscountRequest.newBuilder()
13             .setCode("RETURNING")
14             .build(),
15         Hello.GetDiscountResponse.newBuilder()
16             .setPercent("5")
17             .build());
18
19 stubFor(PricingAdjustmentServiceGrpc
20             .getGetAdjustmentMethod(),
21         Hello.GetDiscountRequest.newBuilder()
22             .setCode("DAILY")
23             .build(),
24         Hello.GetDiscountResponse.newBuilder()
25             .setPercent("1")
```

```
            .build());

stubFor(CartServiceGrpc
            .getUpdateCartMethod(),
        Hello.UpdateCartRequest.newBuilder()
            .setCartId(cartId.toString())
            .setDiscountAmount("10").build(),
        Hello.UpdateCartResponse.newBuilder()
            .setCartId(cartId.toString())
            .setTotal("9000").build());

stubFor(CartServiceGrpc.getUpdateCartMethod(),
        Hello.UpdateCartRequest.newBuilder()
            .setCartId(cartId.toString())
            .setDiscountAmount("5").build(),
        Hello.UpdateCartResponse.newBuilder()
            .setCartId(cartId.toString())
            .setTotal("9500").build());

stubFor(CartServiceGrpc.getUpdateCartMethod(),
        Hello.UpdateCartRequest.newBuilder()
            .setCartId(cartId.toString())
            .setDiscountAmount("1").build(),
        Hello.UpdateCartResponse.newBuilder()
            .setCartId(cartId.toString())
            .setTotal("9900").build());
```

Next, the expected invocation counts for each stub in the `assertStubBlock` must be updated, otherwise FILIBUSTER will not proceed past the initial, reference execution.

```
for (Map.Entry<String, String> discountCode :
        PurchaseWorkflow.getDiscountCodes()) {
    Hello.GetDiscountRequest request =
        Hello.GetDiscountRequest.newBuilder()
            .setCode(discountCode.getKey()).build();

    verifyThat(
        PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
```

```
9               request, 1);
10  }
```

12.2.2 Updating the Application's Failure Behavior

In the `failureBlock`, it is necessary to start considering how these faults can impact the application in *isolation*. Here, it is necessary to state that if the 10% discount is not applied, the 5% will be; if either the 5% or 1% discount fails, the 10% discount will be applied.

```
1   // If they don't get 10% discount they will get 5%.
2   assertOnFault(
3           PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
4           Hello.GetDiscountRequest.newBuilder()
5                   .setCode("FIRST-TIME")
6                   .build(),
7           () -> { assertTestBlock(9500); }
8   );
9
10  // If either the 5% or 1% fail or succeed,
11  // the 10% will be applied.
12  assertOnFault(
13          PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
14          this::assertTestBlock
15  );
```

Next, it is necessary to start thinking about *compositions* of faults. For example, what happens if both the 10% and 5% requests fail? Then, the user will get a 1% discount.

```
1   // What happens if the 10% and 5% fail together? 1%.
2   CompositeFaultSpecification firstTwoDiscounts =
3       new CompositeFaultSpecification.Builder()
4           .faultOnRequest(
5               PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
6               Hello.GetDiscountRequest.newBuilder()
7                   .setCode("FIRST-TIME").build())
8           .faultOnRequest(
9               PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
```

```
            Hello.GetDiscountRequest.newBuilder()
                .setCode("RETURNING").build())
        .build();
assertOnFault(firstTwoDiscounts, () -> { assertTestBlock(9900); });
```

This must then be repeated for each composition of failures for all possible discount codes. Even the combination where all three fail. In each of these cases, the test cannot determine the code's outcome when the failures occur.

```
// What happens if all three fail?
CompositeFaultSpecification allDiscounts =
    new CompositeFaultSpecification.Builder()
        .faultOnRequest(
            PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
            Hello.GetDiscountRequest.newBuilder()
                .setCode("FIRST-TIME").build())
        .faultOnRequest(
            PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
            Hello.GetDiscountRequest.newBuilder()
                .setCode("RETURNING").build())
        .faultOnRequest(
            PricingAdjustmentServiceGrpc.getGetAdjustmentMethod(),
            Hello.GetDiscountRequest.newBuilder()
                .setCode("DAILY").build())
        .build();
assertOnFault(allDiscounts, () -> { assertTestBlock(10000); });
```

From here, the developer will have to consider what happens when the first two discount code `getAdjustment` RPCs fail *in combination* with the `updateCart` call, where no discount will be applied because the cart cannot be updated. This is not included here for brevity.

✓ **As demonstrated, Filibuster can surface and provoke the developer into thinking about failure scenarios they may not have imagined: the first two discount lookups fail, the user only gets a 1% discount, but because the cart could not be updated, they end up without a discount.**

12.3 Adding Another Soft Dependency

Let's assume that one may want to add one more piece of functionality to our application where an email is sent to the user after purchase. This will be done with a new RPC on the cart service, `notifyOrder`.

As one does not want to abort the entire purchasing process if they cannot send the email, this is another example of a *soft dependency* that the user will not immediately see the effect of (or not even be aware that it should have happened.)

First, it is necessary to update both the `stubBlock` and `assertStubBlocks` to stub and verify the invocation of the stub. Next, it is necessary to update the `failureBlock` to indicate that when this RPC fails, it did not affect the outcome of the test (other than the stub not being invoked.)

```
1  assertFaultHasNoImpact(CartServiceGrpc.getNotifyOrderMethod());
```

Finally, is it required to write assertions to capture the failure of the `notifyOrder` RPC that sends the email with all possible combinations of failures of the `getAdjustment` RPC calls?

> ✓ **No, compositional reasoning will infer automatically, and check with test execution, that executions where some combination of `getAdjustment` RPCs fail with `notifyOrder` RPC act the same way as invocations where the email can be sent.**

12.4 Takeaways

As demonstrated, *p*-SFIT's structured test interface not only assists application developers in writing better happy path functional tests for their microservice applications but also enforces that they author those tests in such a way that can be used to identify bugs related to a microservice application's failure handling code.

Instead of relying on the developer to understand what the possible impacts of a fault are, *p*-SFIT prompts them when faults are injected to ensure that the effects of that fault are explicitly encoded, thereby avoiding situations where developers ignore the impact of failures that seemingly do not affect their application.

Whenever possible, this structured way of writing tests also enables compositional reasoning, preventing developers from writing individual test cases and assertions for each failure scenario that combines already observed failures. Only when the system cannot determine the system's behavior or when an existing assumption does not hold is the developer required to specify the behavior manually.

Chapter 13

Conclusions

> *"The race doesn't always belong to the swift nor the battle to the strong. It belongs rather to those who run the race, who stay the course, and who fight the good fight."*
>
> Carl Yastrzemski

Fault injection testing is one of the predominant techniques used by developers and operators of industrial microservice applications for testing their applications for resilience to infrastructure and downstream service failures. It is mainly performed on a microservice application *in production* where customers may be affected with application resilience bugs detected using coarse-grained metrics. Rather unfortunately, the use of coarse-grained metrics for bug detection may hide customer-affecting issues if they are either infrequent or do not effect a large segment of the customer's of an application.

While this problem is not new to academic researchers, access to industrial systems has limited existing research on fault injection in microservice applications. When academic proposals do appear to address these issues, they are often mismatched with what the practitioners working on industrial applications are looking for or need.

This dissertation makes significant advancements over the existing work in microservice application fault injection testing in several areas. First, it is improves accesibility to realistic application by constructing an application corpus inspired by industrial applications. Then, it presents the design of a new technique for RPC identification in a microservice application and uses that technique to design a novel technique for microservice application fault injection. This technique is both evaluated on that corpus and in an industrial setting. Using the knowledge gained from that evaluation, the corpus is then extended with a more realistic industrial

application, and the technique redesigned to align more closely with the industrial practices of microservice application development.

To achieve this, several key social and technical challenges were overcome:

- A microservice application corpus was constructed to use as the basis for the research in this thesis. (Chapter 4)
- A method for identification of the RPCs executed by a microservice application was invented. This method, *Distributed Execution Indexing* (DEI), improves on the state-of-the-art in distributed tracing by encoding information consistent with changes in flow control, looping, and function abstraction. (Chapter 5)
- A developer-centric fault injection technique, *Service-level Fault Injection Testing* (SFIT), was designed to provide an exhaustive search with test case reduction optimizations that leverage the existing functional tests that microservice application developers are already writing. (Chapter 6) This technique leverages foundational work with DEI's and is evaluated using the created microservice corpus. (Chapter 7).
- This technique was then evaluated in an industrial context at Foodly (Chapter 8) to identify its deficiencies and whether it could identify bugs in an industrial microservice application. (Chapter 9) This evaluation extended the microservice application corpus with an example inspired by application behavior at Foodly.
- From there, a new fault injection technique and testing process, *Principled Service-level Fault Injection Testing* (*p*-SFIT), was derived based on industrial principles around microservice design and resilience (Chapter 10). *p*-SFIT improves on many of the deficiencies in SFIT by ensuring that developers write tests that adequately capture happy path behavior in their application and ensure those tests capture all possible behavior under fault. (Chapter 11)
- Finally, a tutorial on *p*-SFIT is presented to demonstrate how an application developer can use *p*-SFIT to identify an application resilience bug and encode the behavior of their application under failure into a functional test. (Chapter 11)

Impact. Since the development and evaluation of both SFIT and *p*-SFIT, Foodly has asked application developers across all of their 500+ services to use Filibuster for testing their applications for resilience.

While the ultimate approach *p*-SFIT has not been fully adopted at Foodly yet for technical reasons, at least six (6) different teams within Foodly have implemented

FILIBUSTER tests in 2023 for their application using components taken from both SFIT and *p*-SFIT: the testing process (from SFIT) and the integrated IDE plugin for debugging failures (from *p*-SFIT.) More teams are expected to adopt FILIBUSTER for fault injection testing in the coming year, and since writing this dissertation, at least three additional teams have started writing FILIBUSTER tests.

To quote an anonymous developer at Foodly when first writing a FILIBUSTER test:

> *"My team really looks forward to the first filibuster test to work for this service so that we could add more."*

Similarly, one developer stated this when examining one of the application behaviors, provoked by FILIBUSTER, where an actual bug was discovered:

> *"right yeah i think this is a solid callout from the FILIBUSTER test"*

In fact, since the evaluations performed in this dissertation, additional bugs that match the design the fictional applications presented in this dissertation have been identified in services that have not yet employed fault injection testing. This indicates the pressing concerns addressed by this dissertation and the need for more widespread adoption of fault injection testing.

Future Work. Moving forward, several exciting avenues exist for improving the reach of *Service-level Fault Injection Testing*.

First, and most notably, FILIBUSTER has been used to understand the impact of resilience mechanisms such as circuit breakers and load shedding. More specifically, research into understanding how resilience mechanisms can introduce adverse effects when activated by disabling correctly functioning parts of the application to contain a failure has been performed by the author of this dissertation at Foodly [Mei+22].

Second, service failures are not the only thing that can affect microservice applications, as microservices also take dependencies on queues, databases, and caches. To that end, the author of this dissertation also supervised a master's thesis exploring this work: how can SFIT be extended to fault injection testing of databases? This work involved exploring not only faults related to connection failures and service unavailability but also the impact of *incorrect* (*i.e.*, byzantine faults) answers from services, such as empty string or null. This work resulted in not only extensions to the SFIT algorithm, through the use of a new recursive byzantine fault injection algorithm, but also extensions of the IDE plugin of FILIBUSTER to improve developer feedback and debugging of the impact of injected faults. This master's thesis not only improved on SFIT and *p*-SFIT but also resulted in the extension of the application corpus with several examples from Foodly of actual bugs that resulted in customer-affecting outages.

Finally, a long-term study of *p*-SFIT in an industrial context would be valuable in identifying how the fault injection approach could be further improved. However, such a study would involve a long-term collaboration with an industry partner as they evolved a microservice application and wrote tests to capture that behavior over time. This study would yield the most exciting results but may be limited by researchers' access to industrial microservice applications and the developers of those applications.

List of Algorithms

List of Figures

List of Tables

List of Definitions

Bibliography

[AAE16a] 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)*. IEEE. 2016, pp. 44–51.

[AAE16b] 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)*. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[ABH06] Cyrille Artho, Armin Biere, and Shinichi Honiden. "Exhaustive Testing of Exception Handlers with Enforcer". In: *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO'06. Amsterdam, The Netherlands: Springer-Verlag, 2006, 26–46. ISBN: 3540747915.

[Alv+16a] 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. Santa Clara, CA, USA: Association for Computing Machinery, 2016, 17–28. ISBN: 9781450345255. DOI: 10.1145/2987550.2987555. URL: https://doi.org/10.1145/2987550.2987555.

[Alv+16b] 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. Santa Clara, CA, USA: Association for Computing Machinery, 2016, 17–28. ISBN: 9781450345255. DOI: 10.1145/2987550.2987555. URL: https://doi.org/10.1145/2987550.2987555.

[AMS13] Sheeva Afshan, Phil McMinn, and Mark Stevenson. "Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 352–361. DOI: 10.1109/ICST.2013.11.

[ARH15a] 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. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 331–346. ISBN: 9781450327589. DOI: 10.1145/2723372.2723711. URL: https://doi.org/10.1145/2723372.2723711.

[ARH15b] 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. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 331–346. ISBN: 9781450327589. DOI: 10.1145/2723372.2723711. URL: https://doi.org/10.1145/2723372.2723711.

[Atc19] Lee Atchison. *How Service Tiers Can Help to Avoid Microservices Disasters*. https://thenewstack.io/how-service-tiers-can-help-to-avoid-microservices-disasters/. Accessed: 2023-07-21. 2019.

[Aud] *Audible*. https://www.audible.com. Accessed: 2021-05-21. 2021.

[aut22] Anonymized authors. "Suppressed title". In: ID: 2. 2022.

[Avi+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[Bal+18] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, and Theo Lynn. "Microservices migration patterns". In: *Software: Practice and Experience* 48.11 (2018), pp. 2019–2042.

[Bas+16] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. "Chaos Engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41. DOI: 10.1109/MS.2016.60.

[Bas+19a] 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)*. 2019, pp. 31–40. DOI: 10.1109/ICSE-SEIP.2019.00012.

[Bas+19b] 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)*. IEEE. 2019, pp. 31–40.

[BC12a] 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. Bern, Switzerland: Association for Computing Machinery, 2012, 281–294. ISBN: 9781450312233. DOI: 10.1145/2168836.2168865. URL: https://doi.org/10.1145/2168836.2168865.

[BC12b] 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. Bern, Switzerland: Association for Computing Machinery, 2012, 281–294. ISBN: 9781450312233. DOI: 10.1145/2168836.2168865. URL: https://doi.org/10.1145/2168836.2168865.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: 10.1109/MS.2016.64.

[BST02a] 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.

[BST02b] 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.

[Cel20] Cesare Celozzi. *Future-proofing: How DoorDash Transitioned from a Code Monolith to a Microservice Architecture*. https://doordash.engineering/2020/12/02/how-doordash-transitioned-from-a-monolith-to-microservices/. Accessed: 2023-07-21. 2020.

[Chaa] *Chaos Blade*. https://chaosblade.io. Accessed: 2022-06-05. 2022.

[Chab] "Chaos Engineering Saved Your Netflix Extreme stress testing of online platforms has become its own science". In: *IEEE Spectrum* 58.3 (2021), pp. 4–10. ISSN: 1939-9340. DOI: 10.1109/MSPEC.2021.9370069.

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

[Chad] *ChaosToolkit*. https://chaostoolkit.org. Accessed: 2022-06-05. 2022.

[Che+23] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. "{Push-Button} Reliability Testing for {Cloud-Backed} Applications with Rainmaker". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 1701–1716.

[Daw+96] S. Dawson, F. Jahanian, T. Mitton, and Teck-Lee Tung. "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection". In: *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*. FTCS '96. USA: IEEE Computer Society, 1996, p. 404. ISBN: 0818672617.

[Deu94] Peter Deutsch. "The eight fallacies of distributed computing". In: *URL: http://today. java. net/jag/Fallacies. html* (1994).

[Fid91] C. Fidge. "Logical time in distributed computing systems". In: *Computer* 24.8 (1991), pp. 28–33. DOI: 10.1109/2.84874.

[GKS05] 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*. 2005, pp. 213–223.

[Grea] *Gremlin*. http://www.gremlin.com. Accessed: 2021-05-21. 2021.

[Greb] *Gremlin*. http://www.gremlin.com. Accessed: 2021-05-21. 2021.

[Gun+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. "FATE and DESTINI: A Framework for Cloud Recovery Testing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, 238–252.

[GY11] 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: USENIX Association, Mar. 2011. URL: https://www.usenix.org/conference/nsdi11/model-checking-networked-system-without-network.

[Heo+16a] 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)*. 2016, pp. 57–66. DOI: 10.1109/ICDCS.2016.11.

[Heo+16b] 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)* (2016), pp. 57–66. DOI: 10.1109/ICDCS.2016.11.

[Hig20] High Scalability Blog. http://highscalability.com/blog/2020/4/8/one-team-at-uber-is-moving-from-microservices-to-macroservic.html. Accessed: 2023-07-21. 2020.

[Hil23] Jeremy Hillpot. *4 Microservices Examples: Amazon, Netflix, Uber, and Etsy*. https://blog.dreamfactory.com/microservices-examples/. Accessed: 2023-07-21. 2023.

[HSS23] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 419–432. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/huye.

[Jam+18a] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. "Microservices: The journey so far and challenges ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35.

[Jam+18b] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. "Microservices: The journey so far and challenges ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35.

[JC18] Jay Judkowitz and Mark Carter. *SRE fundamentals: SLIs, SLAs and SLOs*. https://cloud.google.com/blog/products/devops-sre/sre-fundamentals-slis-slas-and-slos. Accessed: 2023-07-21. 2018.

[JC19] Christina Terese Joseph and K Chandrasekaran. "Straddling the crevasse: A review of microservice software architecture foundations and recent advancements". In: *Software: Practice and Experience* 49.10 (2019), pp. 1448–1484.

[JD10] Lukasz Juszczyk and Schahram Dustdar. "Programmable Fault Injection Testbeds for Complex SOA". In: *Service-Oriented Computing*. Ed. by Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 411–425. ISBN: 978-3-642-17358-5.

[JGS11] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. "PREFAIL: A Programmable Tool for Multiple-Failure Injection". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, 171–188. ISBN: 9781450309400. DOI: 10.1145/2048066.2048082. URL: https://doi.org/10.1145/2048066.2048082.

[JM09] Ranjit Jhala and Rupak Majumdar. "Software Model Checking". In: *ACM Comput. Surv.* 41.4 (2009). ISSN: 0360-0300. DOI: 10.1145/1592434.1592438. URL: https://doi.org/10.1145/1592434.1592438.

[Jos+09] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. "A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks". In: *SIGPLAN Not.* 44.6 (June 2009), 110–120. ISSN: 0362-1340. DOI: 10.1145/1543135.1542489. URL: https://doi.org/10.1145/1543135.1542489.

[Jos+13] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. "SETSUDO: Perturbation-Based Testing Framework for Scalable Distributed Systems". In: *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. TRIOS '13. Farmington, Pennsylvania: Association for Computing Machinery, 2013. ISBN: 9781450324632. DOI: 10.1145/2524211.2524217. URL: https://doi.org/10.1145/2524211.2524217.

[Kil+07] 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: USENIX Association, Apr. 2007. URL: https://www.usenix.org/conference/nsdi-07/life-death-and-critical-transition-finding-liveness-bugs-systems-code.

[KKA95] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. "FERRARI: a flexible software-based fault and error injection system". In: *IEEE Transactions on Computers* 44.2 (1995), pp. 248–260. DOI: 10.1109/12.364536.

[Kub23] Kubernetes. *Configure Liveness, Readiness and Startup Probes*. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. Accessed: 2023-07-21. 2023.

[Lee+14] 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. Broomfield, CO: USENIX Association, 2014, 399–414. ISBN: 9781931971164.

[Lei+20] Leonardo Leite, Fabio Kon, Gustavo Pinto, and Paulo Meirelles. "Building a theory of software teams organization in a continuous delivery context". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2020, pp. 296–297.

[LF14] James Lewis and Martin Fowler. "Microservices: a definition of this new architectural term". In: *MartinFowler. com* 25 (2014), pp. 14–26.

[Li+21a] 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". In: *Information and Software Technology* 131 (2021), p. 106449.

[Li+21b] 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". In: *Information and Software Technology* 131 (2021), p. 106449. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106449. URL: https://www.sciencedirect.com/science/article/pii/S0950584920301993.

[Lin] *LinkedOut: A Request-Level Failure Injection Framework*. https://engineering.linkedin.com/blog/2018/05/linkedout--a-request-level-failure-injection-framework. Accessed: 2021-05-21. 2018.

[Lit] *Litmus*. https://litmuschaos.io. Accessed: 2022-06-05. 2022.

[Luk+19] 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. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303986. URL: https://doi.org/10.1145/3302424.3303986.

[MALH21] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. "The Migration Journey Towards Microservices". In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2021, pp. 20–35.

[Mat88] Friedemann Mattern. "Virtual Time and Global States of Distributed Systems". In: *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 1988, pp. 215–226.

[MC09a] 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*. 2009, pp. 379–388. DOI: 10.1109/DSN.2009.5270313.

[MC09b] 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*. 2009, pp. 379–388. DOI: 10.1109/DSN.2009.5270313.

[McC15] Caitie McCaffrey. "The Verification of a Distributed System: A Practitioner's Guide to Increasing Confidence in System Correctness". In: *Queue* 13.9 (2015), 150–160. ISSN: 1542-7730. DOI: 10.1145/2857274.2889274. URL: https://doi.org/10.1145/2857274.2889274.

[Mei+21a] 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*. 2021, pp. 388–402.

[Mei+21b] 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*. 2021, pp. 388–402.

[Mei+22] Christopher Meiklejohn, Lydia Stark, Cesare Celozzi, Matt Ranney, and Heather Miller. "Method Overloading the Circuit". In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. San Francisco, California: Association for Computing Machinery, 2022, 273–288. ISBN: 9781450394147. DOI: 10.1145/3542929.3563466. URL: https://doi.org/10.1145/3542929.3563466.

[Men+20] 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)*. 2020, pp. 114–124. DOI: 10.1109/ICSA47634.2020.00019.

[Mic23a] Michal Kaczmarski. *Which company is winning the restaurant food delivery war?* https://secondmeasure.com/datapoints/food-delivery-services-grubhub-uber-eats-doordash-postmates/. Accessed: 2023-11-26. 2023.

[Mic23b] Microsoft. *Bulkhead pattern*. https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead. Accessed: 2023-07-21. 2023.

[MRF18] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *ACM Trans. Comput. Syst.* 35.4 (2018). ISSN: 0734-2071. DOI: 10.1145/3208104. URL: https://doi.org/10.1145/3208104.

[MW18] 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. Pau, France: Association for Computing Machinery, 2018, 1733–1735. ISBN: 9781450351911. DOI: 10.1145/3167132.3167427. URL: https://doi.org/10.1145/3167132.3167427.

[Ner+20] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. "Design principles, architectural smells and refactorings for microservices: a multivocal review". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 3–15. DOI: 10.1007/s00450-019-00407-8. URL: https://doi.org/10.1007/s00450-019-00407-8.

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

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

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

[Netd] *Netflix*. https://www.netflix.com. Accessed: 2021-05-21. 2021.

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

[Nor17] Nora Jones. *AWS re:Invent 2017: Performing Chaos at Netflix Scale (DEV334)*. https://www.youtube.com/watch?v=LaKGx0dAUlo. Accessed: 2022-06-05. 2017.

[OEC16] Rory O'Connor, Peter Elger, and Paul M Clarke. "Exploring the impact of situational context—A case study of a software development process for a microservices architecture". In: *2016 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. IEEE. 2016, pp. 6–10.

[PJ16] Claus Pahl and Pooyan Jamshidi. "Microservices: a systematic mapping study." In: *CLOSER (1)* (2016), pp. 137–146.

[PP21] 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)*. Vol. 4. IEEE. 2021, pp. 195–202.

[PP22] 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)*. 2022, pp. 13–23. DOI: 10.1109/ICSA53651.2022.00010.

[PSS17] 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. Whistler, BC, Canada: Association for Computing Machinery, 2017, 30–36. ISBN: 9781450350686. DOI: 10.1145/3102980.3102986. URL: https://doi.org/10.1145/3102980.3102986.

[Res] *Resiliency in Distributed Systems*. https://blog.pragmaticengineer.com/resiliency-in-distributed-systems/. Accessed: 2023-07-21.

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

[RJ20a] Casey Rosenthal and Nora Jones. *Chaos engineering: system resiliency in practice*. O'Reilly Media, 2020.

[RJ20b] Casey Rosenthal and Nora Jones. *Chaos engineering: system resiliency in practice*. O'Reilly Media, 2020.

[Rob+12] 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". In: *Queue* 10.9 (2012), 20–28. ISSN: 1542-7730. DOI: 10.1145/2367376.2371297. URL: https://doi.org/10.1145/2367376.2371297.

[RPA22] Kamala Ramasubramanian, Eliana Phillips, and Peter Alvaro. "Mining microservice design patterns". In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 190–195.

[SBG10] 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: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/ssv10/dbug-systematic-evaluation-distributed-systems.

[Sig+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[SKT21] 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. Online, United Kingdom: Association for Computing Machinery, 2021, 4–10. ISBN: 9781450383363. DOI: 10.1145/3447851.3458740. URL: https://doi.org/10.1145/3447851.3458740.

[SMAP19] 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. Limassol, Cyprus: Association for Computing Machinery, 2019, 1243–1251. ISBN: 9781450359337. DOI: 10.1145/3297280.3297402. URL: https://doi.org/10.1145/3297280.3297402.

[SS+21] 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*. Vol. 1077. 1. IOP Publishing. 2021, p. 012065.

[SSZ20] Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. "Exploring the microservice development process in small and medium-sized organizations". In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2020, pp. 453–460.

[STVDH18] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232.

[Tig+20] 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. Virtual Event, Germany: Association for Computing Machinery, 2020. ISBN: 9781450377690. DOI: 10.1145/3424771.3424812. URL: https://doi.org/10.1145/3424771.3424812.

[TLP18] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural patterns for microservices: a systematic mapping study". In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. -. 2018.

[Tol+19] Saulo Soares de Toledo, Antonio Martini, Agata Przybyszewska, and Dag IK Sjøberg. "Architectural technical debt in microservices: a case study in a large company". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2019, pp. 78–87.

[Tuc+18] Haley Tucker, Lorin Hochstein, Nora Jones, Ali Basiri, and Casey Rosenthal. "The business case for chaos engineering". In: *IEEE Cloud Computing* 5.3 (2018), pp. 45–54.

[Tyl18] Tyler Lund. *AWS re:Invent 2018: Chaos Engineering and Scalability at Audible.com (ARC308)*. https://www.youtube.com/watch?v=7uJG3oPw_AA. Accessed: 2022-06-05. 2018.

[Val+20] 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". In: *Programming and Computer Software* 46.8 (2020), pp. 594–608. DOI: 10.1134/S0361768820080253. URL: https://doi.org/10.1134/S0361768820080253.

[Wal+96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. "A note on distributed computing". In: *International Workshop on Mobile Object Systems*. Springer. 1996, pp. 49–64.

[Was+21a] 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*. 2021, pp. 201–210.

[Was+21b] 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*. 2021, pp. 201–210.

[WKR21] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. "Promises and challenges of microservices: an exploratory study". In: *Empirical Software Engineering* 26.4 (2021), pp. 1–44.

[WLS20] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. "A systematic mapping study on microservices architecture in devops". In: *Journal of Systems and Software* 170 (2020), p. 110798.

[XSZ08] Bin Xin, William N. Sumner, and Xiangyu Zhang. "Efficient Program Execution Indexing". In: *SIGPLAN Not.* 43.6 (June 2008), 238–248. ISSN: 0362-1340. DOI: 10.1145/1379022.1375611. URL: https://doi.org/10.1145/1379022.1375611.

[Yab+10] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. "Predicting and Preventing Inconsistencies in Deployed Distributed Systems". In: *ACM Trans. Comput. Syst.* 28.1 (2010). ISSN: 0734-2071. DOI: 10.1145/1731060.1731062. URL: https://doi.org/10.1145/1731060.1731062.

[Yan+09] 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: USENIX Association, Apr.

2009. URL: https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems.

[Zha+19] 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". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: 10.1109/TSE.2019.2954871.

[Zha+21a] 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. Seattle, WA, USA: Association for Computing Machinery, 2021, 458–472. ISBN: 9781450386388. DOI: 10.1145/3472883.3486986. URL: https://doi.org/10.1145/3472883.3486986.

[Zha+21b] 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". In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2534–2548. DOI: 10.1109/TSE.2019.2954871.

[Zho+18a] 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". In: *IEEE Transactions on Software Engineering* 47.2 (2018), pp. 243–260.

[Zho+18b] 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". In: *IEEE Transactions on Software Engineering* 47.2 (2018), pp. 243–260.