Method Overloading the Circuit

Christopher Meiklejohn*
Carnegie Mellon University
Pittsburgh, PA, USA
cmeiklej@cs.cmu.edu

Lydia Stark†
University of Alaska Anchorage
Anchorage, AK, USA
lrstark2@alaska.edu

Cesare Celozzi
DoorDash, Inc.
San Francisco, CA, USA
cesare.celozzi@doordash.com

Matt Ranney
DoorDash, Inc.
Pittsburgh, PA, USA
matt.ranney@doordash.com

Heather Miller
Carnegie Mellon University
Pittsburgh, PA, USA
heather.miller@cs.cmu.edu

ABSTRACT

Circuit breakers are frequently deployed in microservice applications to improve their reliability. They achieve this by short circuiting RPC invocations issued to overloaded or failing services, thereby relieving pressure on those services and allowing them to recover. In this paper, we systematically examine the state of the art in industrial circuit breakers designs. We first present a taxonomy of existing, open-source circuit breaker designs and implementations based on a systematic mapping study. We then examine the relationship between these circuit breaker designs and application reliability. We make a clear case that incorrect application of circuit breakers to an application can hurt reliability in the process of trying to improve it. To address the deficiencies in the state of the art, we propose two new circuit breaker designs and provide guidance on how to properly structure microservice applications for the best circuit breaker use. Finally, we identify several open challenges in circuit breaker usage and design for future researchers.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**.

KEYWORDS

fault tolerance, fault injection, verification

*Research performed as an independent contractor for DoorDash, Inc.
†Research performed as an REUSE student at Carnegie Mellon University.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563466

ACM Reference Format:
Christopher Meiklejohn, Lydia Stark, Cesare Celozzi, Matt Ranney, and Heather Miller. 2022. Method Overloading the Circuit. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563466

1 INTRODUCTION

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

Fault injection is typically used to identify vulnerabilities in application code before [36] and after [49] deployment to production. One example is identifying latent application bugs, activated by the unavailability of a service dependency, as part of functional testing [36]. Fault tolerance both compliments and leverages fault injection by ensuring that a fault does not propagate to other services and result in a cascading failure of the application. For example, when an undetected latent application bug is activated by the unavailability of a dependency, checking that the invoking service dynamically re-configures itself to avoid invoking the unavailable dependency until it becomes available again.

The desired outcome from the use of fault tolerance in microservice applications is to ensure that application code can both tolerate and react, in an application-specific way, to both non-fatal and fatal errors of service dependencies. For example, by hiding malfunctioning features in the case of non-fatal errors, or asking the user to retry their request later or to contact customer support in the case of fatal errors.

Rather unfortunately, the fault tolerance techniques used today are primitive in design, focusing primarily on the failure of an entire service instead of considering the case where

only a single method of a service may be malfunctioning due to an application bug (*i.e.,* runtime exception.) Under this design assumption, one such fault tolerance technique, *circuit breakers*, may simultaneously reduce the reliability of an application, while trying to improve it, by dynamically reconfiguring the application to avoid invoking all methods of a dependency when only certain methods of that dependency are producing errors. This specific case represents a growing concern, as the incremental development and increased deployment cadence, common to microservice applications, results in an increased chance of shipping bugs that affect only a subset of a given service's functionality.

Inspired by observations made at DoorDash, a food ordering and delivery platform built using a microservice architecture composed of over 500 services, we systematically examine the state of the art of one of the most commonly used fault tolerance techniques for microservice applications today: *circuit breakers*. We develop a taxonomy of existing circuit breaker designs and identify two key properties that may impact their successful application for fault tolerance: first, their *scope*, or where the circuit breakers can be applied in application code, and their *transparency*, whether the application or use of a circuit breaker is visible in application code. We use this taxonomy to examine the relationship between circuit breaker design and application reliability. We demonstrate that, when it comes to using circuit breakers to contain failures as a result of application bugs, that all existing designs are *at odds* with commonly used abstraction mechanisms in modern, high-level programming languages.

We identify two clear outcomes of this tension between circuit breaker usage and abstraction. First, application developers must perform extra, and often redundant, work in order to achieve the proper fault tolerance scope when failure inevitably occurs. This work may not be immediately obvious to developers when adding new features if the application of the circuit breaker is in shared functions that are being reused, but not directly modified. Second, that transparency, while a desirable property, may complicate the correct scoping of circuit breakers by hiding its usage from application developers. With transparency, application developers may not even be aware that code with a circuit breaker needs refactoring nor that a circuit breaker is present.

Finally, to address this tension, we propose two new circuit breaker designs that mitigate the impact of abstraction on circuit breaker scope — thereby enabling code reuse and reducing duplication — and provide guidance on selection of the correct design based on the application structure.

2 BACKGROUND: MICROSERVICES

Microservice architectures are the most common software architectural style chosen by the developers of rich-web and mobile applications today. In fact, all companies in the Fortune 50 are hiring developers to work on microservice applications, as some component of their infrastructure or application relies on microservices [36]. From an architectural point of view, microservice architectures decompose an application's implementation into separate services by business logic or feature: these services then take dependencies on other services to deliver their functionality. Each of these services then are developed and deployed by independent teams to improve the delivery of software at scale [34].

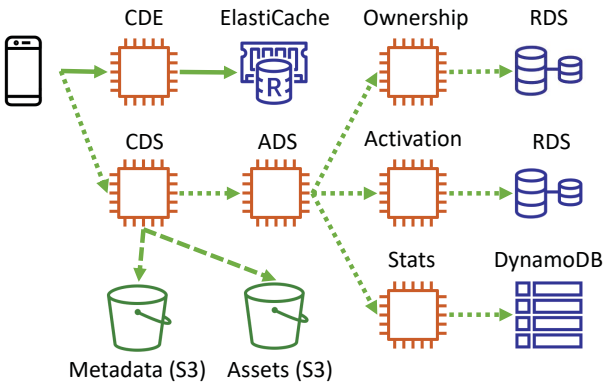

Figure 1: Audible, with audiobook retrieval process.

2.1 Audible

To provide the reader with an understanding of how applications are structured, we use an example from our prior work [36]. In Figure 1, we depict the microservice application behind Audible [1]. This application uses both services that are developed by Audible and cloud services, such as databases. We highlight the audiobook retrieval process to demonstrate how all of the services in this microservice application work in concert to deliver end-user functionality.

When a user requests an audiobook using the Audible app, it first issues a request to the Content Delivery Engine to find the URL of the Content Delivery Service (CDS) that contains the audiobook assets. The app then makes a request to that CDS. The CDS first makes a request to the Audible Download Service (ADS). The ADS is responsible for first verifying that the user owns the audiobook. Next, the ADS verifies that a DRM license can be acquired for that audiobook. Finally, it updates statistics at the Stats service before returning a response to the CDS. If ownership of the book cannot be verified or a license cannot be activated, an error response is returned to the ADS, which propagates back to the user through an error in the client. Once the ADS completes it work, the CDS contacts the Audio Assets service to retrieve

the audiobook assets. Then, the CDS contacts the Asset Metadata Service to retrieve associated metadata. Finally, these assets are returned to the user's app for playback.

To understand how partial failure can result in application outages when not tolerated, we describe an outage that Audible experienced between 2017-2018. There is an assumption that if an audiobook exists in the system and the assets are available, the metadata will also be available. However, this may not always be the case. In fact, and as demonstrated in [36], this precise fault can be detected through fault prevention techniques that use fault injection. 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, in this outage the invariant was violated, for reasons not disclosed by Audible and are presumably related to operator or database error. This resulted in a cascading failure and subsequent outage of the Audible application. The outage is a result of several faults. Starting with the lack of error handling, a generic error is propagated back to the mobile application that, upon receiving the generic error, assumes the failure was transient and consequently retries the request a finite number of times. When all of the retries return failure, a generic error is provided back to the user in the mobile application that causes the user to issue a retry. This combination of user-initiated retries for requests that will ultimately fail, paired with the combination of a popular audiobook, is enough to exhaust available compute capacity and cause the application to fail.

Takeaways. If the developers had used a fault tolerance technique, such as a circuit breaker, it could have prevented the system from processing client-initiated retries upon repeated failure through dynamic reconfiguration. However, use of a circuit breaker in this case requires proper scoping. For example, this circuit breaker would have to prevent further requests for *that specific audiobook*. A circuit breaker applied too broadly — to the request that is used to retrieve *any audiobook* — would have effectively induced an application outage while simultaneously trying to prevent one. Therefore, developers *must be aware of application structure* when selecting and implementing a circuit breaker.

2.2 DoorDash

DoorDash is a food ordering and delivery platform built using a microservice architecture composed of over 500 services, primarily written in the Kotlin programming language, which runs on the JVM. Our decision to adopt microservices was made to improve developer productivity at scale [6]. Each service in our microservice application platform provides a specific set of functionality that is grouped by business logic (*e.g.,* ordering, delivery) and the software lifecycle for those services are managed by an independent team.

While DoorDash has only just started our journey into fault prevention through the use of fault injection [19], we rely heavily on load shedding and circuit breaking to ensure the fault tolerance of our platform. More specifically, we combine cluster orchestration, circuit breaking, and load shedding towards achieving our reliability goals.

Cluster orchestration is used to support rolling deploys of services where those services are deployed into replica sets using load balancers to distributed load across all of the instances in the replica set. Services are automatically health checked and removed from the cluster when malfunctioning or are in a failed state due to a crash failure. Auto-scaling rules and restart policies are used to ensure that a minimum set of nodes are operational at any given time.

Load shedding is used at each service instance to ensure that services refuse, or short-circuit, incoming requests when the service is either already overloaded or at risk of an overload should they process the incoming request.

Circuit breaking at each service is used to prevent repeated invocation of a failing or malfunctioning service by dynamically re-configuring the application to short-circuit the remote call until the remote service begins functioning properly. However, since remote calls are automatically retried at least once — only in the event of a failure response and not a timeout error which may indicate a failure of a transitive dependency — and since cluster orchestration automatically removes instances that are failing their health checks, more often than not the retried request will succeed because it will be routed a non-crashed replica upon retry.

Takeaways. Circuit breakers are used to minimize the impact of service dependencies that may be malfunctioning in application code and not affect all methods of the dependent service. For example, application bugs that trigger runtime exceptions such as `ArrayOutOfBoundsException` or `NullPointerException` due to the arguments provided by the invoker or cascading failures due to application bugs triggered by transitive downstream dependencies.

3 CIRCUIT BREAKERS

Two commonly used fault tolerance techniques used in microservice applications are *circuit breakers* and *load shedding*.

Circuit breakers [41, 47], named after their electrical circuit counterparts [20], are implemented where a dependency is invoked, in order to prevent repeated invocation when the remote service either fails to respond, responds too slowly, or returns a failure response too often. Load shedding [25, 26, 39, 52], also named after the similar technique used by electrical power companies (*i.e.,* rolling blackout [21]) to avoid system risk by limiting access to electrical power, is typically implemented at the invoked service, where requests can be short-circuited by immediately returning an error if a service

is in an already overloaded condition or risks overloading if it processes the request. Previous academic work has considered these two variations of the same mechanism, with the former being an instance of a client-side circuit breaker and the latter an instance of a server-side circuit breaker [28, 40].

Circuit breakers have seen adoption as a fault tolerance technique in large, industrial microservice applications such as Netflix, with their open-source Hystrix [14] library. While early designs of circuit breakers required developers to wrap invocations to remote services in conditional branches and manually increment success or failure counters used by the conditional's predicate, the momentum is shifting away from this design. Hystrix [14], for example uses method annotations in Java to guard invocations to that method using a circuit breaker. The `circuitbreaker` [11] library in Python uses a similar annotation-based design. One obvious assumption of this design is that circuit breaking should be performed at the level of the *method* issuing the remote procedure call (RPC). However, this design is problematic when the method used to invoke the RPC is either parameterized or overloaded to support reuse by multiple callers who may be invoking different RPC methods on the same service or different RPC methods on different services.

Most recently, Netflix has spearheaded an initiative within Envoy [10], a communication proxy for microservice applications that is commonly used in combination with the Kubernetes [13] cluster manager, to integrate circuit breakers directly into the infrastructure layer. This design eliminates the need for manual developer-written annotations, thereby giving developers circuit breakers "for free." This design could be seen as a reaction to the potential ad-hoc and error prone use of circuit breakers that require manual annotations: for example, where a developer may simply fail to implement one where necessary and leave the application open to a cascading failure as a result of a unhandled fault. Transparency may also adversely affect scoping by failing to alert the developer through a visual cue that a certain application abstraction choice — for example, method indirection, if transparency is used in the application — is not compatible with the current circuit breaker design.

Generally speaking, circuit breakers interpose on the RPCs that are issued between two different services. They observe the responses from each RPC issued by the service and accumulate counters for various metrics such as response time, number of errors received, and number of outstanding requests using a sliding window. If implemented in the application, this interposition is typically either done using automatic instrumentation provided by the language runtime, through the use of interceptors or decorators, or explicitly by inspecting the request and response prior to, and after, an RPC invocation. If implemented in the infrastructure, this interposition is typically done using a proxy service.

Circuit breakers start in the *closed* state where RPCs are issued as normally. If response times or error counts exceed a threshold within this window, the circuit moves into an *open* state where RPCs are short-circuited and a predetermined error response is returned to the application to indicate that the circuit is open. From there, circuit breakers eventually, dependent on their configuration, move into the *half-open* state where some requests are allowed through in order to determine if the circuit should move back into a closed state: this is the circuit breaker's initial state. From there, any subsequent failure moves it back into the open state.

Implementations may provide the ability [16, 17] to perform the process of determining whether a circuit should move back into the closed state asynchronously using an API provided by the invoked service. This is commonly referred to as a health check. These health check APIs may be provided by a cluster orchestration system (*e.g.,* Kubernetes [13]) or by the service itself, if the health of the service is based on non-trivial application logic or the availability of other dependent services or data stores. In short, these health checks return either a success or error response based on whether the service is able to accept more requests using cluster state, service state, or other application metrics.

3.1 Taxonomy Construction

Our methodology for taxonomy construction is both motivated by our experience of deploying and debugging circuit breakers in production and builds upon a previous academic systematic mapping study [28]. The taxonomy is not meant to be comprehensive. Rather, this paper focuses specifically on a software engineering perspective: abstraction, when applying circuit breakers to application code.

3.1.1 Methodology. To identify circuit breaker implementations, we used our own experience at DoorDash with deploying circuit breakers augmented with the implementations identified by the Falahah *et al.* [28] study.

In terms of infrastructure-level circuit breakers, we were only aware of one design, implemented in Envoy as an "adaptive concurrency control filter" [5] and both integrated in Kubernetes and packaged together as Istio. This is the infrastructure-level circuit breaker that we have started using at DoorDash. In terms of application-level circuit breakers, we combined our own knowledge of applying circuit breakers at DoorDash with the implementations identified by the aforementioned systematic mapping study.

For each application-level implementation, we used their open-source documentation to identify how each circuit breaker could be used. We were specifically concerned with two aspects of use. First, *how* the circuit breaker could be applied in application code. For example, by wrapping the RPC invocation in a conditional based on circuit breaker state,

applying to a method that encapsulated an RPC invocation, or attaching to an RPC client. Second, *whether* the circuit breaker was visible in application code or applied automatically by a supporting library, framework, or other automatic instrumentation method. These categories emerged from the data organically and were refined as each implementation was examined as necessary.

3.2 Taxonomy

Circuit breakers may be implemented in either the *application-* or in the *infrastructure*-layer. Application-layer circuit breakers may be used *explicit*ly or *transparent*ly. Transparent usage does not require developers implement circuit breakers in the application code, but are installed through some manner of automatic instrumentation or use of an external library. Explicit usage requires that developers manually install them in application code at the call site, surrounding function, or elsewhere. Circuit breakers may be installed at the *callsite*, the encapsulating *method*, or on the RPC *client*.

3.2.1 Explicit. There are three explicit circuit breakers, of increasing coarseness: callsite, method, and client.

A *callsite-explicit* circuit breaker is installed directly at an RPC invocation site and wraps the invocation using a conditional. This conditional reads the state of the circuit breaker to determine whether or not to actually invoke the RPC. Developers are responsible for incrementing both success and failure counters that are used by the conditional's predicate to control the circuit breaker's state. For example, Akka's `CircuitBreaker` [3] for Java, Ameria's `CircuitBreaker` [4] for Java, App-vNext's Polly for .NET [7], `circuitbreaker` [15] for Go, Comcasts's jrugged [8] for Java, and pybreaker [9] for Python can all be used explicitly at each callsite.

A *method-explicit* circuit breaker is installed using annotations on a method invoking an RPC. This annotation indicates the thrown exceptions or return values that should increment the error counter used by the circuit breaker. For example, Netflix's Hystrix [14] for Java, Resilience4j [18] for Java, App-vNext's Polly for .NET [7], and pybreaker [9] for Python all provide the ability to decorate methods or other functional (*e.g.,* lambda) interfaces.

A *client-explicit* circuit breaker is installed by explicitly adding a decorator on the RPC client (*e.g.,* gRPC interceptor, HTTP client decorator) to enable the circuit breaker. This circuit breaker can be shared across multiple clients, if in a programming language that shares objects by reference. For example, Ameria's `CircuitBreaker` [4] for Java, Resilience4j [18] for Java, App-vNext's Polly for .NET [7], and `circuitbreaker` [15] for Go all allow circuit breakers to decorate a RPC client.

Client-explicit circuit breakers can also be used differently by the application, either by sharing a single client or instantiating different clients where needed in the application. With a *1 client-explicit* circuit breaker, a single client is used for multiple RPCs from different call sites. This may be important to reduce overhead of establishing a new RPC client, resolving DNS, and potentially setting up required TLS connections. With an *N client-explicit* circuit breaker, a new RPC client is used for each RPC invocation. Rather obviously, when a new client is used at each invocation site, the circuit breaker is equivalent to a callsite-explicit circuit breaker.

3.2.2 Transparent. Any design can be made transparent by abstraction (*e.g.,* a client library used by developers for the creation of RPC clients) or through the use of automatic instrumentation. Typically, these implementations are not open-source, but kept private for internal use, as they contain logic specific to that company's RPC usage. Therefore, we highlight one example that demonstrates an implementation strategy that is widely used.

A *client-transparent* circuit breaker is the same as a *client-explicit* circuit breaker using decorators or interceptors, but is automatically installed through automatic instrumentation (*e.g.,* `javaagent`) or inclusion of a third party RPC library with the circuit breaker already attached. For example, DoorDash's Hermes [2], a RPC library that wraps GRPC invocations, automatically attaches circuit breakers to each RPC client using decorators. Similar to client-explicit circuit breakers, it can be used in the *1 client* or *N client* style.

We did not find any instances of a *method-transparent* or a *callsite-transparent* circuit breaker. This does not indicate that these designs do not exist or have not been implemented, but rather states that none of the implementations that we examined exhibited these designs. We provide a description below on why we believe these designs may not exist.

A *method-transparent* circuit breaker would be automatically installed at each method that invokes an RPC. This may be non-trivial for several reasons in practice.

First, while a static analysis may be able to identify where RPCs are defined in application code, this relies on knowing precisely what libraries, classes, modules, methods, or functions called by the application result in RPCs. This is complicated by RPC mechanisms that use code generation, as the circuit breaker library would have to be aware of application-specific RPC stub classes that are generated at compile time, in order for this to be complete.

Further complicating the analysis is the use of higher-order functions in modern high-level programming languages where methods or functions can be passed between functions to be executed later. This would require a sufficiently advanced intraprocedural analysis at or before compile time,

which may not be possible nor practical, and would necessarily have to either under- or over-approximate where these calls are made to keep the analysis tractable.

Once the analysis is completed, the circuit breakers would have to be automatically installed at either compile time — through bytecode transformations, macro application, or using a technique like aspect-oriented programming.

A *client-transparent* circuit breaker would be automatically installed at each callsite where an RPC is invoked. Similar to the *method-transparent* circuit breaker, it would require a similar analysis and the same bytecode transformation or macro application required to rewrite this code to use a circuit breaker. This circuit breaker would provide both the most granular scope, that of the precise callsite where the RPC invocation occurs, and minimize developer overhead in using the circuit breaker through transparent application. In that respect, out of the possible designs outlined in this section, it is the most desirable.

3.2.3 Sensitivity. Sensitivity is the property that indicates how the circuit breaker state — the counters that determine state transitions between open, closed, and half-open — is partitioned with respect to the RPC that is being invoked. For example, whether that circuit breaker is sensitive to the invoked service, the RPC method, or the arguments provided to the RPC when invoked.

In the majority of cases, sensitivity is inherited from the circuit breaker's scope. For example, when using a client-explicit circuit breaker, all RPCs that are issued using that client, despite the specific RPC method or arguments provided, will affect a single state for that circuit breaker. Therefore, any failures of one RPC method will be counted against the circuit breaker used for all RPC methods. Similarly, a method-explicit circuit breaker used on a method that is parameterized for the RPC service, method, and arguments to invoke, will increment success and failure counters for any and all RPCs issued by that method.

When it comes to circuit breakers that are scoped to the callsite, it depends on the transparency. For example, with a client-explicit circuit breaker, the developer has control over when the success and failure counters are incremented, as they are incremented manually. Therefore, they may choose to increment the success and failure counters based on arbitrary conditions, along with the returned value from the RPC invocation. With the client-transparent circuit breaker, success and failure counters would have to be incremented uniformly, as it is generally undecidable to determine what in-scope variables are used to represent the RPC service, RPC methods, and RPC arguments when issuing the RPC. This is true because different methods used to issue RPCs may take parameters in different positions or use global variables to alter execution at runtime.

3.2.4 Modifying Sensitivity. One example of a circuit breaker implementation that allows for modifications of the sensitivity is Armeria's circuit breaker. Using a `CircuitBreakerRule` at the location where the circuit breaker is instantiated, developers are able to specify that a circuit breaker should be sensitive to the RPC host, RPC method, or RPC path. These terms can be a bit confusing in practice as Armeria's circuit breaker client is designed specifically for HTTP requests, but can be used with gRPC, as HTTP is the underlying transport mechanism for gRPC. Therefore, when issuing a HTTP-based RPC request, the host is the destination DNS or IP name, the method is one of the HTTP verbs (*e.g.,* GET, PUT, POST) and the path is the URI, excluding scheme and port. For gRPC, host represents the hostname of the gRPC endpoint, the method is the name of the RPC method, and the path a combination of RPC service name — taken from the generated gRPC stub — and RPC method. Finally, developers can also use a custom function to prepend or append strings to these attributes to partition circuit breakers in the manner they wish.

3.3 DoorDash

At DoorDash, our Hermes library [2] — an application-level client-transparent circuit breaker library — is built on top of Armeria's circuit breaker library. Its usage in most applications follows an N-client style, where a single RPC client is reused for repeated RPC calls; however, at least one client exists *per invoked RPC service*. For example, callers must provide the RPC service's definition when retrieving an RPC client, thereby forcing *at least one* client per RPC service that is invoked. In general, we cannot prevent developers from retrieving more than a single client per RPC service.

It associates a circuit breaker with each client generated by the library and provides a default configuration for that circuit breaker containing thresholds and the errors that should be considered when incrementing the success and failure counters. Because it is a transparent circuit breaker, application developers working in service code do not see any code related to circuit breakers at all, unless they provide a overridden circuit breaker configuration at the location where they retrieve the client from the library: this is typically done using dependency injection in the code that initializes the service. In the majority of cases, circuit breaker configurations are not changed through the use of overrides.

The circuit breakers that are associated with each client are scoped using the `perPath` circuit breaker rule to provide sensitivity to both the invoked RPC service and method. This rule is shared across both gRPC and HTTP clients, thereby providing sensitivity to service and method for gRPC, but with more sensitivity for HTTP methods than just the HTTP verb. Since RPCs are issued to service names that resolve

to load balancer instances that distribute requests across a pool of nodes, they are not sensitive to the individual host. Instead, cluster orchestration and health checks are used to handle and respond to crash failures of individual instances.

3.4 Takeaways

When it comes to using circuit breakers as a fault tolerance mechanism for application bugs, the key is maximizing circuit breaker *sensitivity*. For example, bugs may affect a single RPC method of a service or may be dependent on the arguments provided to that method. However, as we have seen, sensitivity is often tied to the *scope* of the circuit breaker.

Transparency, a property that is orthogonal to both the scope and the sensitivity of the circuit breaker, is a desirable property for organizations. It mitigates the risk of a developer forgetting to use circuit breakers in their code, which would leave the application open to risks of cascading failures or other outages. In practice, transparency is only achievable with certain scopes: for example, through the use of an infrastructure-level (*e.g.,* Envoy) or a client-transparent application-level (*e.g.,* DoorDash's Hermes) circuit breaker.

As we will demonstrate using the following two case studies, the impact on sensitivity is not just related to scope, but is further exacerbated through abstraction use in the application code itself.

4 CASE STUDY #1: INDIRECTION

In order to demonstrate the impact of application design and abstraction on circuit breaker choice and it's sensitivity, we present the first of two different case studies. These case studies are inspired by observations on application design and circuit breaker usage at DoorDash. They have been both abstracted and simplified for confidentiality and exposition. An implementation of these examples in Python is available as an extension to the Filibuster microservice application corpus [36] on GitHub [12].

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

One possible implementation of the orders service, presented in Python*esque* pseudocode, is depicted in Figure 2. Here, orders exposes three RPC endpoints: create, update, and delete. Each of these three endpoints takes a dependency on the auth service in order to manage the life cycle of the payment that is associated with the order. We draw the reader's attention specifically to the delete endpoint that is used for order cancellation. When the orders service receives a request to cancel an order, it first performs some business logic and then issues an RPC to the auth service to cancel the payment for the cancelled order. If that succeeds, it responds with a success; otherwise, it returns an error that propagates back to the user and asks them to try again. In Figure 3, we depict an application of method indirection used to reduce duplication: we highlight the changes from Figure 2.

```
@orders.method("create")
def order_creation(...):
    try:
        res = rpc(auth, "create", [order_id, amount])
        return order_id
    except Exception as e:
        // ...

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

@orders.method("delete")
def order_cancellation(order_id : String):
    res = rpc(auth, "delete", [order_id])
```

Figure 2: *Orders* service (abr.) with 3 RPC methods.

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

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

def order_cancellation(order_id : String):
    res = issue_auth_rpc("delete", [order_id])

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

Figure 3: Figure 2 (abr.) with function indirection.

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

Figure 4: Figure 3 (abr.) with *method-explicit* CB.

```
def order_creation(...):
    res = issue_auth_create_rpc([order_id, amount])

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

Figure 5: Figure 2 (abr.) with proper sensitivity.

4.1 Adding Circuit Breakers

At this point, the developer might want to install a circuit breaker to guard against the unavailability or malfunctioning of the auth service. To do this, the developer chooses

a popular circuit breaker library; in our example, we use the `circuitbreaker` library for Python, one example of a *method-explicit* circuit breaker. In Figure 4, we highlight the modifications needed to install this circuit breaker. Here, the circuit breaker annotation that is placed on the method issuing the RPC denotes that any time a `RPCException` is thrown, the circuit breaker's error counter should increase; any other thrown exceptions should not increment the circuit breaker's counters. We imagine that `RPCException` represents a generic exception base type for all possible RPC exceptions. There are two problems with this approach.

(1) First, this design assumes that all RPCs issued by the orders service using the `issue_auth_rpc` helper to different methods of the auth service, will all throw the same exceptions. This simply may not be true. For example, RPCs may use different exception types to indicate different error conditions, where only a subset of types, depending on invoked service or method, should affect the circuit breakers counters. Similarly, exceptions may be parameterized with different error codes, as is the case with gRPC, where only a subset of the possible parameterizations should affect circuit breaker counters. In the case of gRPC specifically, the application may not want to affect circuit breaker counters on a gRPC exception where the error code indicates resource not found, as this may be an non-fatal error condition for this application — this is in direct comparison to the Audible example where a resource not found indicates a fatal error. Therefore, the method indirection used in this example implies that all RPC failures should be treated in the same manner.
(2) Second, if an application bug in the auth service happens to cause just one particular RPC method to return errors (*i.e.,* delete), the circuit breaker will short-circuit *all* RPCs to the auth service (*i.e.,* create, update, delete) even when the other two endpoints may not be malfunctioning. In short, our reliability measures have disabled correctly functioning endpoints when trying to prevent against the malfunctioning of one specific endpoint. Therefore, the method indirection used in this example implies that all RPC failures exhibited by the specific method executing the RPC should be treated in the same manner.

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

> ✓ **Partitioning:** To increase sensitivity, developers must refactor code to partition RPC invocations that need separate circuit breaking.

4.2 Increasing Sensitivity

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

We depict this refactoring in Figure 5. By structuring our code in this manner, it allows developers to specify precisely the failures that should affect the circuit breaker for each individual method. We demonstrate this using a different exception type for each method.

Rather obviously, and as made clear by this example, this is a rather counterintuitive implementation choice: in fact, this implementation choice only makes sense when circuit breakers are present as it goes against many common programming conventions regarding code reuse. In fact, if we look at the progression from Figure 2 to Figure 5, the resulting implementation is arguably the most verbose, done only to support circuit breaker behavior.

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

> ✓ **Scope Partitioning:** When partitioning to increase sensitivity, partitioning must be performed with respect to the *scope* of the circuit breaker, *at minimum*: for example, the *callsite*, the *method*, or the *client*.

5 CASE STUDY #2: NONDETERMINISM

In the previous section, we identified how a minor refactoring of the orders service, in order to abstract the method used for RPC invocation, introduced several complexities when it came to circuit breakers. In that example, existing circuit breaker designs only provided the proper sensitivity when the application was designed with circuit breakers in mind.

In this section, we explore how the use of abstraction, to support a minor variation on the same set of application behaviors, complicates the use of circuit breakers. In short, we demonstrate that if applications need to support these types of designs, existing circuit breakers are only sufficient under one, of many, possible different application designs.

An implementation of these examples in Python is available as an extension to the Filibuster microservice application corpus [36] on GitHub [12].

For this case study, we are going to expand our food delivery application to support takeout orders in addition to delivery. This might sound straightforward; however when a takeout order is cancelled, a different process needs to be performed to cancel the order. As most of the code needs to be parameterized on whether or not an order is a takeout or

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Possible parameterizations of Figure 5 (abr.) to support both delivery and takeout.

delivery order, the developers have a number of design decisions that they now face, which we will discuss below. We assume, as a starting point, the refactored implementation presented in the previous section: see Figure 5.

When implementing this new functionality, the developers of the application realize that the code needs to be parameterized based on whether the order is a takeout or delivery order. There are six possible choices for this parameterization.

First, the developers have to decide on whether or not they want to parameterize the cancellation method's name on whether it is delivery or takeout. If they decide this, they then have to decide which of the following subsequent parameterizations they want: parameterization of the invoked RPC service, method, or arguments. Second, the developers may also choose to include the order type in the parameters of the cancellation method. The same three choices apply for the second parameterization: the RPC service they invoke; the RPC method they invoke; or the invocation arguments.

We depict these parameterizations in Figure 6. In the following discussion of these parameterizations, we refer to the method invoked on the orders service as *invoking*: this indicates that it is currently executing. When discussing the method on the auth service, we will refer to it as *invoked* to indicate that it is called by the invoking service.

(1) by ***Invoking Method*** and
   (a) ***Invoked Service.*** (Figure 6a) Requires that developers both duplicate the invoking method's functionality, once for each order type.
   (b) ***Invoked Method.*** (Figure 6b) Improves on **1a**, as while it still requires duplication of code on the invoking side, it does not require creation of a new service as it parameterizes the method that it calls.
   (c) ***Invoked Args.*** (Figure 6c) Further improves on **1b**, as while it still requires the same duplication, it does not require creation of a new method on the invoked service, but rather allows the developer to use arguments for control flow.

(2) by ***Invoking Args*** and
   (a) ***Invoked Service.*** (Figure 6d) Reduces the need for function duplication by only modifying the argument list for the invoking method to contain the order type. From here, the developer can use the type parameter to derive the service that should be invoked; however, it does require the creation of a new service and that may require the same duplication as we saw in **1a**.
   (b) ***Invoked Method.*** (Figure 6e) Improves on **2a**, as while it still requires the modifications to include a

new argument to the invoking method, it parameterizes the invoked method, similar to **1b.**

(c) ***Invoked Args.*** (Figure 6f) Further improves on **2b**, as while it still requires the modifications to the invoking method, all other changes are made to the method on the invoked service, similar to **1c**.

DoorDash. DoorDash *has opted for **2c*** in existing application code. This maximizes code reuse of the invoked service while minimizing duplication in both the invoked and invoking services. As we will see in the following section, where we introduce an application bug and want to increase sensitivity this design happens to be one of most challenging.

5.1 Adding Circuit Breakers

In this section, we introduce an application bug to demonstrate the challenges of increasing sensitivity using our application designs from the previous section.

This bug we introduce only affects *one* type of order: takeout orders, when cancelled, return an error. However, the bug does not affect when orders are created or modified. This bug is localized in the auth service. In the examples where the auth service has been duplicated with two variants for each order type, we assume the bug only exists in the service for takeout orders. This application bug is inspired by an actual bug experienced by DoorDash where order cancellation was broken for all orders because of a bug affecting one particular order type.

As before, the methods that encapsulate the RPC invocations in Figure 6 are annotated with *method-explicit* circuit breakers. While this presentation is focused on this type, the issues discussed apply to both *callsite-* and *client-explicit*. This is consistent with our **Scope Partitioning** insight.

To start, we consider the case of **1a**. Example **1a**, presented in Figure 6a, duplicates both the invoking method and the method encapsulating the invoked RPC. Therefore, since the RPC's encapsulating method is only used for takeout order cancellations, a *method-explicit* circuit breaker work perfectly for disabling the malfunctioning method on the takeout auth service.

5.1.1 Path-Sensitivity. If we compare this to the rest of the designs, examples **1b** through **2c**, we see that all of these application designs suffer from the aforementioned problems of method indirection. Therefore, when these circuits open, they will disable invocations to *the cancellation for both takeout and delivery*. This reinforces our **Partitioning** insight. That is, to achieve precise method-based fault tolerance using method-explicit circuit breaking, the encapsulating methods must be duplicated for each method.

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

> ✓ **Path-sensitivity:** Circuit breakers aware of the invocation path, improve the sensitivity of circuit breaking.

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

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

> ✓ **Context-sensitivity:** Circuit breakers, aware of the invoking RPC's arguments, can further improve the sensitivity of circuit breaking.

6 IMPLICATIONS

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

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

Now, the developer wants to extend **A** with conditional functionality where **B** will invoke **D** instead of **C** depending on what arguments are provided to **A**. This can be seen as an abstraction of the example Section 5, Figure 6, Example 1a, where a new service is conditionally executed based on the arguments to earlier RPCs. Additionally, the application developer wants to use circuit breakers to ensure either the

failure of **C** or **D** are properly tolerated. For this example, we assume a single RPC method for both **C** and **D**.

The developer has to make several choices at this point. First, do they prefer *infrastructure-* or *application*-level circuit breakers? If they prefer application-level, do they prefer *transparent* or *explicit*? If they opt for explicit, do they prefer *client*, *method*, or *callsite* circuit breakers? Finally, will their selection provide the correct sensitivity?

To understand the implications of application design and circuit breaker choice on sensitivity, we present the following decision tree in Figure 7. Here, we explore the implications of one specific choice of circuit breaker, *callsite-transparent*, on *all possible application parameterizations* when adding this new feature. The callsite-transparent circuit breaker maximizes both transparency and scope, thereby minimizing developer overhead in application and providing the most graunlar scope: the callsite where the RPC invocation occurs. However, transparency implies that the application of the circuit breaker cannot automatically determine the RPC's service, method, or arguments from the in-scope variables, as this is generally undecidable. Therefore, it can only be aware of what has *already occurred* prior to the RPC invocation. This is generally true for all of the designs we observed, with the exception of the *client-explicit* design.

To understand the implications of the *client-transparent* choice, we start by looking at the implications of choosing a transparent infrastructure-level circuit breaker, as it forms the core component of an application-level circuit breaker.

(1) When a new service **E** is created to support the additional functionality of **D**, a infrastructure-level circuit breaker provides the necessary sensitivity for application bugs as invocations occur on a new service.
(2) Alternatively, when an existing service **B** is parameterized to support the conditional invocation of **D**, either path or context sensitivity is necessary to provide the correct sensitivity.

We note that any further parameterization of **A** to support the conditional invocation of either **C** or **D** has no effect on circuit breaker selection.

Recall that we did not observe any implementations of the *callsite-transparent* circuit breaker design: its an ideal design that combines all of the desirable properties of the implementations we did observe. Therefore, we use it as a reference point. In Figure 8, we present decision trees for the four circuit breaker designs that we observed and identified concrete implementations for: *callsite-explicit*, *method-explicit*, *1 client-explicit*, and *N client-explicit*. We highlight the differences in each diagram from the *callsite-transparent* design.

- *Callsite-explicit.* If a different circuit breaker is used and manually installed at each call site of an RPC, a developer can manually configure that circuit breaker accordingly so that it has the necessary sensitivity. This is by far the approach with the most overhead. It requires manually creating a circuit breaker for the proper sensitivity, manually incrementing the failure and success counters, and writing the appropriate conditionals to guard the invocation.
- *Method-explicit.* Path-sensitivity is needed to provide the correct sensitivity for invocations in shared methods and differ only by the RPC invocation path when the RPC service, method, and arguments are inaccessible — as they are when using decorators that guard method invocations. When the path differs only by arguments, context-sensitivity is required.
- *1 client-explicit.* The introduction of a different path early in the RPC invocation chain ensures that path-sensitivity can provide the correct sensitivity. This avoids the need for context-sensitivity.
- *N client-explicit.* If developers are willing to stomach the performance penalties and development overhead of using a new RPC client for each invocation, existing *client* circuit breaker designs provides the correct sensitivity, as they mimic the behavior of the *callsite-explicit*.

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

6.0.1 Armeria. One notable exception here is the Armeria circuit breaker. Ameria allows for the specification of circuit breaker rules that can be added to either a client or callsite circuit breaker, although this information could be provided manually for any callsite circuit breaker. These rules allow for partitioning of the circuit breaker state for the RPC invocation based on the invoked host, service, or method. However, these rules only get you so far.

For example, consider the scenario where both a Service W and a Service X invoke a method on Service Y, which invokes a single RPC method on Service Z before returning an answer to its caller. Now, only the invocations that originate from W — not X — cause exceptions in the call from Y to Z. In this case, partitioning the circuit breaker state by invoked host, service, or method is insufficient for the proper sensitivity to failures caused by the path originating at W.

6.1 Proposed Implementations

Both path- and context-sensitive circuit breakers require that the RPC invocation path is tracked and propagated across all RPC invocations for a single request issued by the end user.

To achieve this, we envision the use of a technique we have been developing separately as part of a larger initiative in fault injection testing at DoorDash [19]: *distributed execution indexing* (DEI) [37].

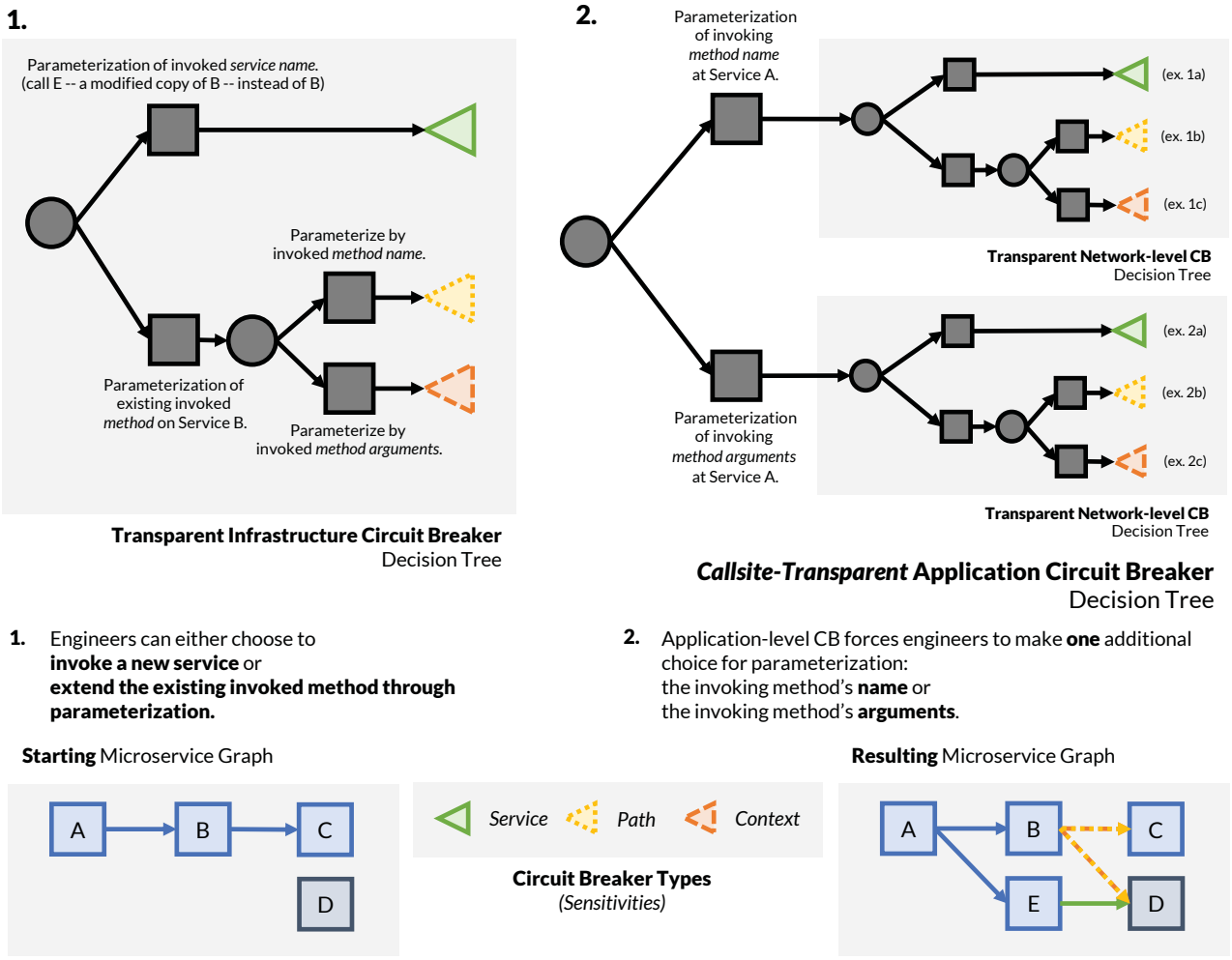

Figure 7: Decision tree relating abstraction choices to required circuit breaker sensitivites.

DEI is an algorithm for generating unique identifiers — called execution indexes — for individual RPC invocations that are tolerant to control flow changes, function indirection, data, and scheduling nondeterminism. These identifiers are both more abstract and more precise than existing techniques for assigning identifiers to RPCs. For example, by avoiding sensitivity to execution order under concurrent execution and by ensuring identifiers remain stable under branching control flow (*c.f.,* 3MileBeach [51].)

Our implementation of DEI integrates directly into the widely used open-source OpenTelemetry distributed tracing framework and can automatically instrument our microservices with execution indexes through the use of a runtime parameter that is supplied to the JVM. Therefore, our plan is to augment our existing Armeria circuit breaker rule – the per path rule that contains the invoked RPC service and method – with the inclusion of a DEI that will allow us to encode the RPC invocation path thereby providing path-sensitivity. We envision that a similar design would work for method-explicit circuit breakers as well.

Context-sensitivity may prove more problematic to implement in practice, as it needs to be aware of the arguments of the RPC invocation. While the DEI approach has the ability to include the arguments into the execution index that it generates for each RPC invocation, we believe that more often than not only a *subset* of the arguments should be included into these identifiers. For example, in the case of our second case study, we would most likely want to perform circuit breaking on order type, but perhaps not the user identifier or line item number, or order creation time for a takeout order, as the application bug most likely would not be dependent on these fields — but, may be. Therefore, for context-sensitivity

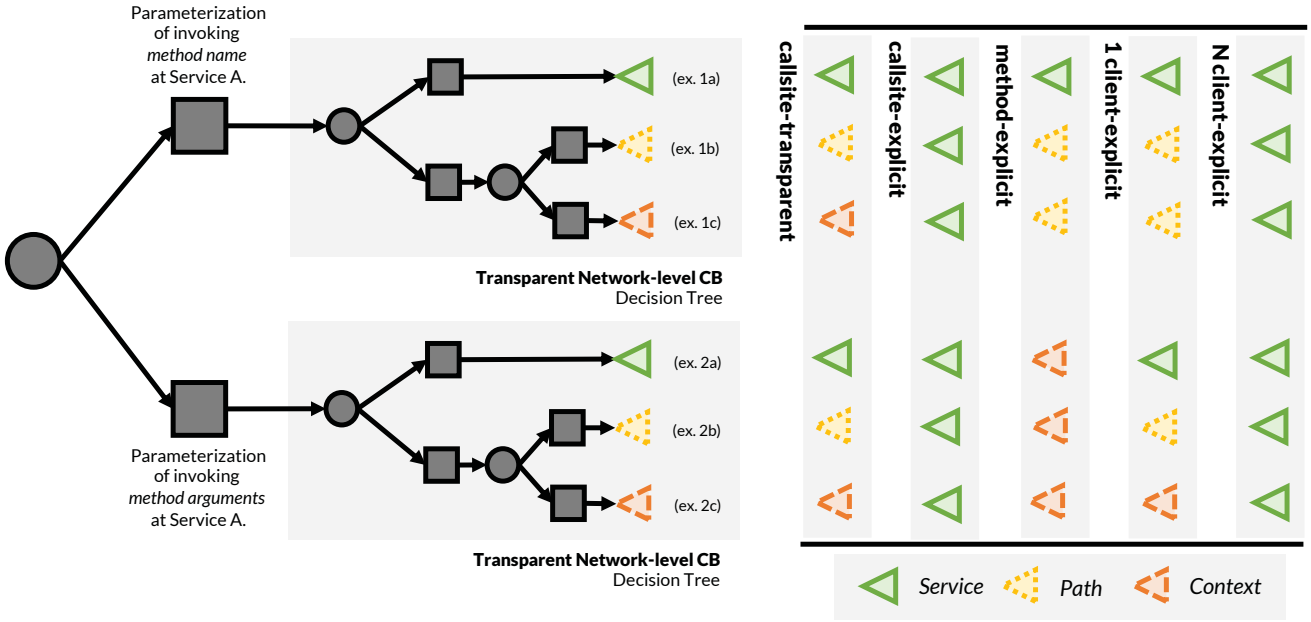

Figure 8: Decision trees determining circuit breaker sensitivity choice based on application structure.

to work in practice, we imagine that a mechanism would be required to determine the arguments that should be included. We envision that this could be achieved through the use of custom circuit breaker rules that indicate arguments that should be excluded when generating execution indexes installed at each RPC location where a RPC client is created.

Takeaways. We look at these two designs as complimentary and feel that there is need for both. Path-sensitivity meets the sweet spot of minimal developer overhead with proper sensitivity when the application is designed accordingly. Context-sensitivity is a more expensive approach but fulfills a need where an application has already been designed and must be retrofitted with circuit breakers to increase reliability until the code can be refactored. Therefore, we recommend developers avoid easier to implement application designs that require context-sensitivity over slightly more verbose designs that support path-sensitivity.

7 OPEN CHALLENGES

Circuit breakers remain an area of limited academic study where further research would be beneficial for addressing the challenges of correctly applying them. These challenges arise because proper placement and configuration of a circuit breaker is non-trivial. If used too coarsely, outside of the application, they may reduce reliability. If used too finely, inside of the application, they may misinterpret control flow or other application behavior as errors and also reduce reliability. Even further, the faults that the system experiences may be unpredictable and interact with circuit breakers in unexpected ways.

The two case studies presented in this paper were abstract descriptions of scenarios and failure modes that we have experienced at DoorDash. We have also experienced issues with infrastructure-level circuit breaking as well, where these circuit breakers have been scoped too broadly and reduced the overall reliability of the application when trying to react and respond to faults where a circuit breaker has disabled correctly functioning parts of our platform. But, failures are not always related only to scope. We demonstrate by providing two other examples that show how external actors can alter internal circuit breaker state and how aspects of the programming language itself — if properly handled — can make circuit breakers not function as expected.

Requests that originate outside of the platform can affect the circuit breakers that are used internally to the platform for reliability. In one example, a bot that was crawling our site was issuing malformed HTTP requests. These requests caused an internal service, several levels deep in the application graph, to return a HTTP 400 Bad Request response back to the service that took it as a direct dependency. The service that received that 400 Bad Request response converted that error into a 500 Internal Server Error when a generic error handler was hit, this response was then returned to its upstream service. This upstream service then counted

this error response against the RPCs error counters until the circuit breaker opened and disabled this path. Therefore, an outside user of the service was able to trigger an internal circuit breaker which caused an outage for other users of the service when the circuit breaker was tripped.

Circuit breakers may also be misconfigured with respect to aspects of the programming language. In one example, Service A does a wide fanout to Services B, C, D, and E using coroutines to issue the RPCs concurrently. Then, the circuit breaker between A and E opens, but E has not received any RPCs from A and therefore did not return any errors that would have counted against the circuit breaker between A and E. In this example the circuit breaker between A and E opened because Service B was immediately returning errors — and because of coroutine scoping where all coroutines were issued under a shared scope — the RPCs between A and C, D, and E were cancelled if not already complete. As cancellation is performed by raising an exception, a circuit breaker that was not configured to ignore cancellation exceptions counted them as errors for the circuit breaker between A and E.

Circuit breaker placement and configuration is also an area that merits further research. For example, what is the impact of failure detection speed when partitioning state across circuit breakers that are sensitive to a particular RPC method, host, or service? Does the placement of the circuit breaker (*e.g.,* callsite vs. method) impact what the threshold configurations should be for that circuit breaker? Does transparency impact how developers write or refactor code for proper sensitivity? How should a developer be alerted when a circuit breaker is scoped too coarsely?

At DoorDash, we have only started investigating these questions [19] by extending the Filibuster [36] fault injection tool with the ability to target a precise circuit breaker for mechanical verification. For example, does the circuit breaker open and then close based on a given threshold of faults, fault types, and a specific workload. However, this is only the start. The next steps are to extend this work to ensure that when a circuit breaker does open that it does not simultaneously affect the system in adverse ways. Understanding how to specify that in a way that can be mechanized and applied across our platform is still an open research question.

8 RELATED WORK

In terms of the "end-to-end argument" [45], circuit breakers address the reliability of individual services when application bugs occur in their dependent services. This aspect of fault tolerance is required even when the underlying cluster manager automatically handles crash failures and the RPC framework ensures reliable delivery. The first reference to the use of circuit breakers as a technique for improving the reliability of a microservice application is from Fowler & Nygard [29]. Hole [31] then identified them as a key technology required in building anti-fragile cloud applications and presented Netflix (and Hystrix [14]) as an exemplar.

Since then, several qualitative studies [22–24, 27, 32, 33, 35, 42, 44, 48, 50] have identified circuit breakers as a core pattern used to improve reliability in microservice applications. However, as noted by some researchers [47], circuit breakers remain an area of limited academic study. When research does exist, it is primarily focused on the configuration of circuit breakers: for example, the dynamic tuning of thresholds to improve availability [46], the use of model checking to determine the optimal configuration to achieve certain quality attributes [38], or the use of epidemic protocols for circuit breaker state dissemination to improve the speed of failure detection [43].

Falahah *et al.* [28] performed a systematic mapping study of 23 articles on circuit breakers, in isolation, to create a conceptual model. Our taxonomy differs in two ways:

(1) They consider two different types of implementation *strategies:* library, which they use for application-level circuit breakers, and proxy, which they use for infrastructure-level circuit breakers. We use the latter, more descriptive terminology — as most software is provided as a library — to indicate whether the application has visibility into the presence (or absence) of a circuit breaker.
(2) They use the property *distribution* to indicate the coarseness of a circuit breaker: per service, per host, or per method. We refer to this property as *sensitivity.*

Academic work has touched on the topic of circuit breaker testing. Heorhiadi *et al.* [30] proposed a fault injection tool, Gremlin, that can verify the correct operation of a circuit breaker given a specification of precisely what RPCs should be prevented when in the open state. Meiklejohn *et al.* [37] proposed a strategy for identifying when circuit breakers were open using a novel indexing scheme for RPCs.

9 CONCLUSION

In this paper, we examined the use of circuit breakers to understand how application design influenced the effectiveness of circuit breakers for fault tolerance. To do this, we used two case studies. These case studies were inspired by both industrial use cases and outages experienced at DoorDash, a large food delivery platform.

We determined that not only are existing circuit breaker designs insufficient for fault tolerance, but identified how small common abstraction changes in application code can drastically alter how circuit breakers work. To address these deficiencies, we proposed two new designs for circuit breakers and envisioned how they could be implemented today.

REFERENCES

[1] 2021. Audible. https://www.audible.com. Accessed: 2021-05-21.

[2] 2022. Building a gRPC Client Standard with Open Source to Boost Reliability and Velocity. https://doordash.engineering/2021/01/12/building-a-grpc-client-standard-with-open-source/. Accessed: 2022-06-05.

[3] 2022. Circuit breaker — Akka documentation. https://doc.akka.io/docs/akka/current/common/circuitbreaker.html. Accessed: 2022-06-05.

[4] 2022. Circuit breaker — Armeria documentation. https://armeria.dev/docs/client-circuit-breaker/. Accessed: 2022-06-05.

[5] 2022. Circuit Breaking with Envoy. https://blog.turbinelabs.io/circuit-breaking-da855a96a61d. Accessed: 2022-06-05.

[6] 2022. 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: 2022-06-05.

[7] 2022. GitHub: App-vNext/Polly. https://github.com/Comcast/jrugged. Accessed: 2022-06-05.

[8] 2022. GitHub: Comcast/jrugged. https://github.com/Comcast/jrugged. Accessed: 2022-06-05.

[9] 2022. GitHub: danielfm/pybreaker. https://github.com/danielfm/pybreaker. Accessed: 2022-06-05.

[10] 2022. GitHub: envoyproxy/envoy. https://github.com/envoyproxy/envoy. Accessed: 2022-06-05.

[11] 2022. GitHub: fabfuel/circuitbreaker. https://github.com/fabfuel/circuitbreaker. Accessed: 2022-06-05.

[12] 2022. GitHub: filibuster-testing/filibuster-corpus. https://github.com/filibuster-testing/filibuster-corpus. Accessed: 2022-09-20.

[13] 2022. GitHub: kubernetes/kubernetes. https://github.com/kubernetes/kubernetes. Accessed: 2022-06-05.

[14] 2022. GitHub: Netflix/Hystrix. https://github.com/Netflix/Hystrix. Accessed: 2022-06-05.

[15] 2022. GitHub: rubyist/circuitbreaker. https://github.com/rubyist/circuitbreaker. Accessed: 2022-06-05.

[16] 2022. Hystrix : How to implement fallback and circuit breaker. https://medium.com/@kullik2/hystrix-how-to-e41cabf34d40. Accessed: 2022-06-05.

[17] 2022. Implementing a Circuit Breaker with Resilience4j. https://reflectoring.io/circuitbreaker-with-resilience4j/. Accessed: 2022-06-05.

[18] 2022. resilience4j. https://resilience4j.readme.io/docs/examples. Accessed: 2022-06-05.

[19] 2022. Using Fault Injection Testing to Improve DoorDash Reliability . https://doordash.engineering/2022/04/25/using-fault-injection-testing-to-improve-doordash-reliability/. Accessed: 2022-06-05.

[20] 2022. Wikipedia: Circuit breaker. https://en.wikipedia.org/wiki/Circuit_breaker. Accessed: 2022-06-05.

[21] 2022. Wikipedia: Rolling blackout. https://en.wikipedia.org/wiki/Rolling_blackout. Accessed: 2022-06-05.

[22] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51. https://doi.org/10.1109/SOCA.2016.15

[23] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33, 3 (2016), 42–52. https://doi.org/10.1109/MS.2016.64

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

[25] Huamin Chen and P. Mohapatra. 2002. Session-based overload control in QoS-aware Web servers. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2. 516–524 vol.2. https://doi.org/10.1109/INFCOM.2002.1019296

[26] L. Cherkasova and P. Phaal. 2002. Session-based admission control: a mechanism for peak load management of commercial Web sites. *IEEE Trans. Comput.* 51, 6 (2002), 669–685. https://doi.org/10.1109/TC.2002.1009151

[27] Cleber Jorge Lira de Santana, Brenno de Mello Alencar, and Cássio V. Serafim Prazeres. 2019. Reactive Microservices for the Internet of Things: A Case Study in Fog Computing. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 1243–1251. https://doi.org/10.1145/3297280.3297402

[28] Falahah, Kridanto Surendro, and Wikan Danar Sunindyo. 2021. Circuit Breaker in Microservices: State of the Art and Future Prospects. *IOP Conference Series: Materials Science and Engineering* 1077, 1 (feb 2021), 012065. https://doi.org/10.1088/1757-899x/1077/1/012065

[29] Martin Fowler. 2014. Circuit Breaker. https://martinfowler.com/bliki/CircuitBreaker.html.

[30] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 57–66. https://doi.org/10.1109/ICDCS.2016.11

[31] Kjell Jørgen Hole. 2016. *Anti-fragile Cloud Solutions*. Springer International Publishing, Cham, 47–56. https://doi.org/10.1007/978-3-319-30070-2_5

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

[33] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. 2018. Challenges When Moving from Monolith to Microservice Architecture. In *Current Trends in Web Engineering*, Irene Garrigós and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 32–47.

[34] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. *MartinFowler. com* 25 (2014), 14–26.

[35] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. 2021. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology* 131 (2021), 106449. https://doi.org/10.1016/j.infsof.2020.106449

[36] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 388–402. https://doi.org/10.1145/3472883.3487005

[37] Christopher S. Meiklejohn, Rohan Padhye, and Heather Miller. 2022. Distributed Execution Indexing. arXiv:2209.08740 [cs.DC]

[38] Nabor C. Mendonca, Carlos M. Aderaldo, Javier Camara, and David Garlan. 2020. Model-Based Analysis of Microservice Resiliency Patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)*. 114–124. https://doi.org/10.1109/ICSA47634.2020.00019

[39] Pieter J. Meulenhoff, Dennis R. Ostendorf, Miroslav Živković, Hendrik B. Meeuwissen, and Bart M. M. Gijsen. 2009. Intelligent Overload Control for Composite Web Services. In *Service-Oriented Computing*, Luciano Baresi, Chi-Hung Chi, and Jun Suzuki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–49.

[40] Fabrizio Montesi and Janine Weber. 2016. Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830* (2016).

[41] Fabrizio Montesi and Janine Weber. 2018. From the Decorator Pattern to Circuit Breakers in Microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1733–1735. https://doi.org/10.1145/3167132.3167427

[42] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. 2020. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2020), 3–15. https://doi.org/10.1007/s00450-019-00407-8

[43] Aashay Palliwar and Srinivas Pinisetty. 2022. Using Gossip Enabled Distributed Circuit Breaking for Improving Resiliency of Distributed Systems. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 13–23. https://doi.org/10.1109/ICSA53651.2022.00010

[44] Dewmini Premarathna and Asanka Pathirana. 2021. Theoretical framework to address the challenges in Microservice Architecture. In *2021 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, Vol. 4. IEEE, 195–202.

[45] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (nov 1984), 277–288. https://doi.org/10.1145/357401.357402

[46] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. 2021. 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* (Online, United Kingdom) *(HAOC '21)*. Association for Computing Machinery, New York, NY, USA, 4–10. https://doi.org/10.1145/3447851.3458740

[47] Kridanto Surendro, Wikan Danar Sunindyo, et al. 2021. Circuit Breaker in Microservices: State of the Art and Future Prospects. In *IOP Conference Series: Materials Science and Engineering*, Vol. 1077. IOP Publishing, 012065.

[48] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. 2020. On the Study of Microservices Antipatterns: A Catalog Proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020* (Virtual Event, Germany) *(EuroPLoP '20)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. https://doi.org/10.1145/3424771.3424812

[49] H. Tucker, L. Hochstein, N. Jones, A. Basiri, and C. Rosenthal. 2018. The Business Case for Chaos Engineering. *IEEE Cloud Computing* 5, 03 (may 2018), 45–54. https://doi.org/10.1109/MCC.2018.032591616

[50] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. O. Ocharán-Hernández. 2020. Patterns Related to Microservice Architecture: a Multivocal Literature Review. *Programming and Computer Software* 46, 8 (2020), 594–608. https://doi.org/10.1134/S0361768820080253

[51] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3MileBeach: A Tracer with Teeth. In *Proceedings of the ACM Symposium on Cloud Computing*. 458–472.

[52] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/3267809.3267823