Method Overloading the Circuit

Christopher S. Meiklejohn
Ph.D. Candidate, Software Engineering, SCS/S3D

Lydia Stark
University of Alaska, Anchorage

Cesare Celozzi, Matt Ranney
DoorDash, Inc.

Heather Miller
Assistant Professor, SCS/S3D CMU

Microservice Architectures are Complex

Microservice architecture is an architectural style where applications are constructed from services that communicate over the network using RPC and are developed, scaled and deployed independently.

NETFLIX

1,000 services
(2021)

UBER

2,200 services
>120 for getting ride
(2016)

DOORDASH

500 services
>100 involved in core flow
(2022)

As of 2021, **all 50 companies in the Fortune 50** were hiring for roles that **mentioned microservices.** *[SoCC '21]*

Microservice applications are the **most common and complex** type of distributed application being built today.

Twitter (2017) operates a > 10k node distributed Hadoop cluster.
However, **most nodes have the same behavior, running the exact same code.**

DoorDash (2022) operates 500 microservices.
Each service provides **different functionality, has a different API, and is deployed continuously.**

Microservice Application Example

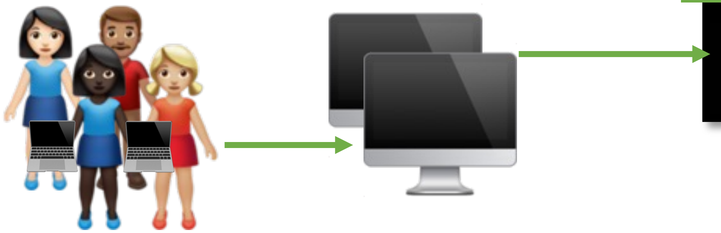
Why microservice architectures?
Improves **developer productivity** (*e.g.*, Fowler '14, DoorDash '20) and **application scalability.**

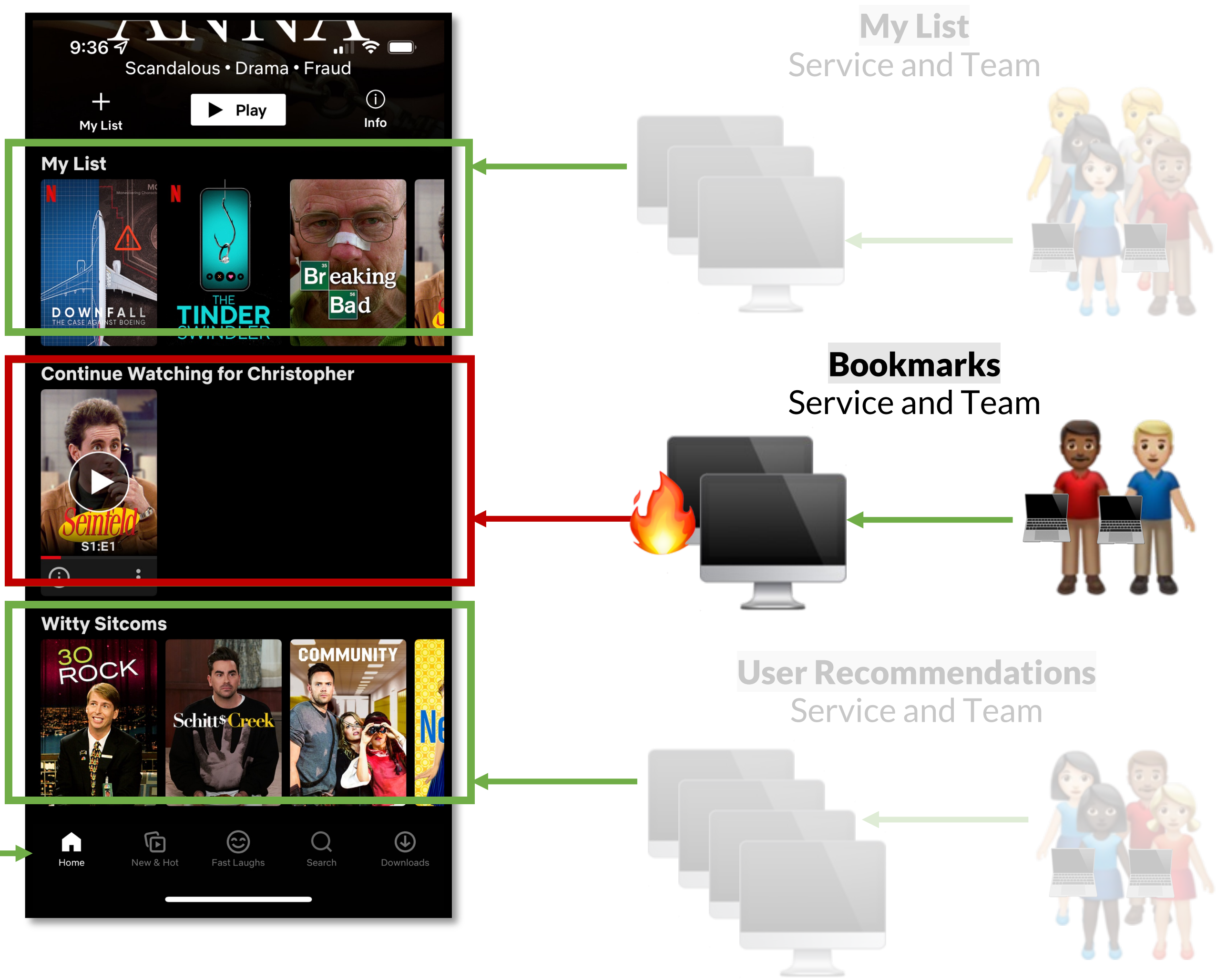
Trade-off of technical complexity.
Reduces whole application knowledge but forces developers to *consider* **partial failure.**

What happens if the bookmarks service is **unreachable** or **producing errors?**

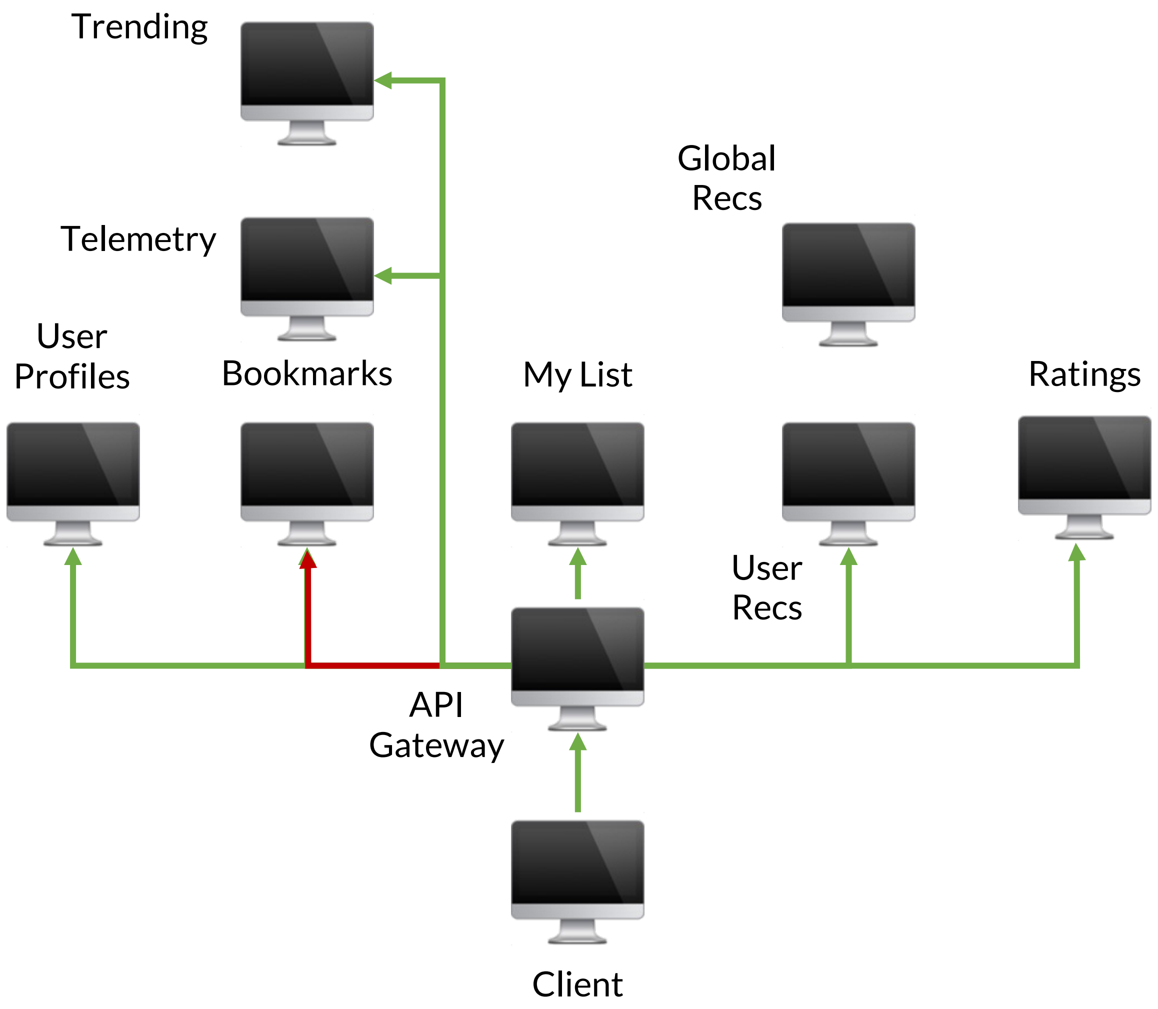
What ***should*** happen if the bookmarks service is **unreachable** or **producing errors?**

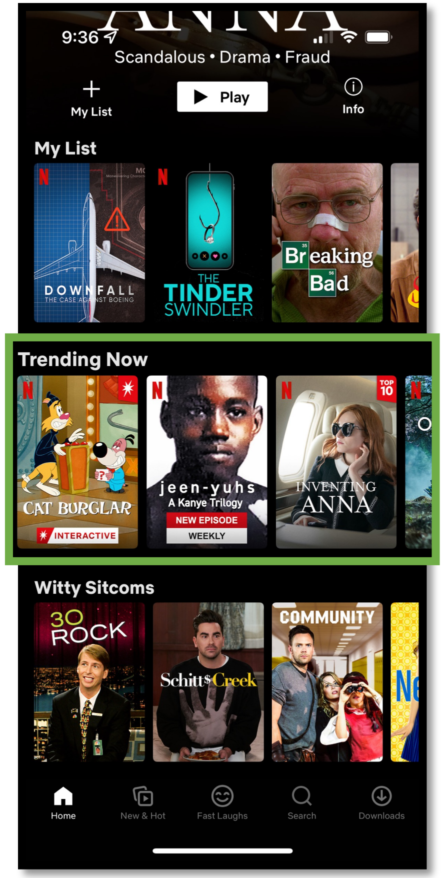
API Gateway
Service and Team





What *should*, and what *does*, happen?





Fallbacks
Developers specify **alternative application logic** in the event of dependency failure.

Other resilience techniques:

1	**Retries**
2	**Timeout**
3	**Load shedding**
4	**Circuit breakers**

Fault injection and chaos engineering used to **verify what *should* happen *does* happen.**
[Meiklejohn et al. 2021, SoCC '21]

Reliability at DoorDash

1. **Fallbacks**
   When dependencies are unavailable, load alternative content from different services or use default responses to allow application to degrade gracefully.
   (e.g., personalized recommendations become generic recommendations.)

2. **Cluster Orchestration**
   Support for rolling deploys with replicas of services supported by load balancing.
   Combined with single retries *(not timeout)*, lets nodes to hit non-failed replica on retry.
   Automatic readiness and liveness checks with auto-scaling and restart policies.

3. **Load Shedding**
   Short-circuit request at the **callee** using a predefined error indicating overload.
   Typically performed based on the number of outstanding concurrent requests.

4. **Circuit Breakers**
   Short-circuit request at the **caller** using a predefined error indicating failure condition.
   Typically performed based on the number of observed errors within a specific period.

Why Do Circuit Breakers Matter?

Microservice fault tolerance **is more complicated**.

Engineers **must also consider:**

1. **Bad deployments of a service.**
   Number of nodes return error responses *(e.g., 500 Internal Server Error)* before removal.
2. **Service failures only with certain arguments due to application bug.**
   Service returns errors when provided with certain arguments by a caller only. *(e.g., NPE, etc.)*
3. **Dependencies of a given RPC method may be malfunctioning.**
   Direct dependencies of a service may slow down, timeout, or fail in other ways.

Circuit breakers are an **important part of the resilience strategy at DoorDash.**

Why Study Circuit Breakers?

Circuit breakers often **weaken the resilience of the application** by disabling unrelated RPCs.

Extremely limited research in academia on circuit breaker design or usage exists.

1. **Taxonomy**
   Understand circuit breaker usage in order to determine if the errors we were experiencing were specific to our usage of a circuit breaker or inherent in circuit breaker design.

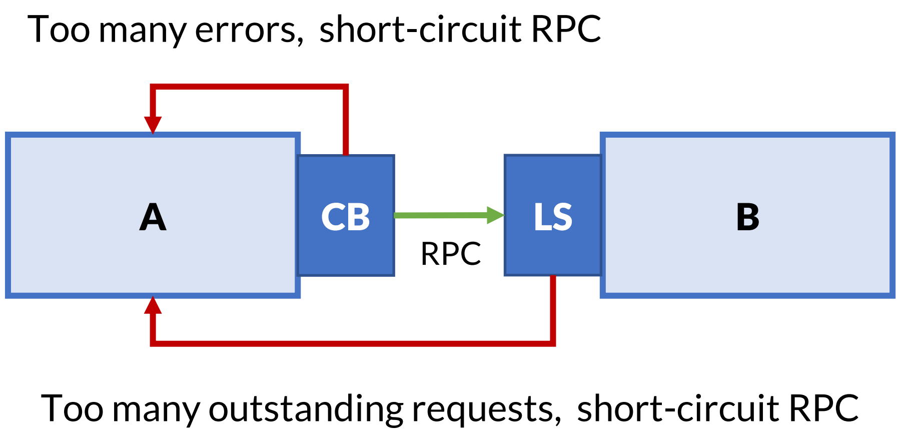
2. **Multiple Case Study Analysis**
   Identify inverse relationship between abstraction and circuit breaker usage through multiple case studies, implemented and open-sourced in the Filibuster application corpus. *[SoCC '21]*

3. **Proposed Designs**
   Propose new designs to address the deviancies in existing circuit breaker designs and discuss how they might be implemented.

Circuit Breakers: Overview

Circuit Breakers
Interpose on RPCs between services and record successes/errors to determine if RPC should be allowed.
With on a min threshold of requests and a sliding window, determine if the num of errors have exceeded a threshold.

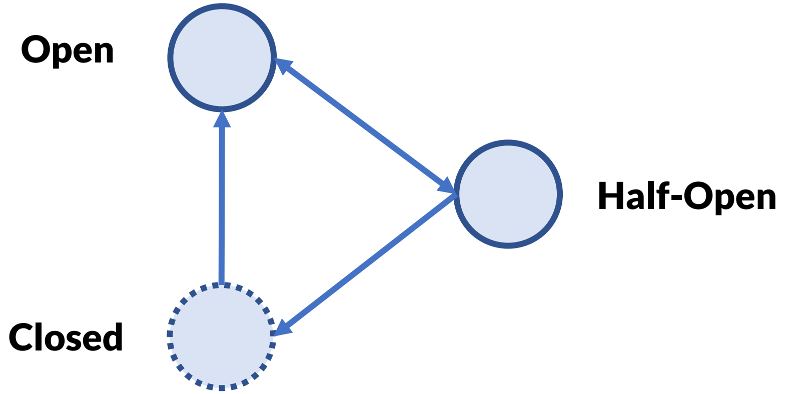
Load Shedding
Special case of circuit breakers that use number of outstanding requests at a given service.



Transitions

1. Circuits begin in the **closed** state.
   When the threshold is exceeded move to the **open** state where all RPCs are refused.

2. *Circuits move to the* ***half-open*** *state to determine if they should move to* ***open*** *if a subset of RPCs succeed.*



Circuit Breakers: Taxonomy

From a **software engineering perspective**, we were concerned with the following properties:

1. **Transparency** *(explicit vs. transparent)*
   Circuit breakers may require that developers **integrate them directly into the application code** or inherit them from the **libraries or infrastructure** they use.

2. **Scope**
   Circuit breakers may be **installed in the network**, at the **clients** of RPC invocations, on **methods** that invoke RPCs, or directly at the **call site** of an invocation in the application.

From this study, we discovered a **third property:**

3. ***Sensitivity***
   How the state of the circuit breaker state (*e.g.*, counters, etc.) is affected.
   This is typically inherited from the scope of the circuit breaker, but not always.

For a full discussion of these properties, see our paper.

Partitioning and Scope Partitioning

Before refactoring:

```
@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])
```

After refactoring:

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


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


@orders.method("delete")
def order_cancellation(order_id : String):
    res = issue_a


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

method-explicit circuit breaker

Partitioning and Scope Partitioning

After refactoring for circuit breaker sensitivity:

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

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

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

After refactoring:

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

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

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

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

Insight #1: **Partitioning**

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

Partitioning and Scope Partitioning

After refactoring for circuit breaker sensitivity:

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

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

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

Insight #1: **Partitioning**

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

Insight #2: **Scope Partitioning**

When partitioning to increase sensitivity, **partitioning must be performed with respect to the scope of the circuit breaker.**

Expanding our Application

Expand our application from only delivery **to delivery and takeout:**

Multiple (6 = 2 * 3) **possible order type parameterizations** (showing 3 representative examples):

1a.

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


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

*parametrization by **method** and **invoked service***

1b.

```
@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)
```

*parametrization by **method** and **invoked method***

2c.

```
@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)
```

*parametrization by **args** and **invoked args** (chosen by DoorDash engineers.)*

Path- and Context-Sensitivity

What happens if a bug **only affects cancellation of takeout orders?**

We must make the circuit breaker **sensitive to order type despite the parameterization choice?**

1b. *parametrization by **method** and **invoked method***

```
@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)
```

RPC invoking method (with CB) **shared for takeout and delivery.**

2c. *parametrization by **args** and **invoked args***

```
@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)
```

All methods **shared for takeout and delivery.**

Insight #3: **Path-sensitivity**

Circuit breakers are **aware of the RPC's invocation path**.

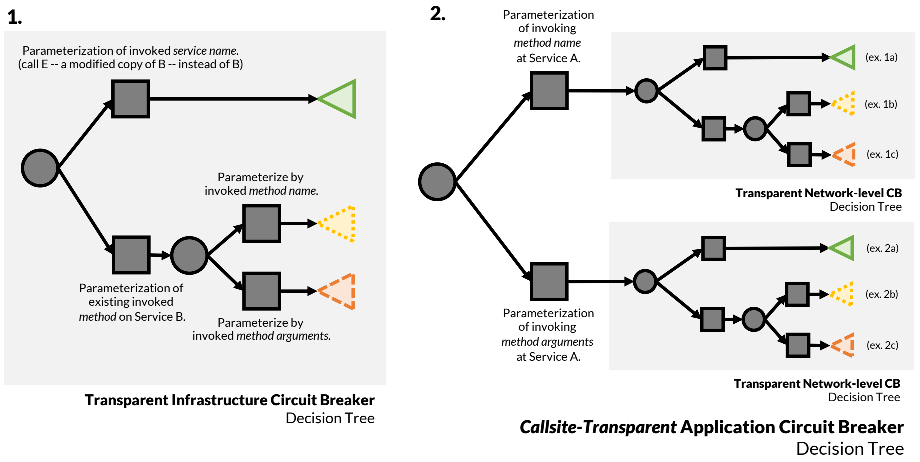
Insight #4: **Context-sensitivity**

Circuit breakers are **aware of the invoking RPC's arguments.**

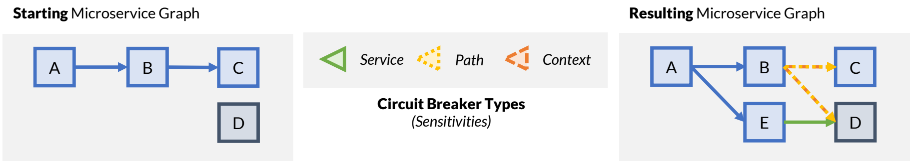
Decision Diagrams

Not meant **to be read or understood!**

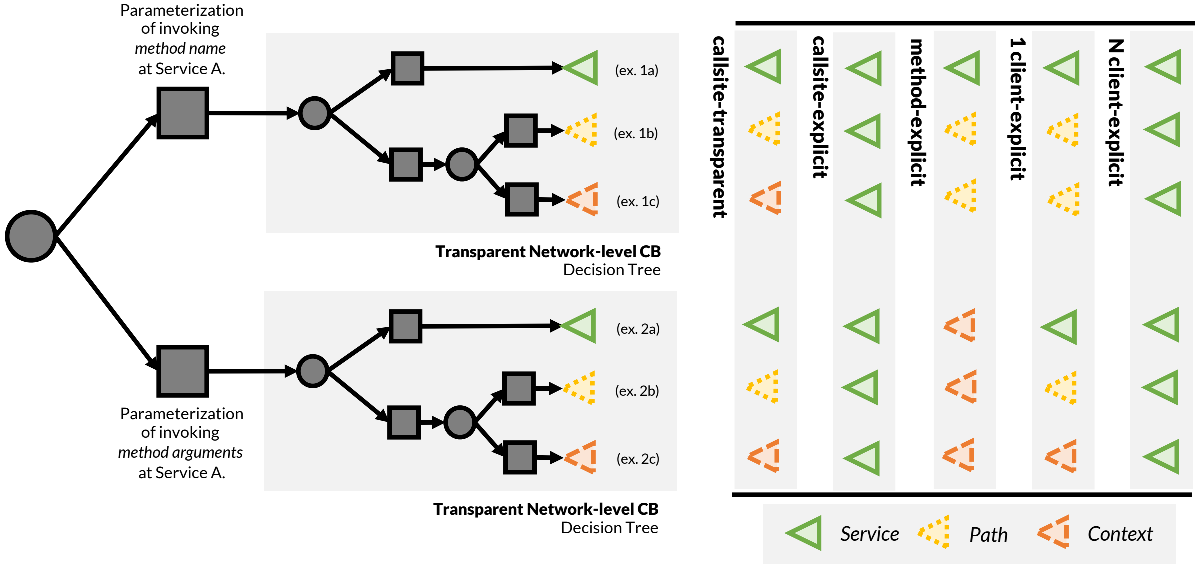
Design choices for a **single circuit breaker scope.**



1. Engineers can either choose to **invoke a new service** or **extend the existing invoked method through parameterization.**
2. Application-level CB forces engineers to make **one** additional choice for parameterization: the invoking method's **name** or the invoking method's **arguments**.



Possible design choices when considering **multiple circuit breaker scopes.**

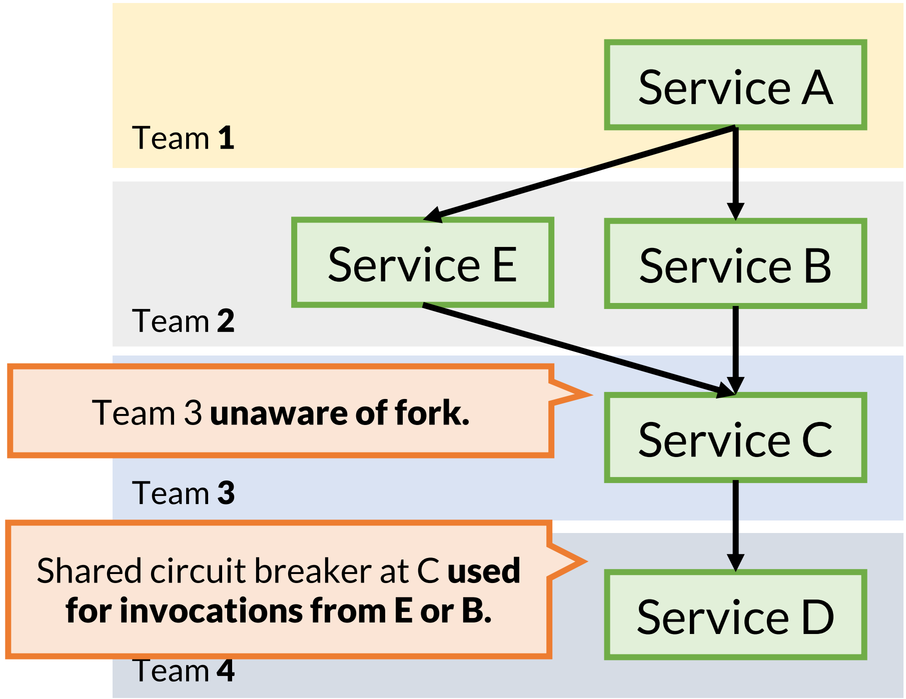


In almost all cases, achieving the correct sensitivity **requires circuit breakers that do not exist yet.**

Exacerbated by Microservices

Path-sensitivity.

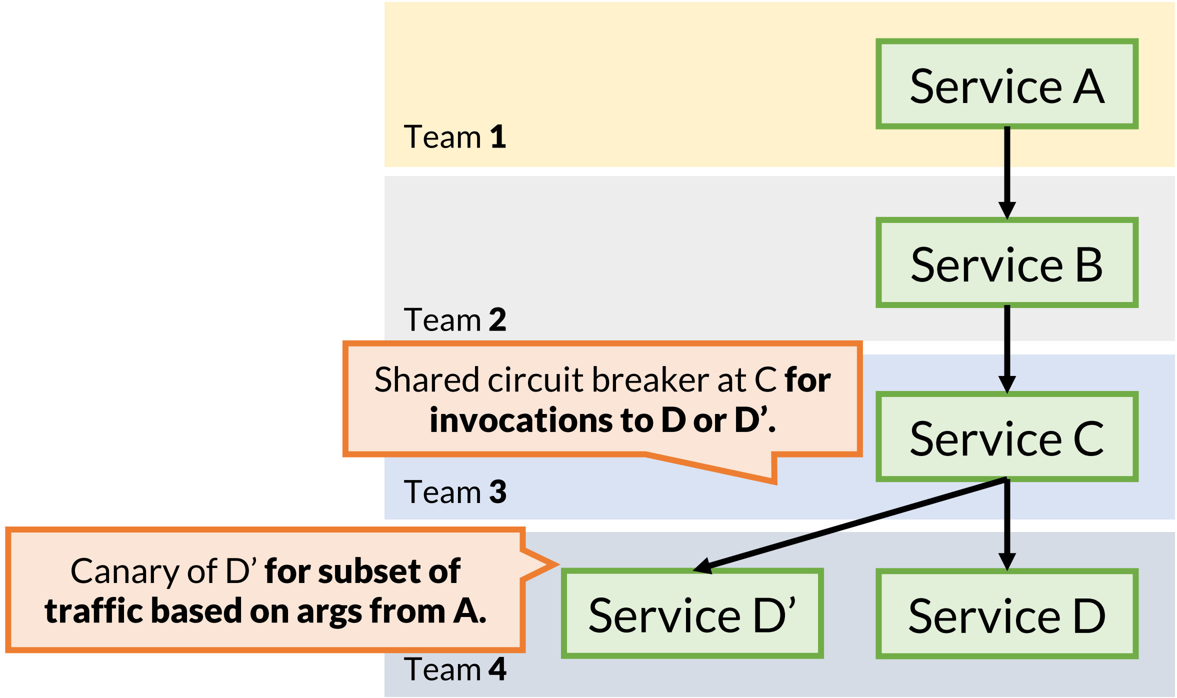
Example: *Downstream dependencies impacted by upstream non-local feature development.*



Problematic **if requests originating at E trigger fault in D.**

Context-sensitivity.

Example: *Canary release of D, D', contains a bug only for certain requests from A.*



Problematic **if D' is returning errors because of active fault.**

Without sensitivity: must refactor into **2 RPC invoking methods** *(method CB)* or to use **2 RPC clients** *(client CB.)*

Contributions

For more, **read our paper:**

1. **Taxonomy**
   Full discussion of process of identifying and classifying existing CB implemtations.
2. **Case Study #1 and Case Study #2**
   Including full discussion and implementation in the Filibuster corpus. *[SoCC '21]*
3. **Decision Process**
   Decision diagrams with walkthrough of extending a example application with CBs.
4. **Proposed Implementation**
   Discussion of implementation strategy for providing path- and context-sensitivity.
   Favor path-sensitive compatible app designs; context- only for retrofitting resilience.
5. **Open Challenges**
   Discussion of open research challenges based on our survey of circuit breakers and experience of using them at scale at DoorDash.

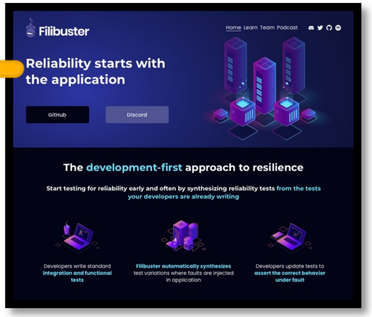
Conclusion

Microservice architectures **solve a socio-technical problem** designed to facilitate organization growth and come with a new set of **fault tolerance challenges.**

Application developers **increasingly rely on circuit breakers** as a fault tolerance mechanism against bad deployments, buggy code, and service unavailability.

However, the **current designs of circuit breakers pose problems** with the way application developers want to write application code.

Learn more about **microservice resilience**
http://filibuster.cloud



Students


Andrea Estrada


Yiwen Song


Lydia Stark (Anchorage)


Eunice Chen


Haoyang Wu

PhD Committee


Rohan Padhye


Claire Le Goues


Peter Alvaro (UCSC)

Advisor


Heather Miller

DoorDash


Matt Ranney


Cesare Celozzi