Designing Services for Resilience Experiments: Lessons from Netflix

Nora Jones, Senior Chaos Engineer
@nora_js

NETFLIX

Trending Now

Popular on Netflix

Because you watched Stranger Things

Continue Watching for Pamela

Trending Now

Popular on Netflix

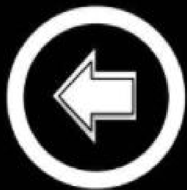
Because you watched Stranger Things

Continue Watching for Pamela

Trending Now

Popular on Netflix

Because you watched Stranger Things



Whoops, something went wrong…

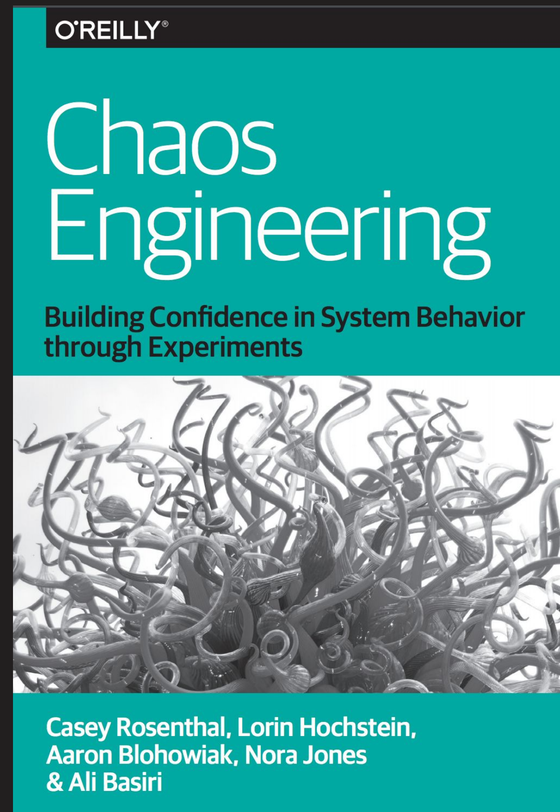
Netflix Streaming Error

We're having trouble playing this title right now. Please try again later or select a different title.

NETFLIX

What kind of upside down
world am I living in?

Designing Services for Resilience Experiments: Lessons from Netflix



Nora Jones, Senior Chaos Engineer
@nora_js

NETFLIX

So, how can teams design services for resilience testing?

- Failure Injection Enabled

So, how can teams design services for resilience testing?

- Failure Injection Enabled
- RPC enabled

So, how can teams design services for resilience testing?

- Failure Injection Enabled
- RPC enabled
- Fallback Paths
  - And ways to discover them

So, how can teams design services for resilience testing?

- Failure Injection Enabled
- RPC enabled
- Fallback Paths
  - And ways to discover them
- Proper monitoring
  - Key business metrics to look for

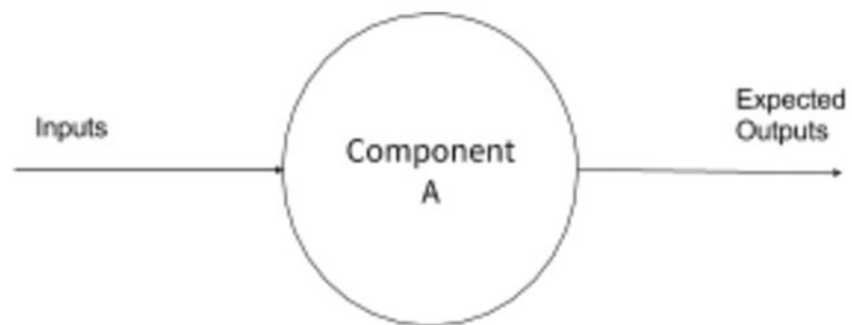
So, how can teams design services for resilience testing?

- Failure Injection Enabled
- RPC enabled
- Fallback Paths
  - And ways to discover them
- Proper monitoring
  - Key business metrics to look for
- Proper timeouts
  - And ways to discover them

Known Ways to Increase
Confidence in Resilience

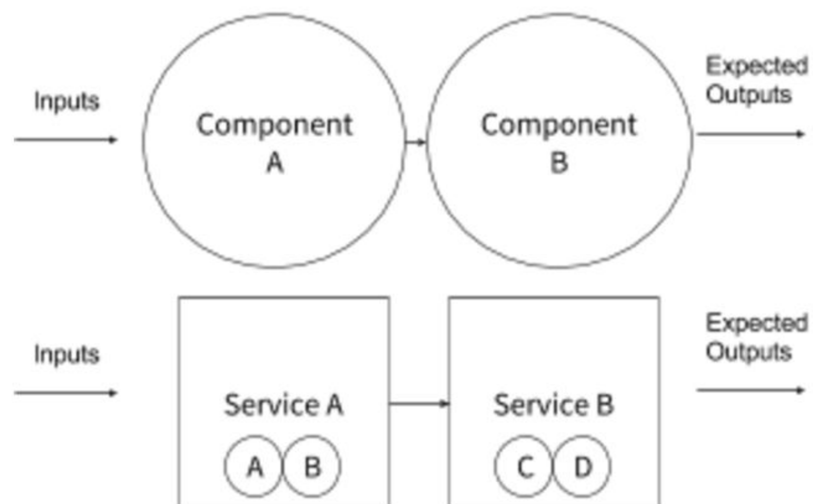
Known Ways to Increase Confidence in Resilience

- Unit Tests


Inputs
Component
A
Expected
Outputs
Unit Testing

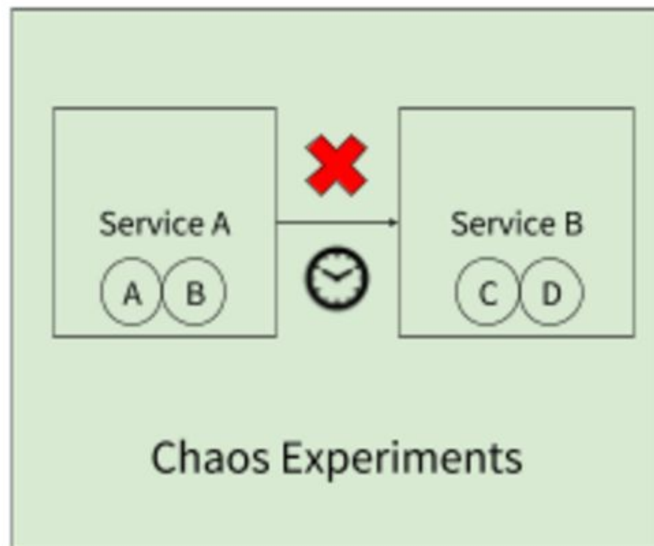
Known Ways to Increase Confidence in Resilience

- Integration Tests

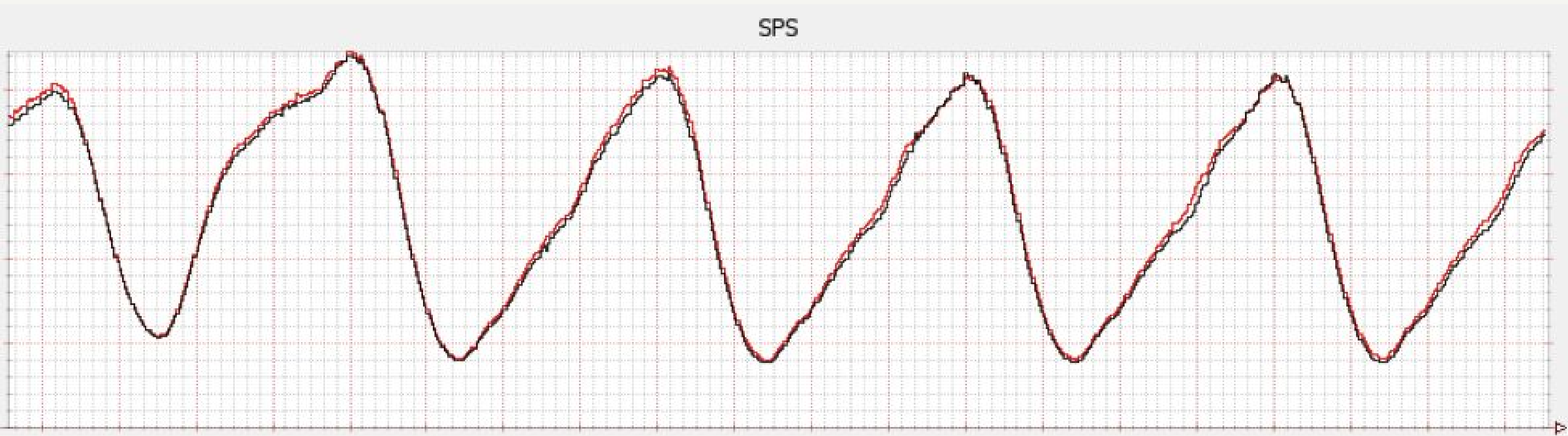

Inputs
Component A
Component B
Expected Outputs
Inputs
Service A
A
B
Service B
C
D
Expected Outputs
Integration Testing

New Ways to Increase Confidence in Resilience

- Chaos Experiments

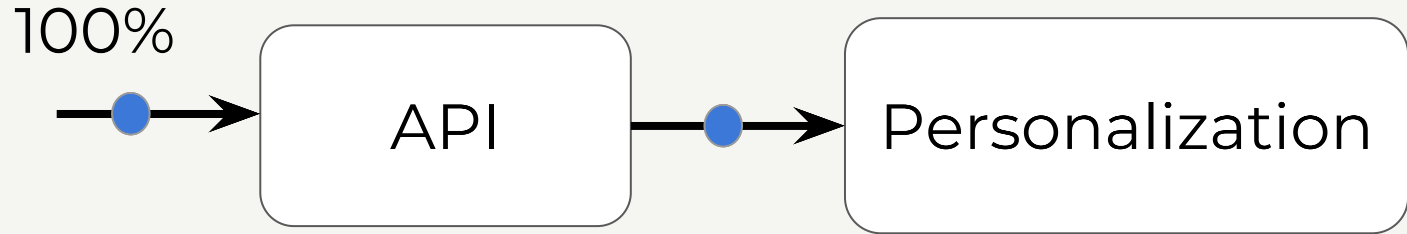

Service A
A
B
Service B
C
D
Chaos Experiments

SPS: Key Business Metric

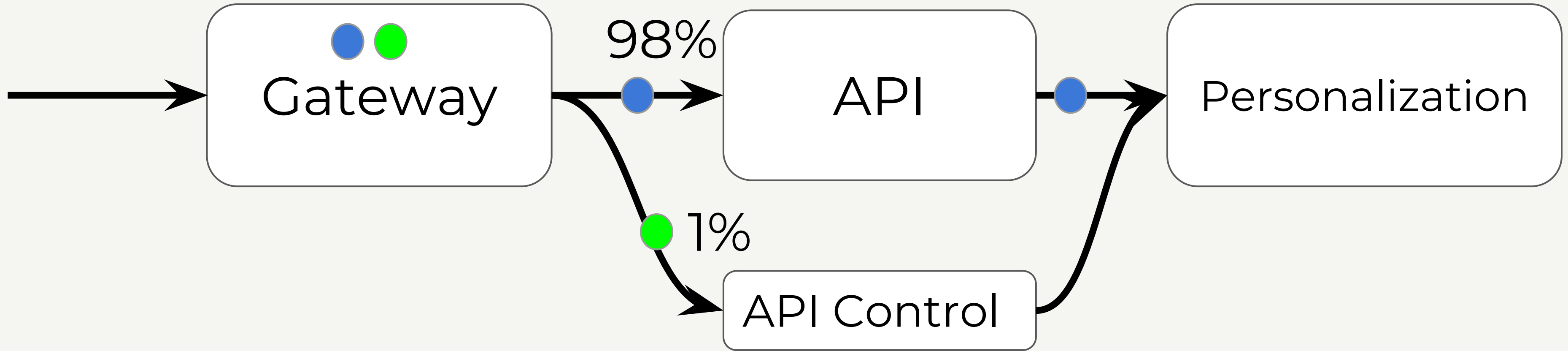


ChAP

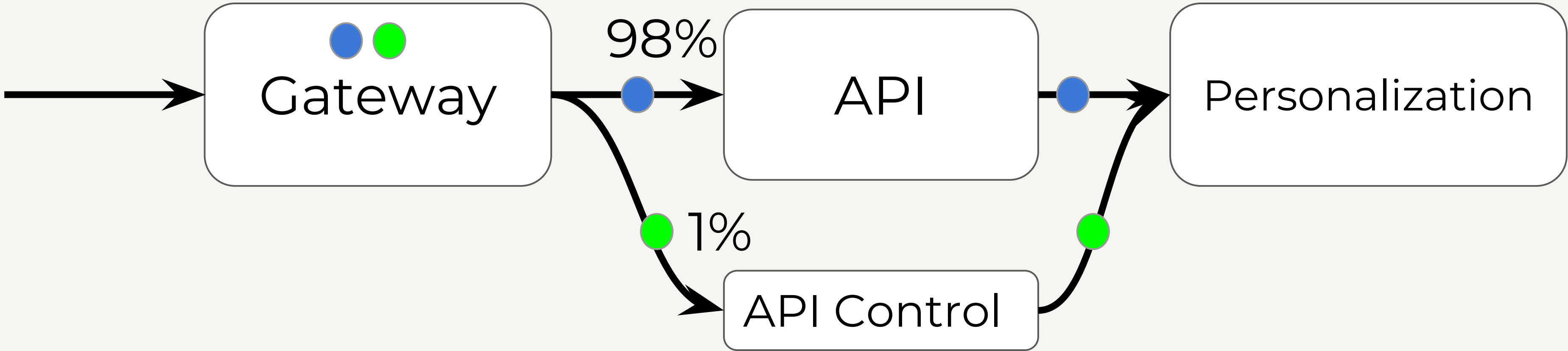
Chaos Engineering: Netflix’s ChAP



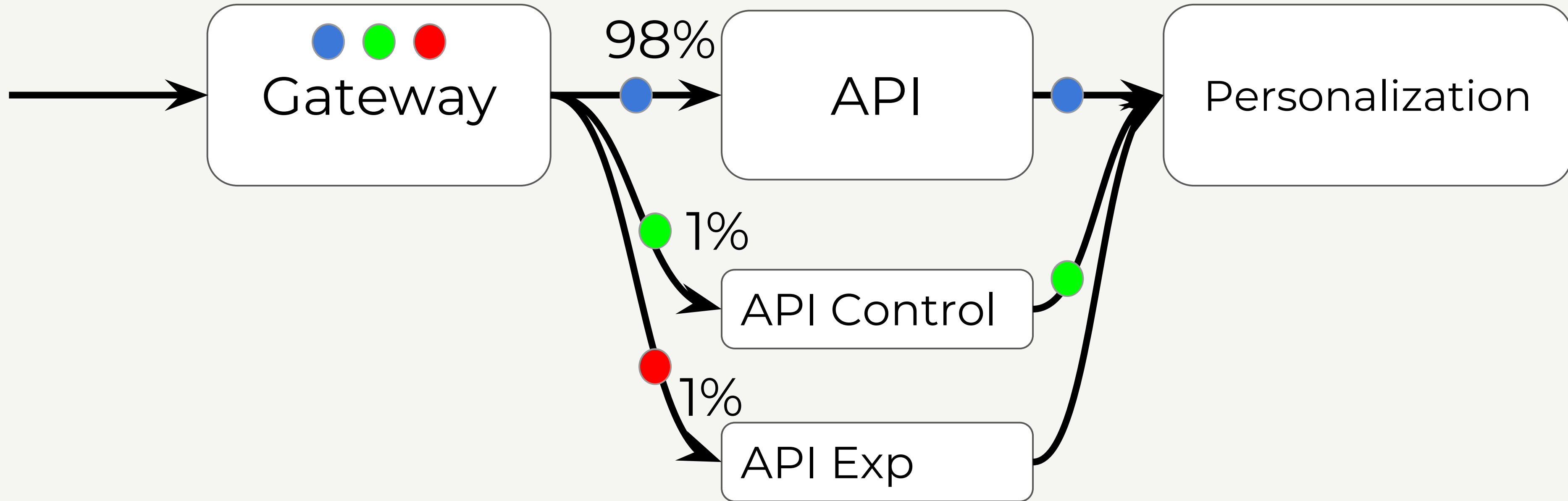
Chaos Engineering: Netflix's ChAP



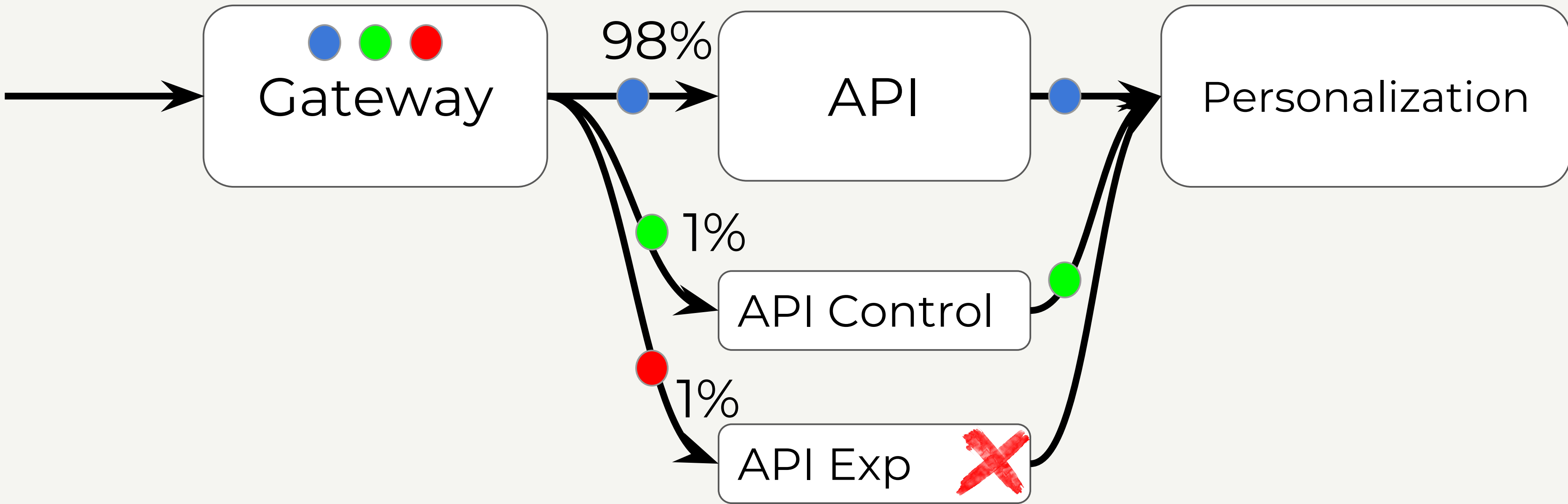
Chaos Engineering: Netflix's ChAP



Chaos Engineering: Netflix's ChAP

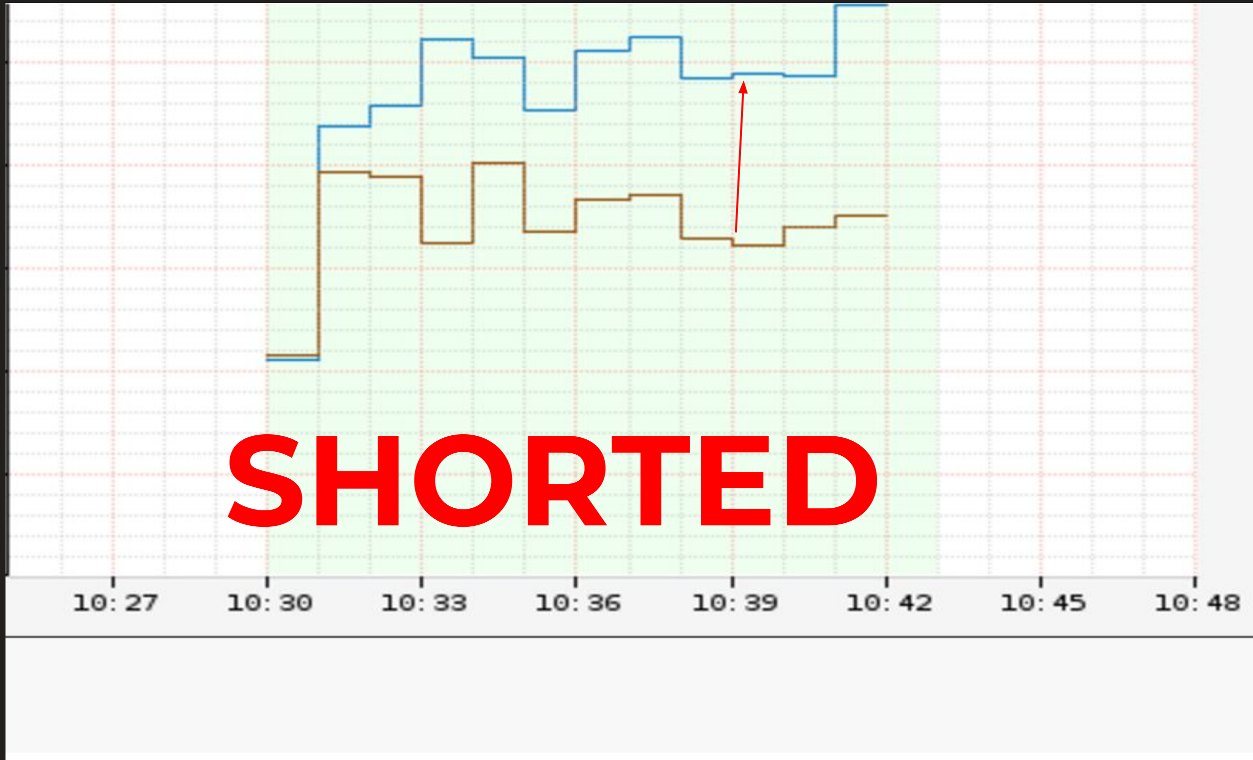


Chaos Engineering: Netflix's ChAP



Monitoring

Monitoring



1. Have Failure Injection
Testing Enabled.

Sample Failure Injection Library

https://github.com/norajones/FailureInjectionLibrary

```
let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
    fun (service:AsyncArrow<_,_>) req -> async {
        if shouldChaos() then
            printfn "%s" name
            do! chaos
        return! service req
}
```

```
let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
    fun (service:AsyncArrow<_,_>) req -> async {
        if shouldChaos() then
            printfn "%s" name
            do! chaos
        return! service req
}
```

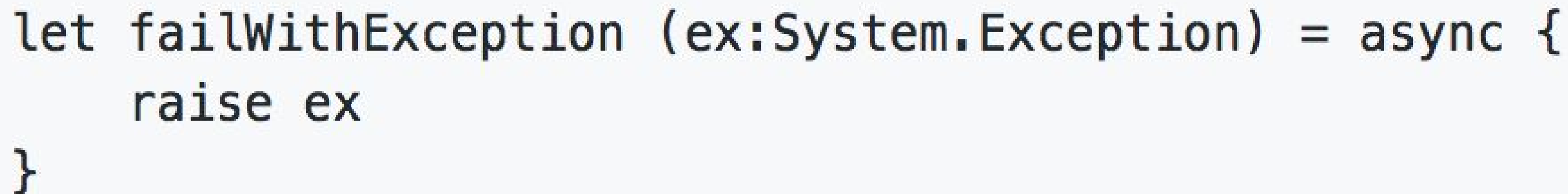
```
let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
    fun (service:AsyncArrow<_,_>) req -> async {
        if shouldChaos() then
            printfn "%s" name
            do! chaos
        return! service req
}
```

```
let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
    fun (service:AsyncArrow<_,_>) req -> async {
        if shouldChaos() then
            printfn "%s" name
            do! chaos
        return! service req
}
```

let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
fun (service:AsyncArrow<_,_>) req -> async {
if shouldChaos() then
printfn "%s" name
do! chaos
return! service req
}

let chaos (name:string) (shouldChaos:unit -> bool) (chaos:Async<unit>) : AsyncFilter<_,_,_,_> =
fun (service:AsyncArrow<_,_>) req -> async {
if shouldChaos() then
printfn "%s" name
do! chaos
return! service req
}

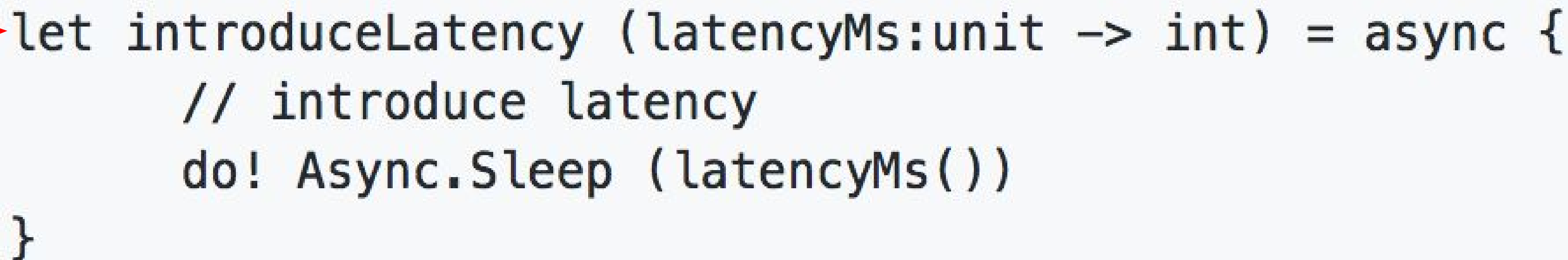
Types of Chaos Failures




```
let introduceLatency (latencyMs:unit -> int) = async {
    // introduce latency
    do! Async.Sleep (latencyMs())
}
```

Types of Chaos Failures

```
let failWithException (ex:System.Exception) = async {
    raise ex
}
```



```
let introduceLatency (latencyMs:unit -> int) = async {
    // introduce latency
    do! Async.Sleep (latencyMs())
}
```

```
// Defines the requirements that need to be met before injecting chaos
let simpleTimeBasedFailure () = System.DateTime.Now.Millisecond = 0

let simpleTimeBasedLatency (latency:int) =
    fun () ->
        if simpleTimeBasedFailure() then latency
        else 0
```

```
// API
let defChaos (a) =
    a
    |> chaos "chaos exception" simpleTimeBasedFailure (failWithException (new System.OutOfMemoryException("chaos")))
    |> chaos "chaos latency 5sec" simpleTimeBasedFailure (introduceLatency (simpleTimeBasedLatency 5000))
```

```
// API
let defChaos (a) =
    a
    |> chaos "chaos exception" simpleTimeBasedFailure (failWithException (new System.OutOfMemoryException("chaos")))
    |> chaos "chaos latency 5sec" simpleTimeBasedFailure (introduceLatency (simpleTimeBasedLatency 5000))
```

Automating Creation of Chaos Experiments

2. Have Good Monitoring in Place for Configuration Changes.

Have Good Monitoring in Place

- RPC Enabled

Have Good Monitoring in Place

- RPC Enabled
  - Associated Hystrix Commands

Have Good Monitoring in Place

- RPC Enabled
  - Associated Hystrix Commands
    - Associated Fallbacks

Have Good Monitoring in Place

- RPC Enabled
  - Associated Hystrix Commands
    - Associated Fallbacks
- Timeouts

Have Good Monitoring in Place

- RPC Enabled
  - Associated Hystrix Commands
    - Associated Fallbacks
- Timeouts
- Retries

Have Good Monitoring in Place

- RPC Enabled
    - Associated Hystrix Commands
        - Associated Fallbacks
- Timeouts
- Retries
- All in One Place!

Lorin Hochstein
@lhochstein

Following

Hypothesis: config changes are more dangerous than code changes.

2:21 PM - 6 Oct 2017

5 Retweets **25** Likes

RPC/Ribbon

- Java library managing REST clients to/from different services
- Fast failing/fallback capability

RPC/Ribbon Timeouts

RPC Timeouts

At what point does the service give up?

Retries

Immediately retrying a failure after an operation is not usually a great idea.

Retries

Understand the logic between your timeouts and your retries.

HYSTRIX

DEFEND YOUR APP

Hystrix Commands/Fallback Paths

If your service is non-critical, ensure that there are fallback paths in place.

Fallback Strategies

Fallback Strategies

Know what your fallback strategy is and how to get that information.

countSuccess
13:10
13:20
13:30
13:40
13:50

countFallbackSuccess
13:10
13:20
13:30
13:40
13:50

countFailure

13:10 13:20 13:30 13:40 13:50

countTimeout

13:10 13:20 13:30 13:40 13:50

3.Ensure Synergy between Hystrix Timeouts, RPC timeouts, and retry logic.

ChAP

ChAP's Monocle

Read Timeout Sequence	Connection Timeout	Max Auto Retries	Max Auto Retries Next Server	Max RPS	Average RPS	Hystrix Commands
150 300	600	0	1	50	35	PersonalizationDependencyCommand No fallback! ⚠

ChAP's Monocle

	Service Name	Client Name	RPC Timeout	Retries	Retries Next Server	Max RPS	Average RPS	Hystrix Commands
+	myService	myClient	5000	0	1	0	0	Unwrapped RPC call! ⚠

ChAP's Monocle

Service Name	Client Name	NIWS App Name	RPC Timeout	Retries
+ dinosaur	DINOSAUR	dinosaur	4000	1 Anti-pattern!

There isn’t always money in microservices

Criticality Score

Criticality Score

*RPS Stats Range bucket * number of retries * number of Hystrix Commands = Criticality Score*

Max RPS	Average RPS
17,295	12,640

Criticality Score

*RPS Stats Range bucket * number of retries * number of Hystrix Commands = Criticality Score*

Max Auto Retries	Max Auto Retries Next Server
0	1

Criticality Score

*RPS Stats Range bucket * number of retries * number of Hystrix Commands = Criticality Score*

Hystrix Commands

Criticality Score

*RPS Stats Range bucket * number of retries * number of Hystrix Commands = Criticality Score*

Chaos Success Stories

“We ran a chaos experiment which verifies that our fallback path works and it successfully caught a issue in the fallback path and the issue was resolved before it resulted in any availability incident!”

“While [failing calls] we discovered an increase in license requests for the experiment cluster even though fallbacks were all successful...

“While [failing calls] we discovered an increase in license requests for the experiment cluster even though fallbacks were all successful. ...This likely means that whoever was consuming the fallback was retrying the call, causing an increase in license requests.”

Don’t lose sight of your company’s customers.

Takeaways

- Designing for resiliency testability is a shared responsibility.
- Configuration changes can cause outages.
- Have explicit monitoring in place on antipatterns in configuration changes.

@nora_js