Applying Principles of Chaos Engineering to Serverless

history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy

est. **400K deaths per year** in 18th Century Europe.

history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy

est. **400K deaths per year** in 18th Century Europe.

1798

first vaccine developed

Edward Jenner

history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy

est. **400K deaths per year** in 18th Century Europe.

1798

first vaccine developed

Edward Jenner

WHO certified
global eradication

1980

hepatitis C
measles
hepatitis B
HIV
influenza
rubella
rotavirus
tetanus
YF
mumps
Ebola virus
diphtheria
tick encephalitis
papillomavirus
tuberculosis
J Enceph
chickenpox
polio
cytomegalovirus
whooping cough
RSV
bacterial meningitis
hep. A
HiB
typhoid
pneumococcus
malaria

1900 1920 1940 1960 1980 2000

Time from identifying the pathogen to the development of vaccine

Vaccine developed

Partially-effective vaccine available

No vaccine yet available

Vaccination is the most effective method of preventing infectious diseases

stimulates the immune system to recognize and destroy the disease before contracting the disease for real

Chaos Engineering

controlled experiments to help us **learn** about our system's behaviour and **build confidence** in its ability to withstand turbulent conditions

Yan Cui

http://theburningmonk.com

@theburningmonk

Principal Engineer @ DAZN

Yan Cui

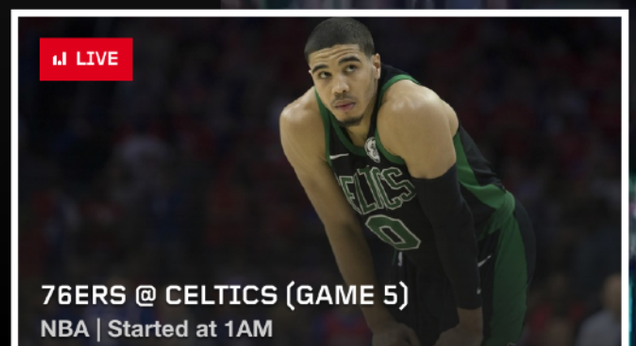
http://theburningmonk.com

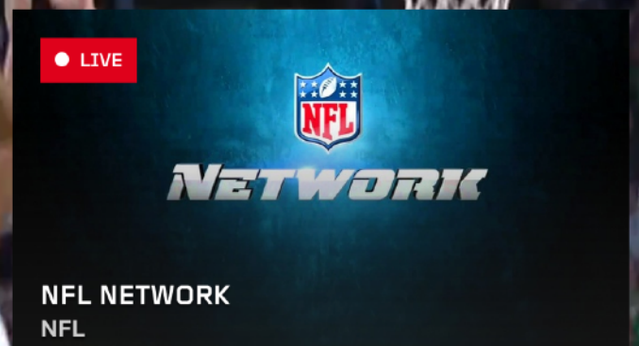
@theburningmonk

Principal Engineer @ DAZN

HOME SCHEDULE SPORTS MENU

WHAT'S ON





DON'T MISS

DAZN

"Netflix for sports"

offices in London, Leeds, Katowice and Amsterdam

HOME SCHEDULE SPORTS

FOOTBALL

AMERICAN FOOTBALL

BASKETBALL

ICE HOCKEY

BOXING

TENNIS

DARTS

BASEBALL

RUGBY UNION

FIELD HOCKEY

MOTORSPORT

GOLF

VOLLEYBALL

FIGHT SPORTS

KICK BOXING

HORSE RACING

POOL

Latest on Perform Group PLC

Sports media group Perform seals $1bn boxing deal

Lock-up expiries will unlock IPO demand

Perform surges on Blavatnik offer

Perform outlines plan to rebuild trust

Perform Group PLC + Add to myFT

DAZN to screen Italian football's top league

Streaming service's deal for Serie A games is latest challenge to broadcasters

Juventus celebrate after winning this year's Serie A

available in Austria, Switzerland,
Germany, Japan, Canada and Italy

US coming soon ;-)

Online TV

UK-based sport streaming service adopts Netflix model after $1bn deal

DAZN aims to revolutionise US pay-per-view boxing after link-up with promoter Eddie Hearn

Mark Sweney

@marksweney | Email

Mon 14 May 2018 15.43 BST

55

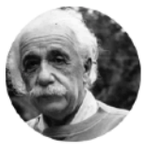
This article is over 4 weeks old

▲ Eddie Hearn's Matchroom Boxing promotes fighters including the world heavyweight boxing champion Anthony Joshua (left). Photograph: Andrew Couldridge/Reuters

The Netflix and Amazon model that has revolutionised entertainment TV viewing is being brought to the world of sport, as a London-based streaming

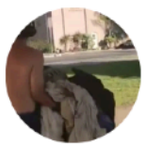
most viewed in US


Laird Hamilton's age-
10-point plan to super
your body


Einstein's travel diari
'shocking' xenophobi


Donald Trump was rig
rest of the G7 were wr
George Monbiot


Populist talkshows fu
far right, German TV b
told


Jogger who trashed h
man's things charged
robbery in new dispu

available on 30+ platforms

~500,000 concurrent viewers

DAZN

"Netflix for sports"

offices in London, Leeds, Katowice and Amsterdam

MARCH 16TH, 2017

Breaking Things on Purpose

Mathias Lafeldt
Infrastructure Developer

Off the top of your head, do you know the answer to these questions: what will happen to your application if Amazon S3—one of the most widely used Amazon Web Services—suddenly

29
MAR

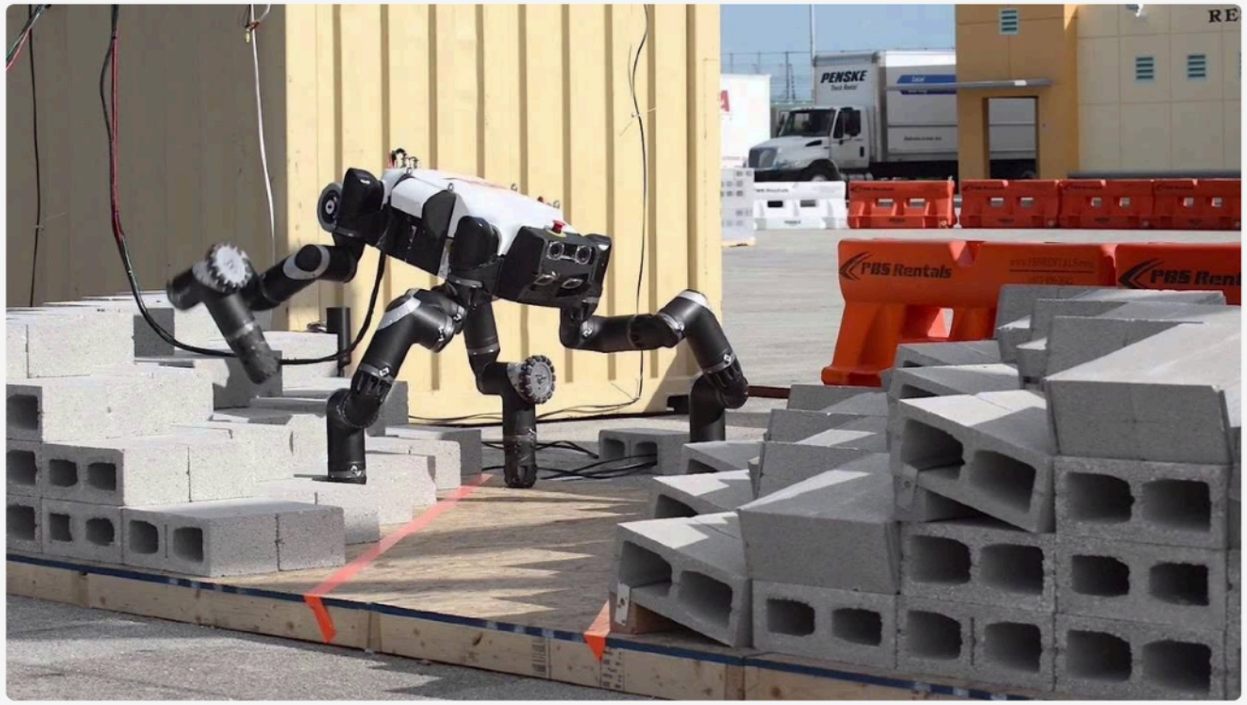
Past Meetup

The joys of destruction

Hosted by Steve Ganly
From London SRE Meetup
Public group

You went 28 people went

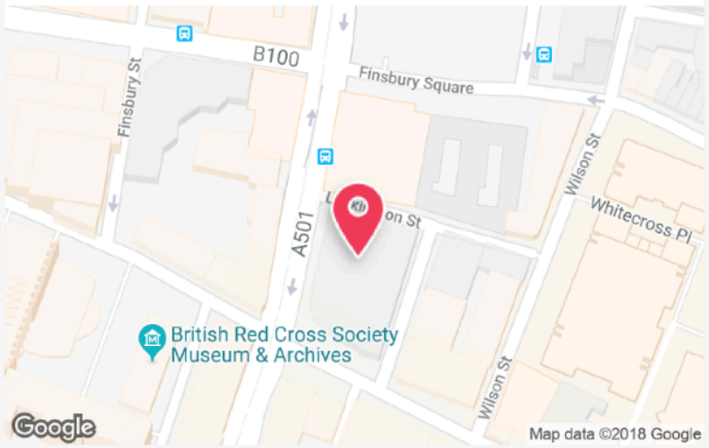
You shared feedback on August 29. As more people give feedback, you'll see the highlights here.



Thursday, March 29, 2018
6:30 PM to 8:30 PM

Skills Matter | CodeNode
10 South Place, London, EC2M 2RB · London

How to find us
The lovely Skills Matter people will tell you which room we're in on the night.



Why did you break production?


Because I can!

Kolton Andrus, CEO of Gremlin

Russ Miles, CEO of ChaosIQ

Nora Jones, Chaos Engineer at Netflix

Kolton Andrus, CEO of Gremlin

Russ Miles, CEO of ChaosIQ

Nora Jones, Chaos Engineer at Netflix

THE FLY

Cartoon-Box 20

it’s about building confidence,

NOT breaking things

PRINCIPLES OF CHAOS ENGINEERING

Last Update: 2017 April

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.

Advances in large-scale, distributed software systems are changing the game for software engineering. As an industry, we are quick to adopt practices that increase flexibility of development and velocity of deployment. An urgent question follows on the heels of these benefits: How much confidence we can have in the complex systems that we put into production?

Even when all of the individual services in a distributed system are functioning properly, the interactions between those services can cause unpredictable outcomes. Unpredictable outcomes, compounded by rare but disruptive real-world events that affect production environments, make these distributed systems inherently chaotic.

We need to identify weaknesses before they manifest in system-wide, aberrant behaviors. Systemic weaknesses could take the form of: improper fallback settings when a service is unavailable; retry storms from improperly tuned timeouts; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes; etc. We must address the most significant weaknesses proactively, before they affect our customers in production. We need a way to manage the chaos inherent in these systems, take advantage of increasing flexibility and velocity, and have confidence in our production deployments despite the complexity that they represent.

An empirical, systems-based approach addresses the chaos in distributed systems at scale and builds confidence in the ability of those systems to withstand realistic conditions. We learn about the behavior of a distributed system by observing it during a controlled experiment. We call this *Chaos Engineering*.

CHAOS IN PRACTICE

To specifically address the uncertainty of distributed systems at scale, Chaos Engineering can be thought of as the facilitation of experiments to uncover systemic weaknesses. These experiments follow four steps:

1. Start by defining 'steady state' as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

The harder it is to disrupt the steady state, the more confidence we have in the behavior of the system. If a weakness is uncovered, we now have a target for improvement before that behavior manifests in the system at large.

http://principlesofchaos.org

STEP 1.

define “Steady State”

aka. what does normal, working condition looks like?


this is not a steady state

STEP 2.

hypothesize steady state will continue in both control group & the experiment group

ie. you should have a reasonable degree of confidence the system would handle the failure before you proceed with the experiment

explore **unknown unknowns**
away from production

treat production with the care it deserves

the goal is **NOT**,

to actually hurt production

If you know the system would break,
and you did it anyway...

then it’s NOT a chaos experiment.

It’s called being **IRRESPONSIBLE**.

STEP 3.

inject realistic failures

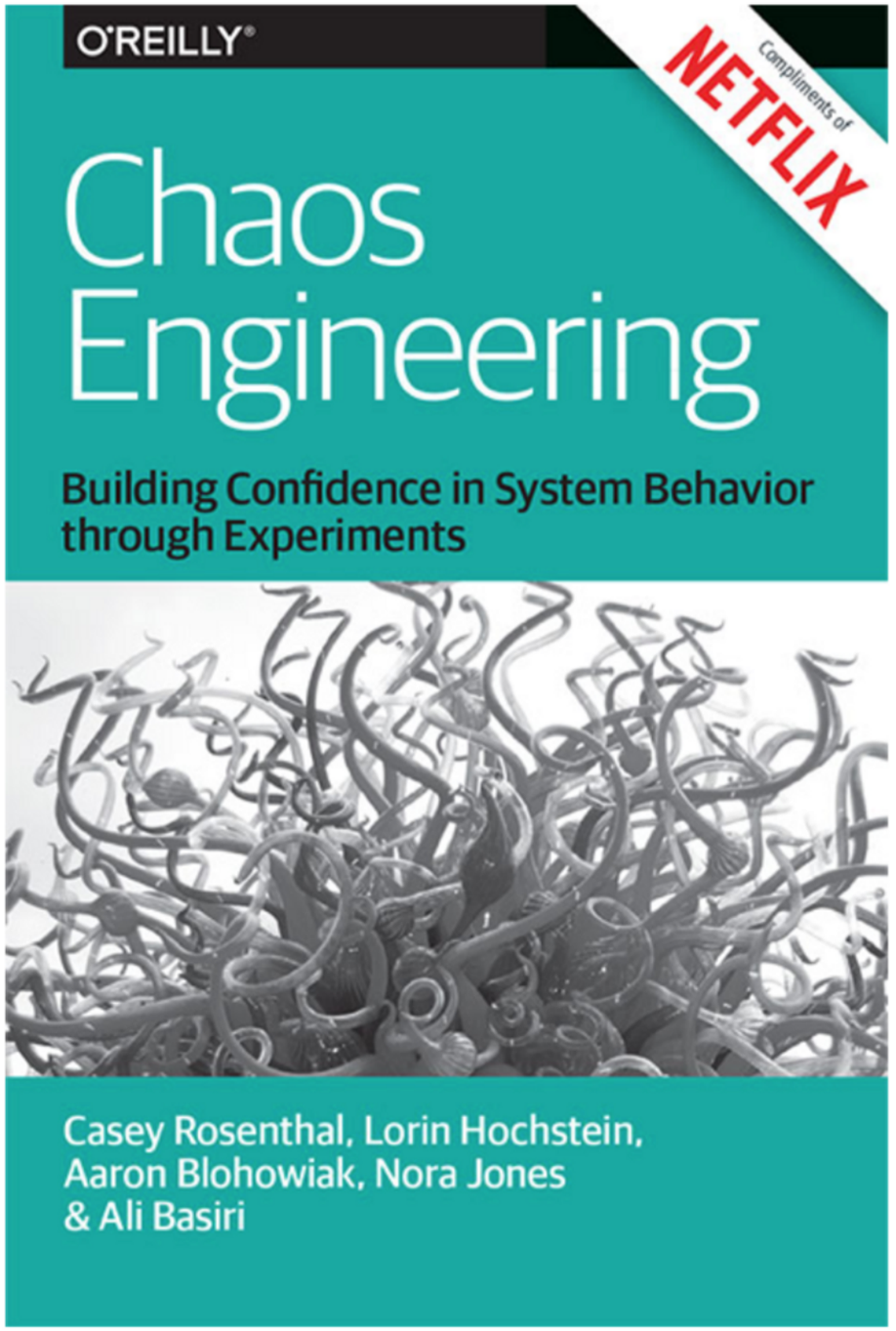
e.g. server crash, network error,
HD malfunction, etc.

NETFLIX

https://github.com/Netflix/SimianArmy

NETFLIX

https://github.com/Netflix/SimianArmy

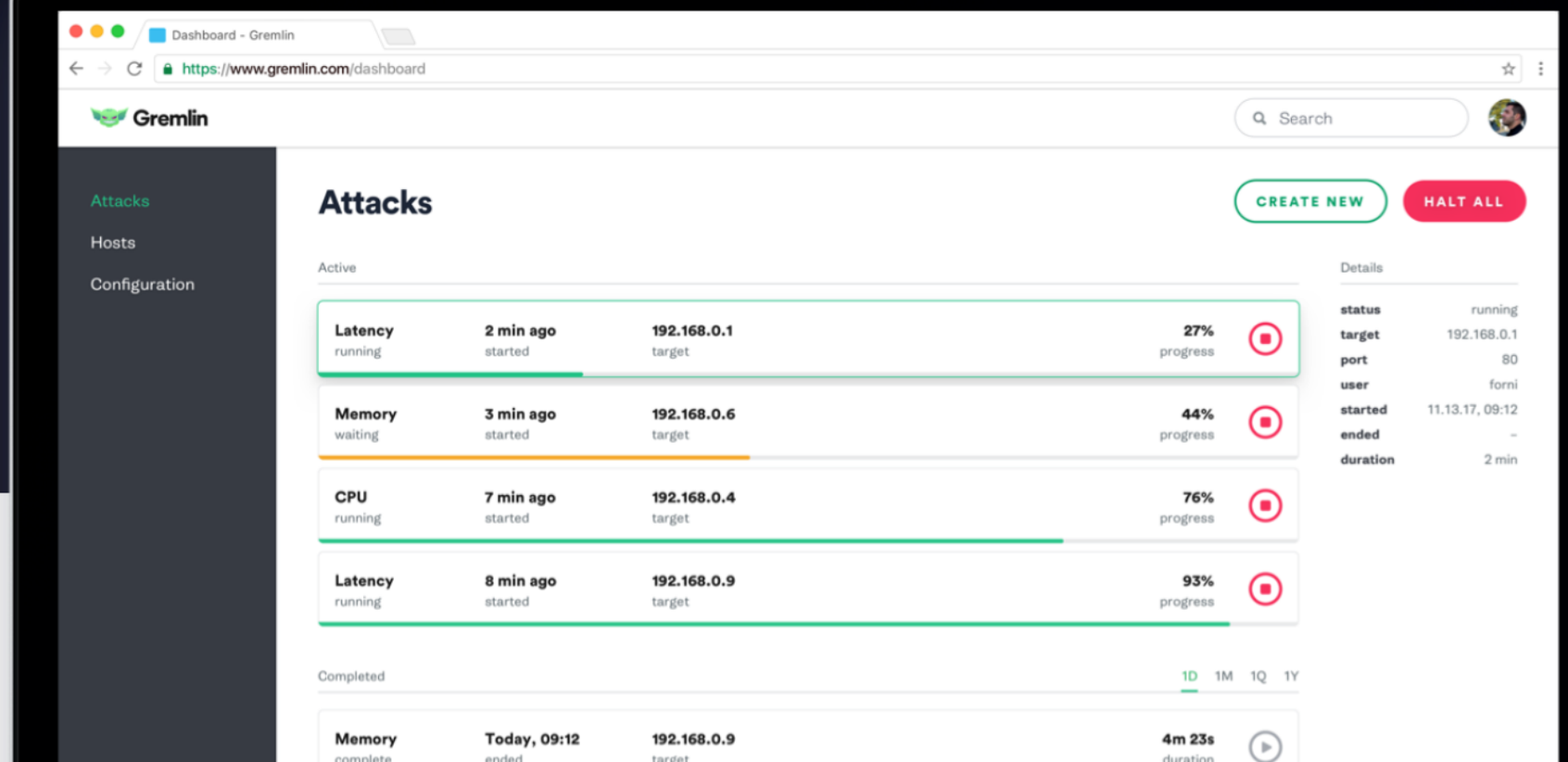


http://oreil.ly/2tZU1Sn

Turn failure into resilience.

Gremlin provides you the framework to safely, securely, and easily simulate real outages with an ever-growing library of attacks.

REQUEST A DEMO



STEP 4.

disprove hypothesis

i.e. look for difference with steady state

if a **WEAkNESS** is uncovered,

IMPROVE it before the behaviour manifests in the system at large

Chaos Engineering

controlled experiments to help us **learn** about our system's behaviour and **build confidence** in its ability to withstand turbulent conditions

Chaos Engineering

controlled experiments to help us **learn** about our system's behaviour and **build confidence** in its ability to withstand turbulent conditions

COMMUNICATION

ensure everyone knows what you’re doing

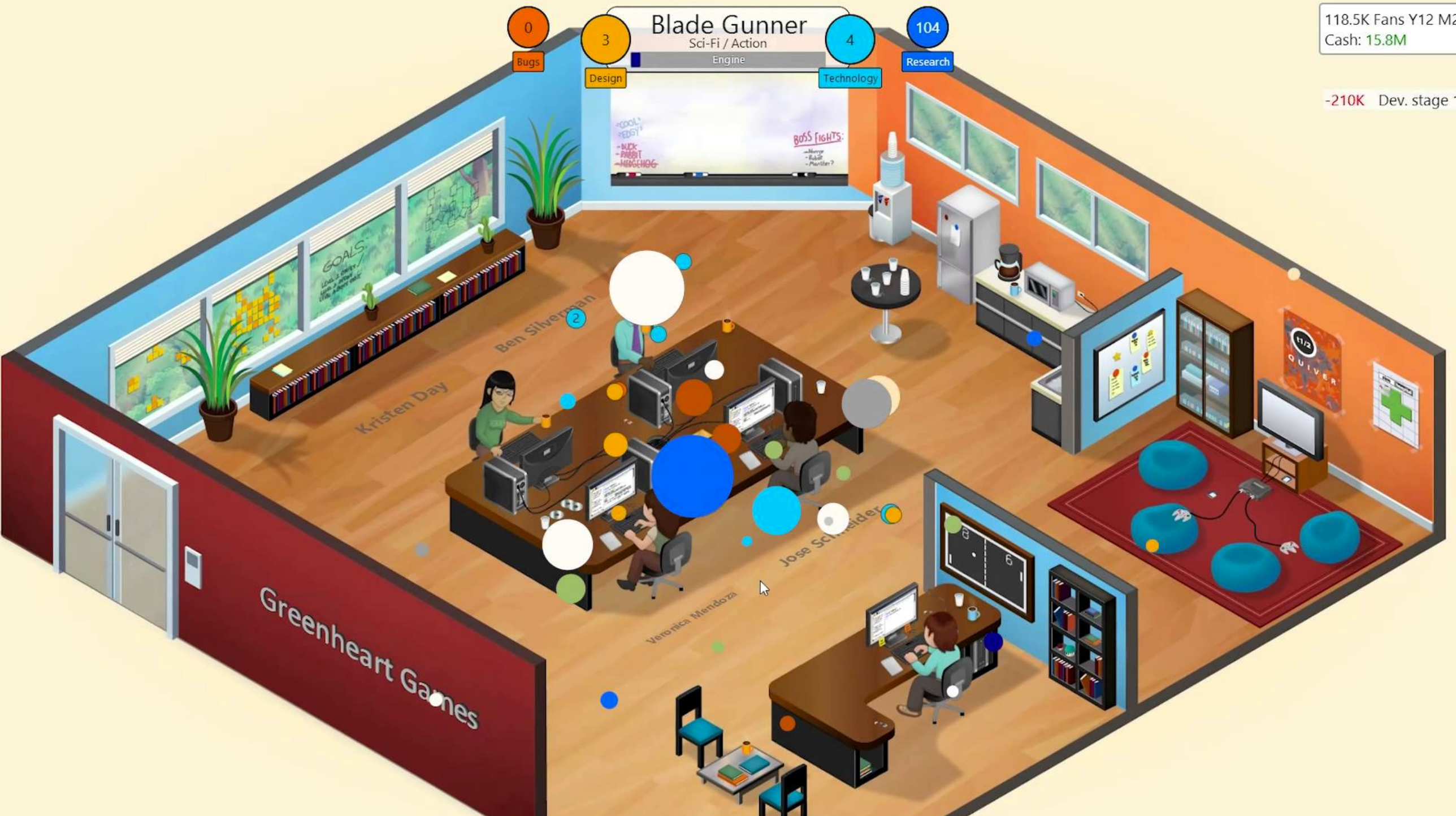
ensure everyone knows what you're doing

NO surprises!

COMMUNICATION

TIMING

run experiments during office hours



AVOID important dates

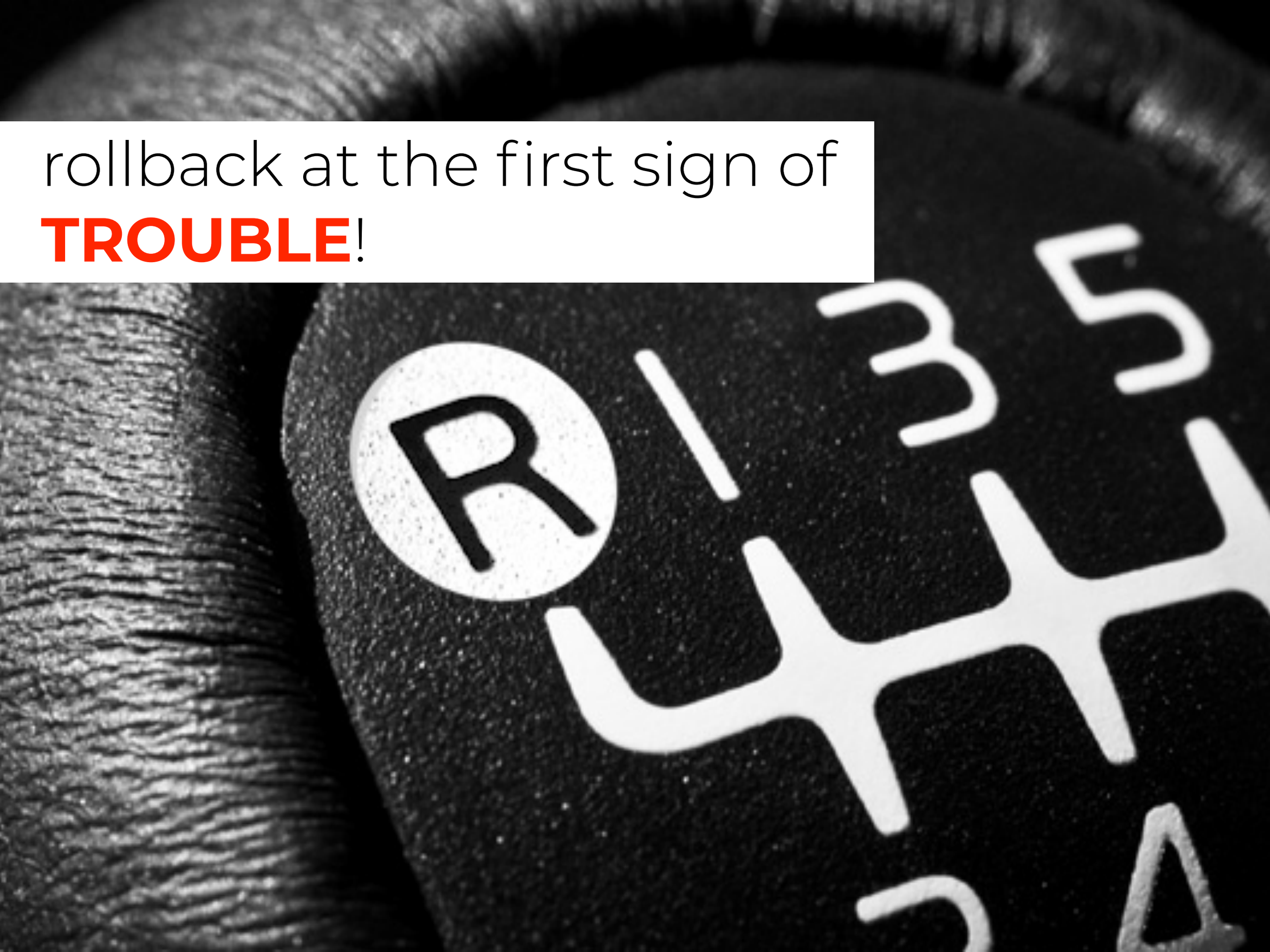
COMMUNICATION

TIMING

CONTAIN BLAST RADIUS

smallest change that allows you to detect a signal that steady state is disrupted

rollback at the first sign of **TROUBLE**!

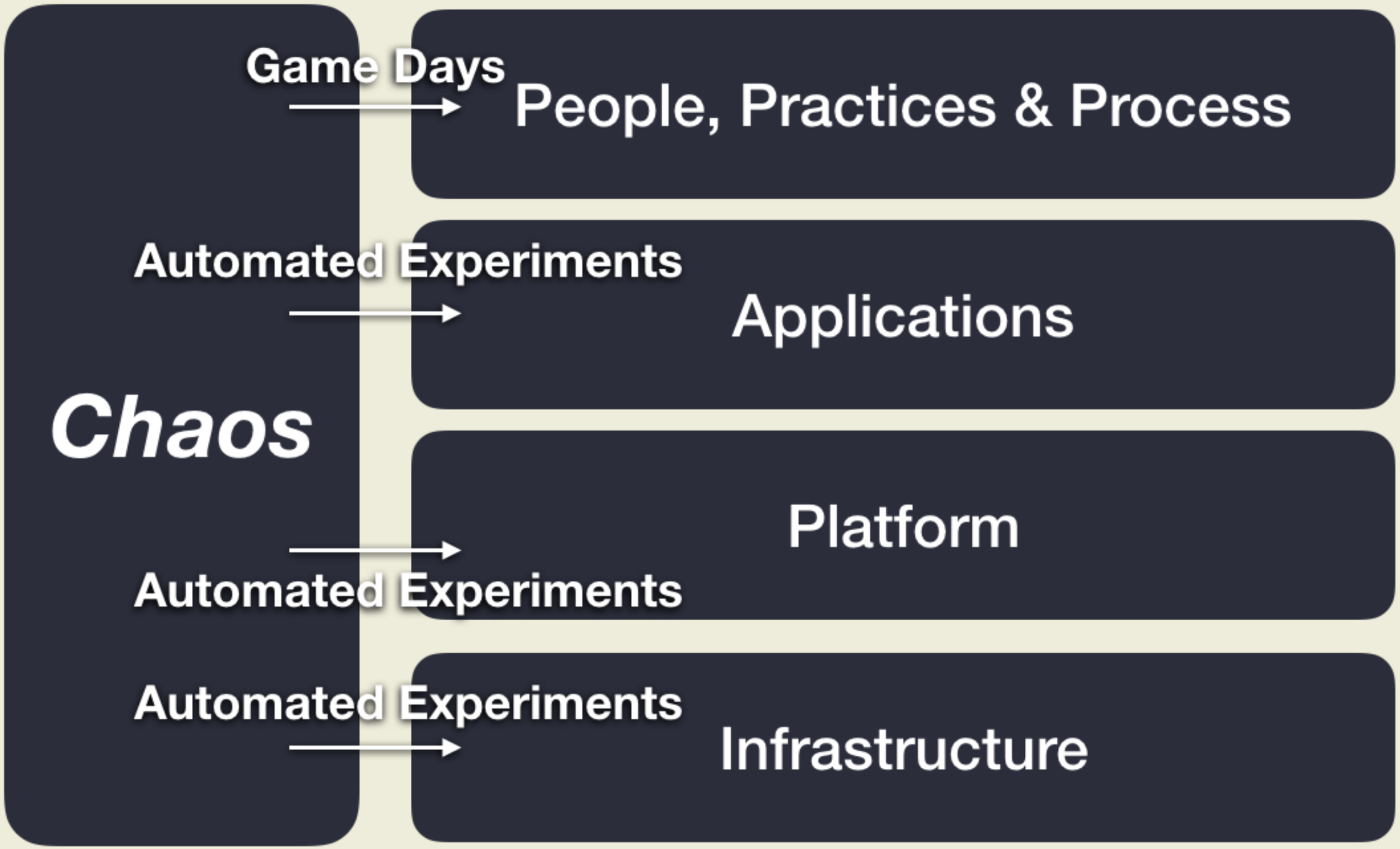


COMMUNICATION

TIMING

CONTAIN BLAST RADIUS

don´t try to run before you know how to walk.



by Russ Miles @russmiles
source https://medium.com/russmiles/chaos-engineering-for-the-business-17b723f26361

NETFLIX

latency monkey induces artificial delay in APIs

chaos monkey kills an EC2 instance

chaos gorilla kills an AWS Availability Zone

chaos kong kills an entire AWS region

shoreditch-ops / **chaos-lambda**

Watch 10 | Star 175 | Fork 9

Code | Issues 8 | Pull requests 1 | Projects 0 | Wiki | Insights

Serverless chaos monkey for AWS (runs on AWS Lambda) ☁️ 💥

reliability-engineering | aws | fault-tolerance | chaos-monkey

58 commits | **1** branch | **6** releases | **3** contributors | MPL-2.0

Branch: **master** | **New pull request** | **Create new file** | **Upload files** | **Find file** | **Clone or download**

hassy Merge pull request #19 from davinerd/tags-support — Latest commit f981e5e on 3 Dec 2017

bin	Rename files for the new name	11 months ago
lambda	added support for tags	4 months ago
lib/commands	fix: Resolve module path correctly	11 months ago
.gitignore	Updating .gitignore	2 years ago
CONTRIBUTING.md	Rename Chaos Lambda to Chaos Llama	11 months ago
Chaosfile.json	added support for tags	4 months ago
LICENSE.txt	All basics in place	2 years ago
README.md	doc: Update link to website in README	8 months ago
package.json	2.0.2	11 months ago

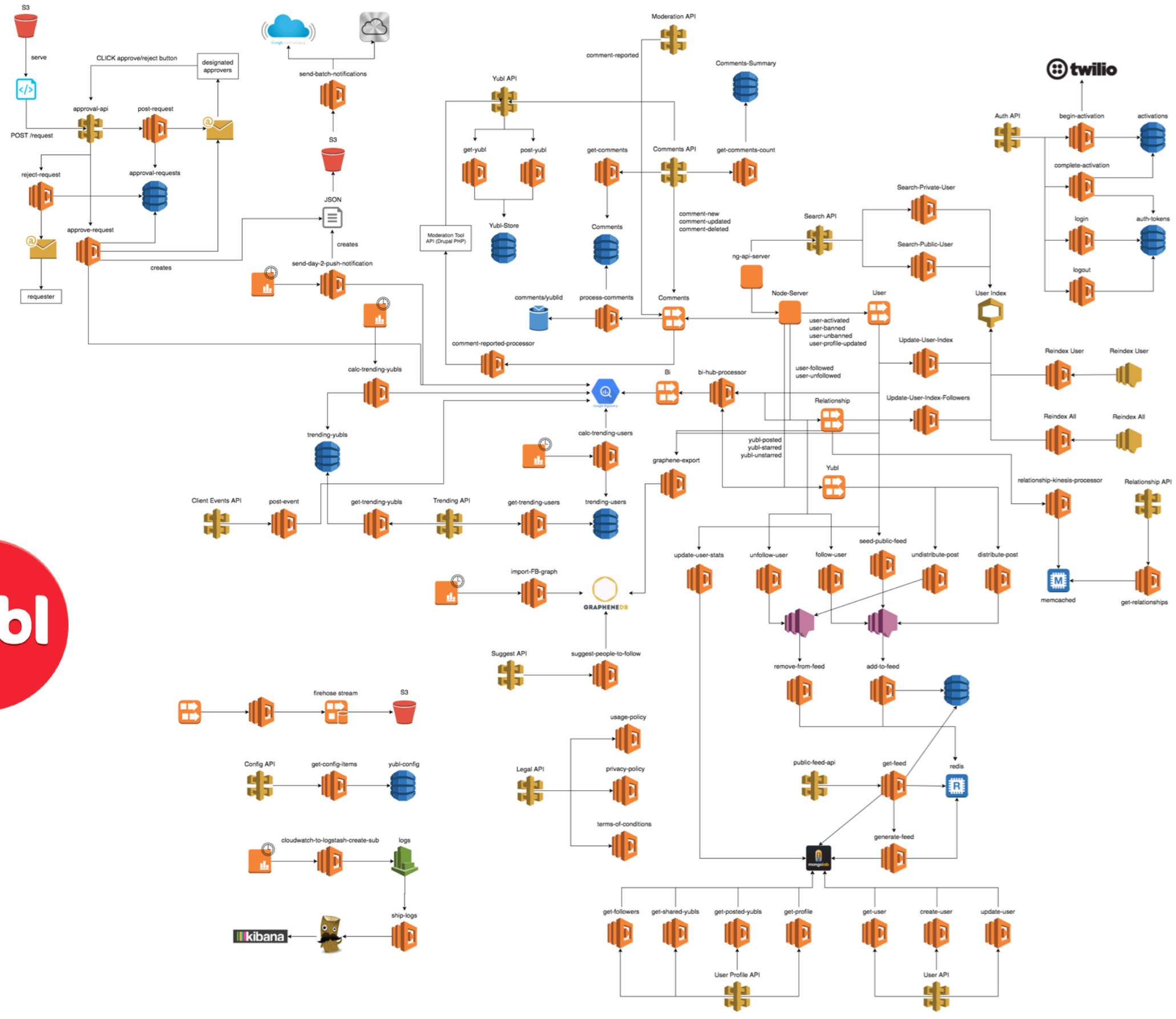
there is no server...

there is no server...

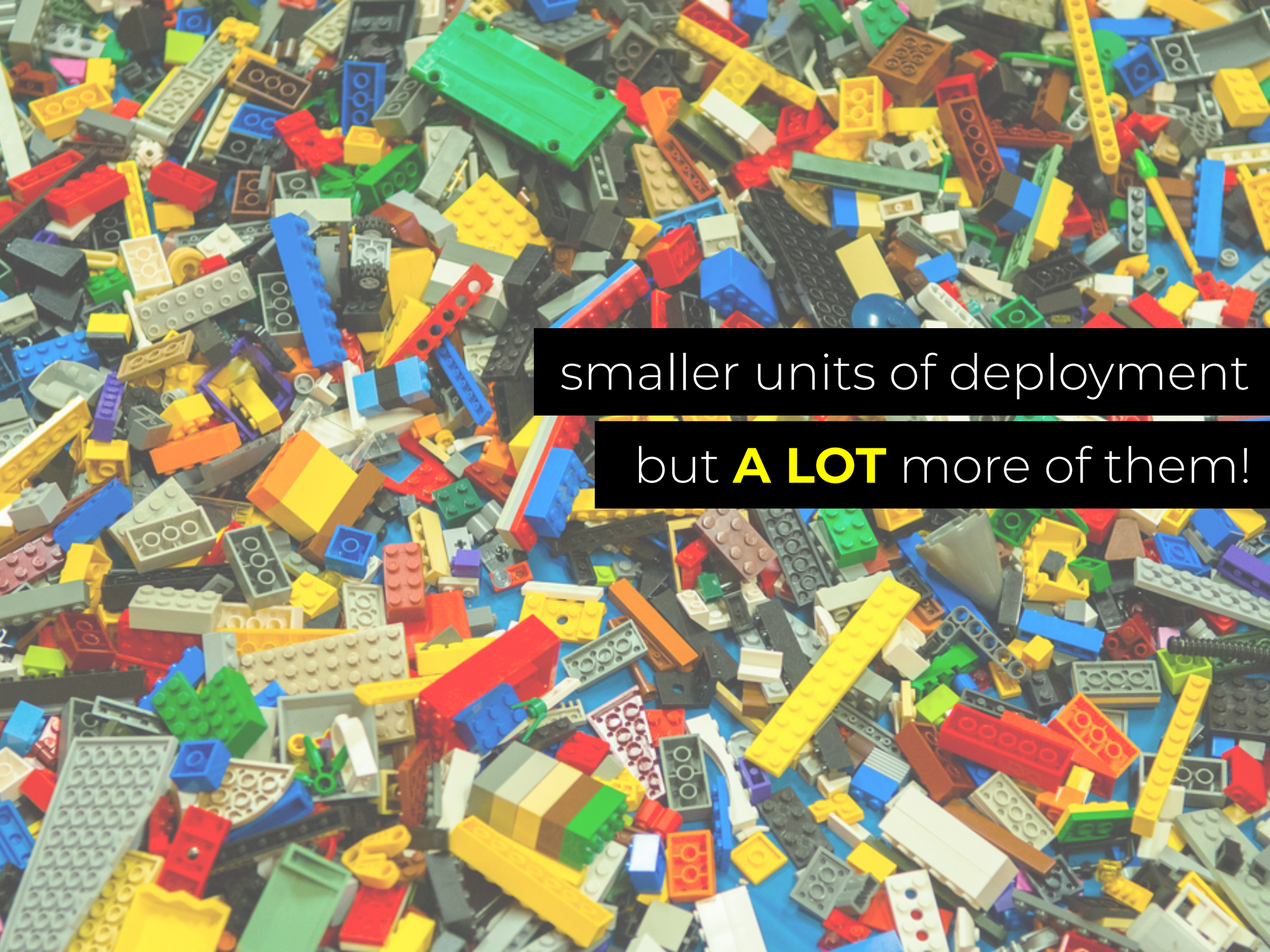
that you can kill

> *"**We need to identify weaknesses before they manifest in system-wide, aberrant behaviors**. Systemic weaknesses could take the form of: improper fallback settings when a service is unavailable; retry storms from improperly tuned timeouts; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes; etc. We must address the most significant weaknesses proactively, before they affect our customers in production. **We need a way to manage the chaos inherent in these systems**, take advantage of increasing flexibility and velocity, and **have confidence in our production deployments despite the complexity that they represent.**"*
>
> *— Principles of Chaos Engineering*

yubl


S3
serve
POST /request
CLICK approve/reject button
designated approvers
approval-api
post-request
reject-request
approval-requests
approve-request
creates
requester
send-batch-notifications
S3
JSON
creates
send-day-2-push-notification
Moderation API
comment-reported
Yubl API
get-yubl
post-yubl
Yubl-Store
Moderation Tool API (Drupal PHP)
get-comments
Comments API
get-comments-count
Comments-Summary
Comments
comment-new
comment-updated
comment-deleted
comments/yublid
process-comments
Comments
comment-reported-processor
twilio
Auth API
begin-activation
activations
complete-activation
login
auth-tokens
logout
Search API
Search-Private-User
Search-Public-User
ng-api-server
Node-Server
User
User Index
user-activated
user-banned
user-unbanned
user-profile-updated
Update-User-Index
Reindex User
Reindex User
user-followed
user-unfollowed
Relationship
Update-User-Index-Followers
Reindex All
Reindex All
calc-trending-yubls
trending-yubls
Bi
bi-hub-processor
calc-trending-users
graphene-export
yubl-posted
yubl-starred
yubl-unstarred
Yubl
relationship-kinesis-processor
Relationship API
Client Events API
post-event
get-trending-yubls
Trending API
get-trending-users
trending-users
update-user-stats
unfollow-user
follow-user
seed-public-feed
undistribute-post
distribute-post
memcached
get-relationships
import-FB-graph
GRAPHENEDB
Suggest API
suggest-people-to-follow
remove-from-feed
add-to-feed
firehose stream
S3
usage-policy
Legal API
privacy-policy
terms-of-conditions
public-feed-api
get-feed
redis
generate-feed
Config API
get-config-items
yubl-config
cloudwatch-to-logstash-create-sub
logs
ship-logs
kibana
get-followers
get-shared-yubls
get-posted-yubls
get-profile
get-user
create-user
update-user
User Profile API
User API

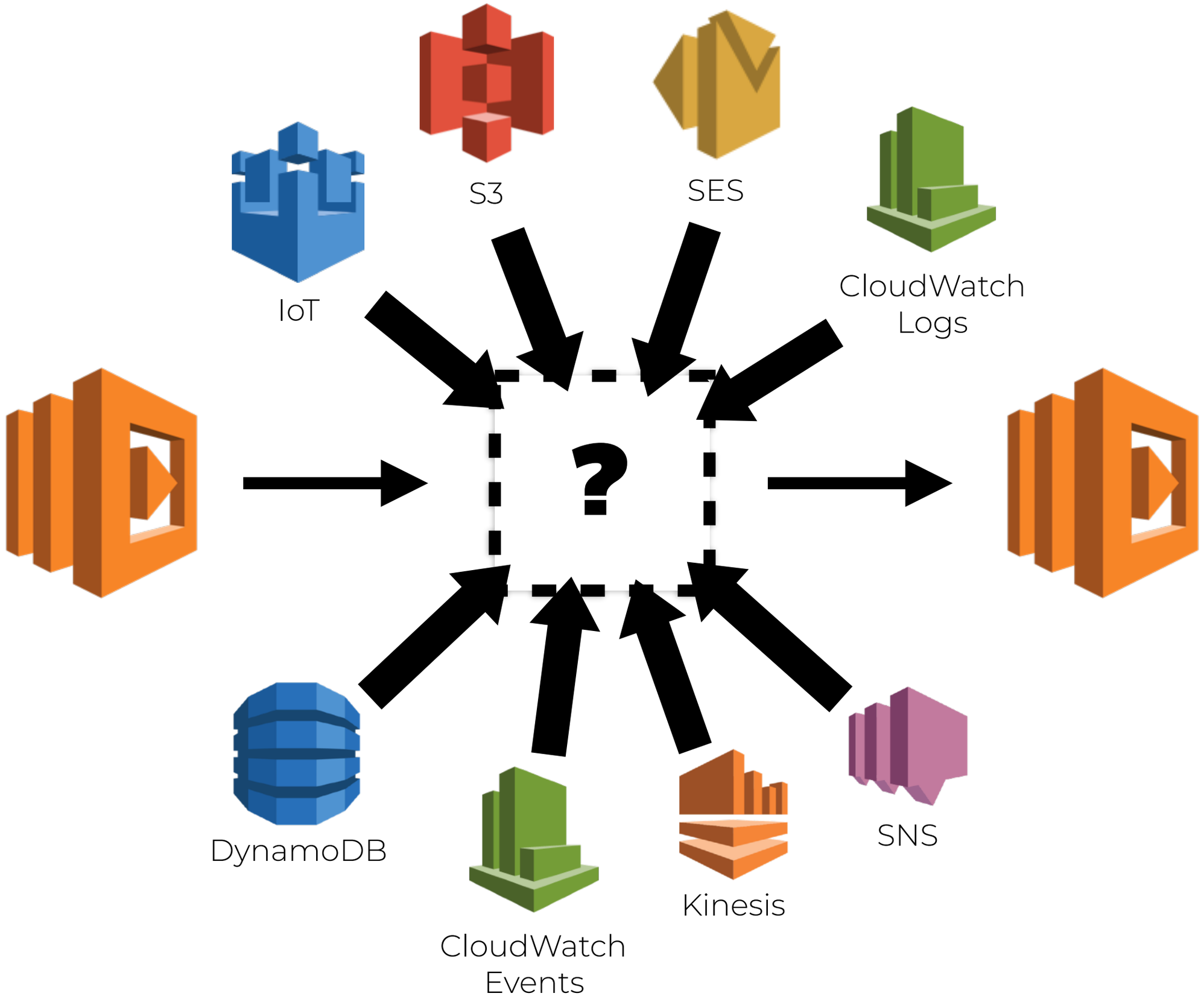
there are **more** inherent chaos and complexity in a Serverless architecture


smaller units of deployment
but **A LOT** more of them!

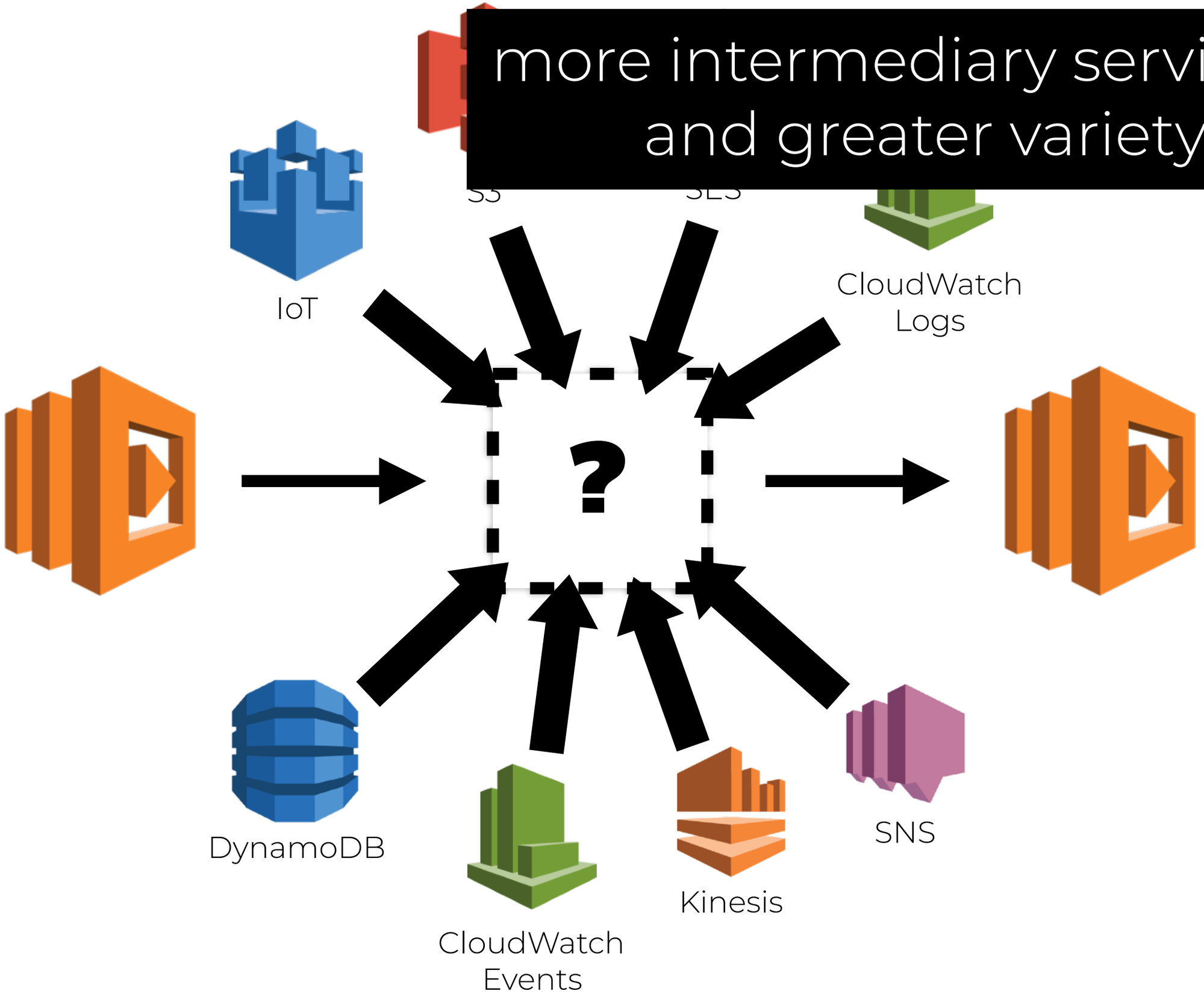
serverful

serverless

more difficult to harden
around boundaries


S3
SES
IoT
CloudWatch
Logs
?
DynamoDB
CloudWatch
Events
Kinesis
SNS

more intermediary services,
and greater variety too



more intermediary services, and greater variety too

S3

IoT

SES

CloudWatch Logs

each with its own set of failure modes

DynamoDB

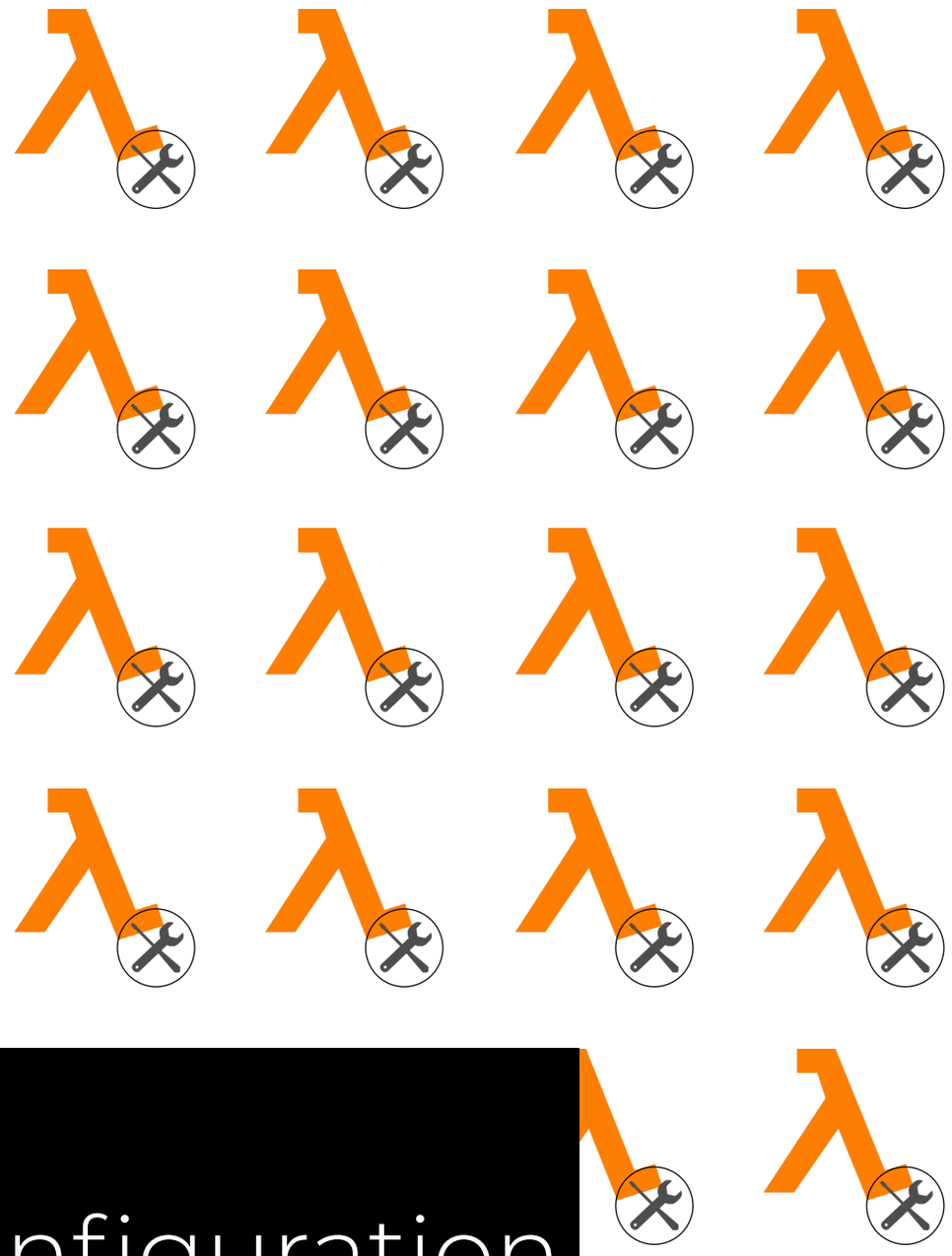
CloudWatch Events

Kinesis

SNS

serverful

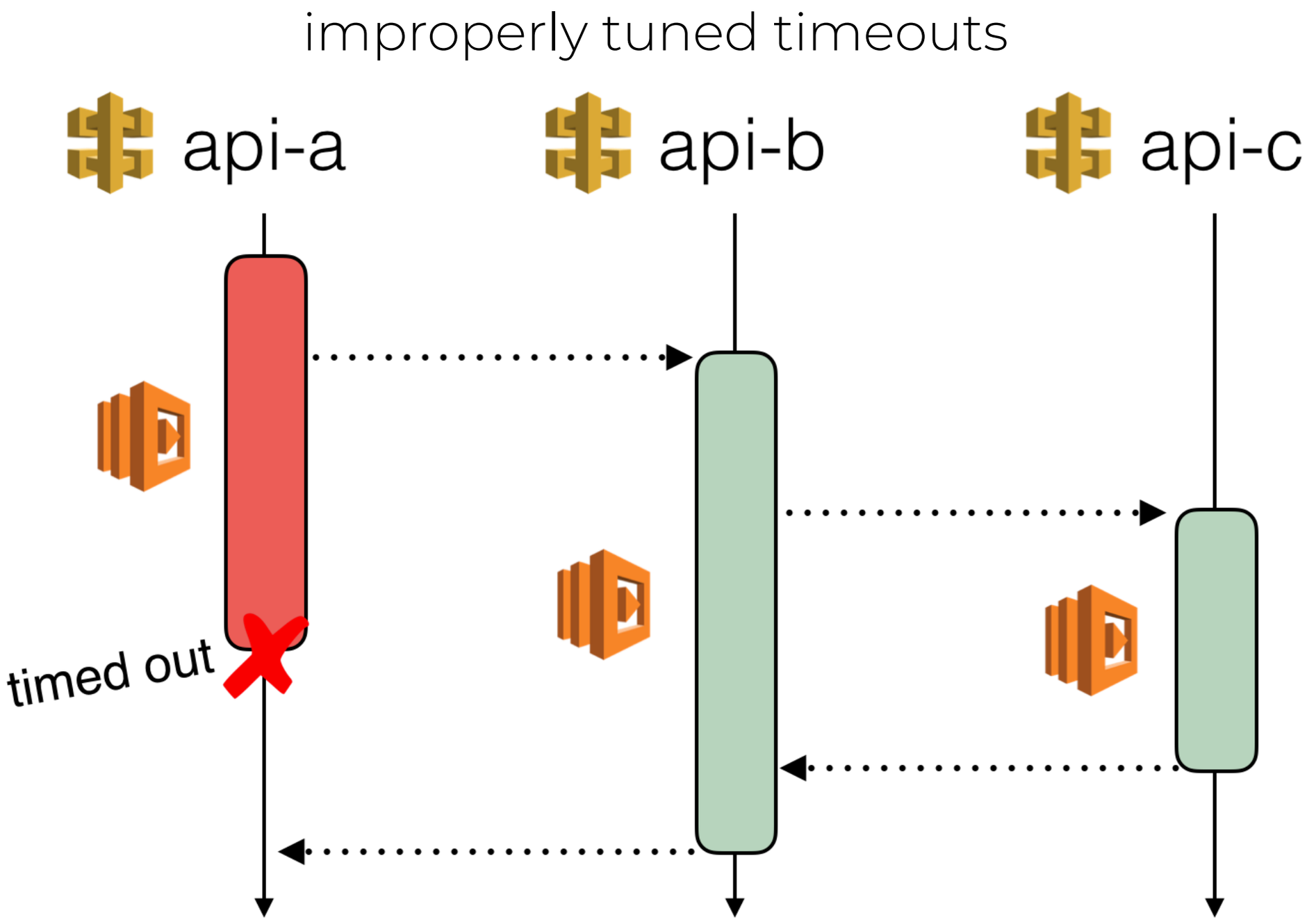
serverless

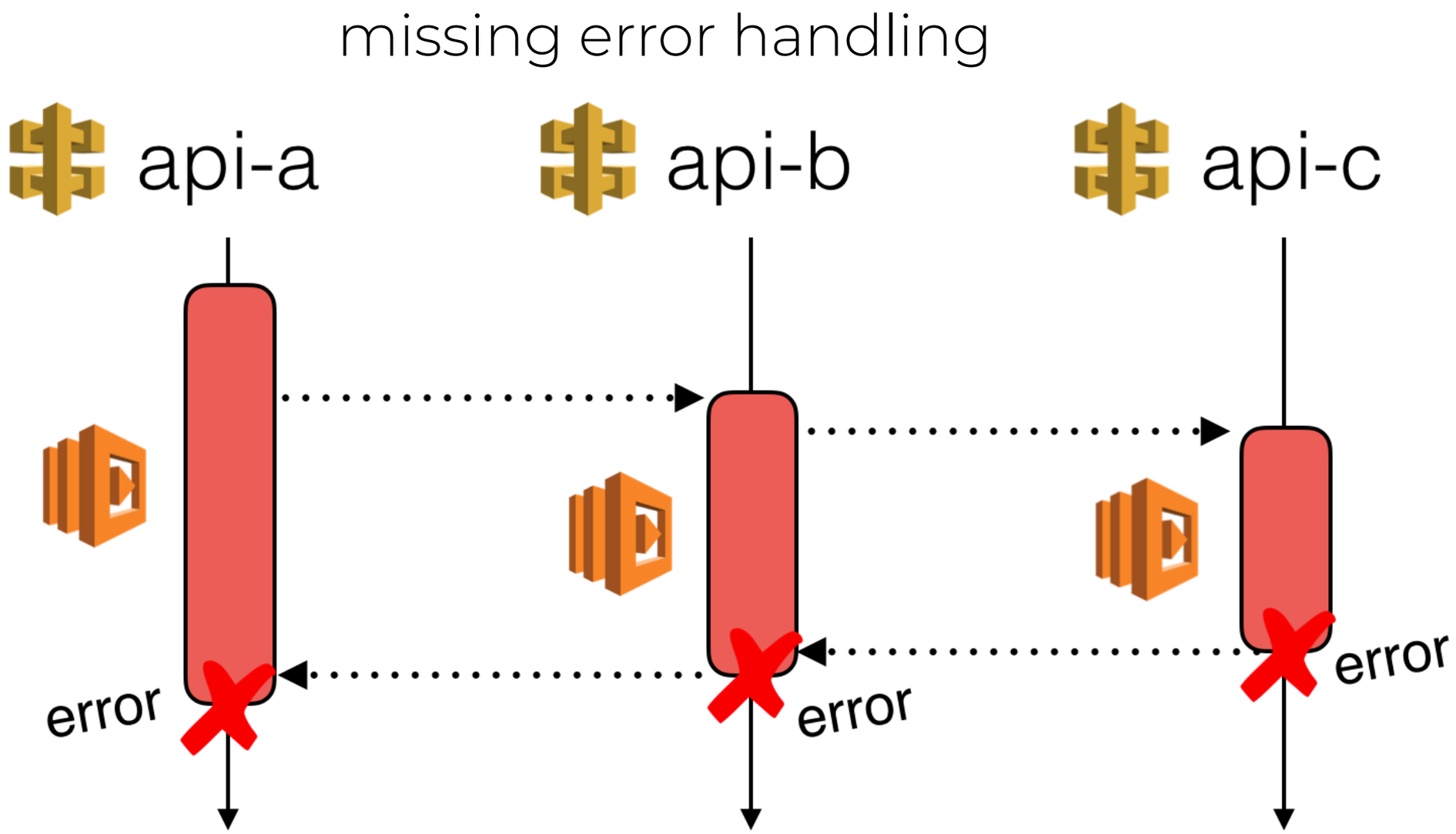


more configurations,
more opportunities for misconfiguration

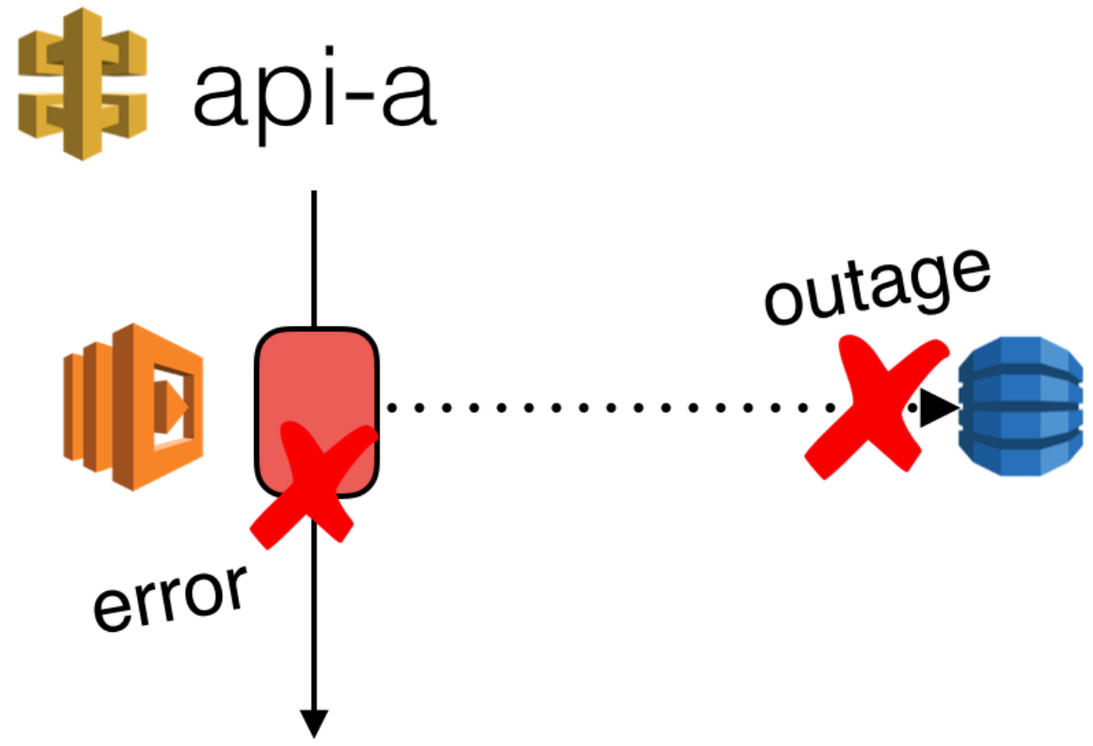
more **unknown** failure modes in infrastructure that we don't control

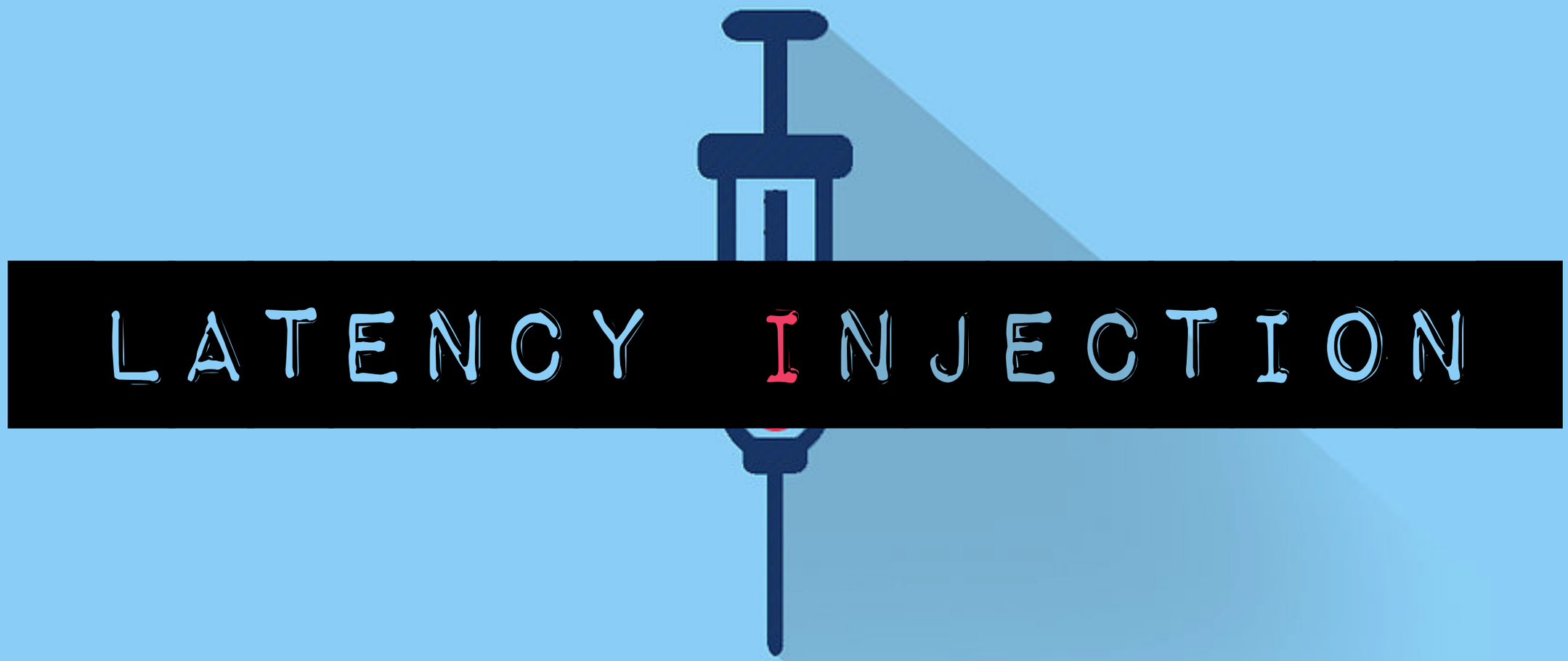
often there’s little we can do when an outage occurs in the platform


improperly tuned timeouts
api-a
api-b
api-c
timed out


missing error handling
api-a
api-b
api-c
error
error
error

missing fallback when downstream is unavailable




LATENCY INJECTION

STEP 1.

define “Steady State”

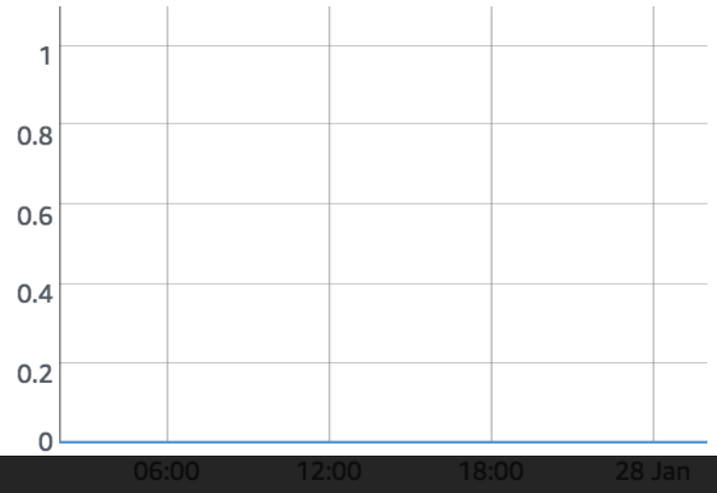
aka. what does normal, working condition looks like?

apig-iscoldstart-dev-hello

Qualifiers Actions Select a test event..

Invocation count Last 24 hours

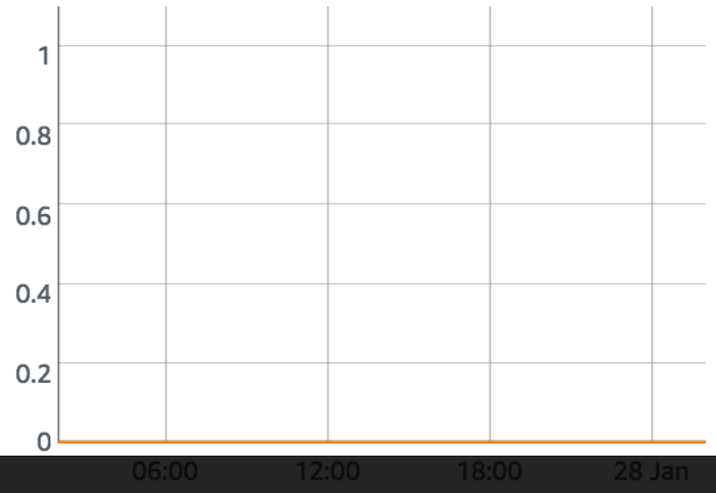
Jump to Metrics Jump to Logs



Count

Invocation duration Last 24 hours

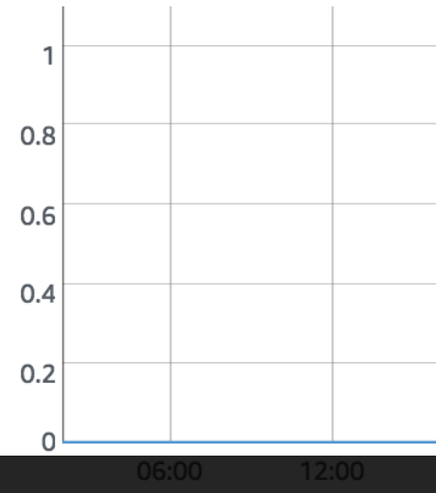
Jump to Metrics Jump to Logs



Max Milliseconds
Min Milliseconds

Invocation errors

Jump to Metrics

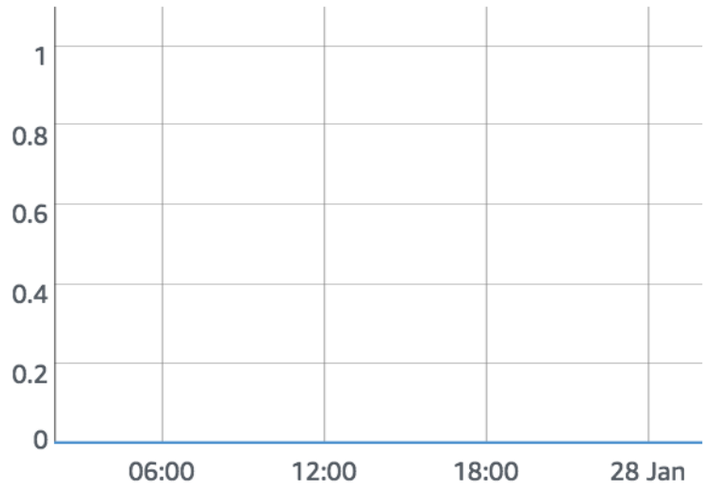


Count

what metrics do you monitor?

Throttled invocations Last 24 hours

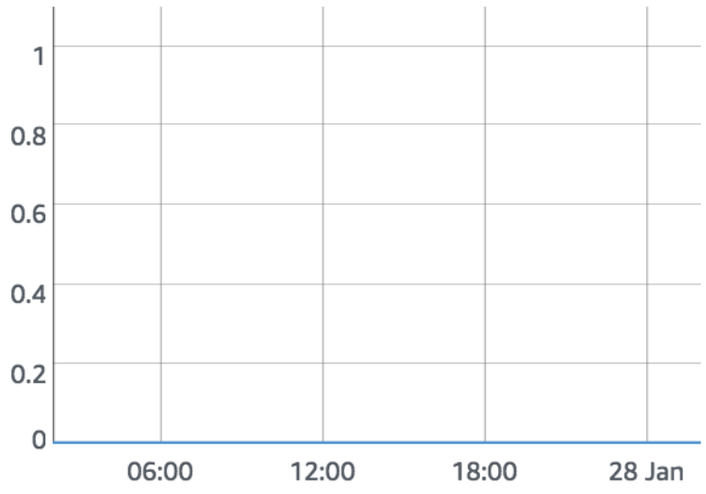
Jump to Metrics Jump to Logs



Count

Iterator age Last 24 hours

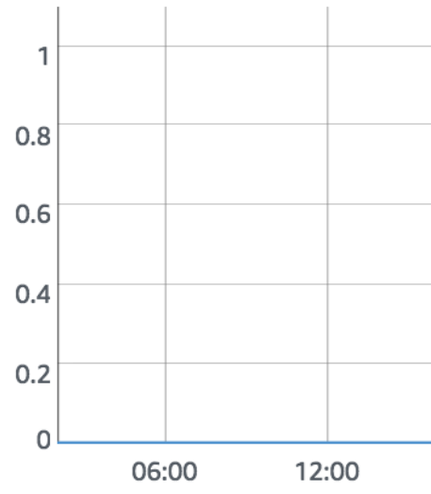
Jump to Metrics Jump to Logs



Max Milliseconds

DLQ errors

Jump to Metrics



Count

9X-percentile latency ✓

error count ✓

yield (% of requests completed) ✓

harvest (completeness of results) ✓

STEP 2.

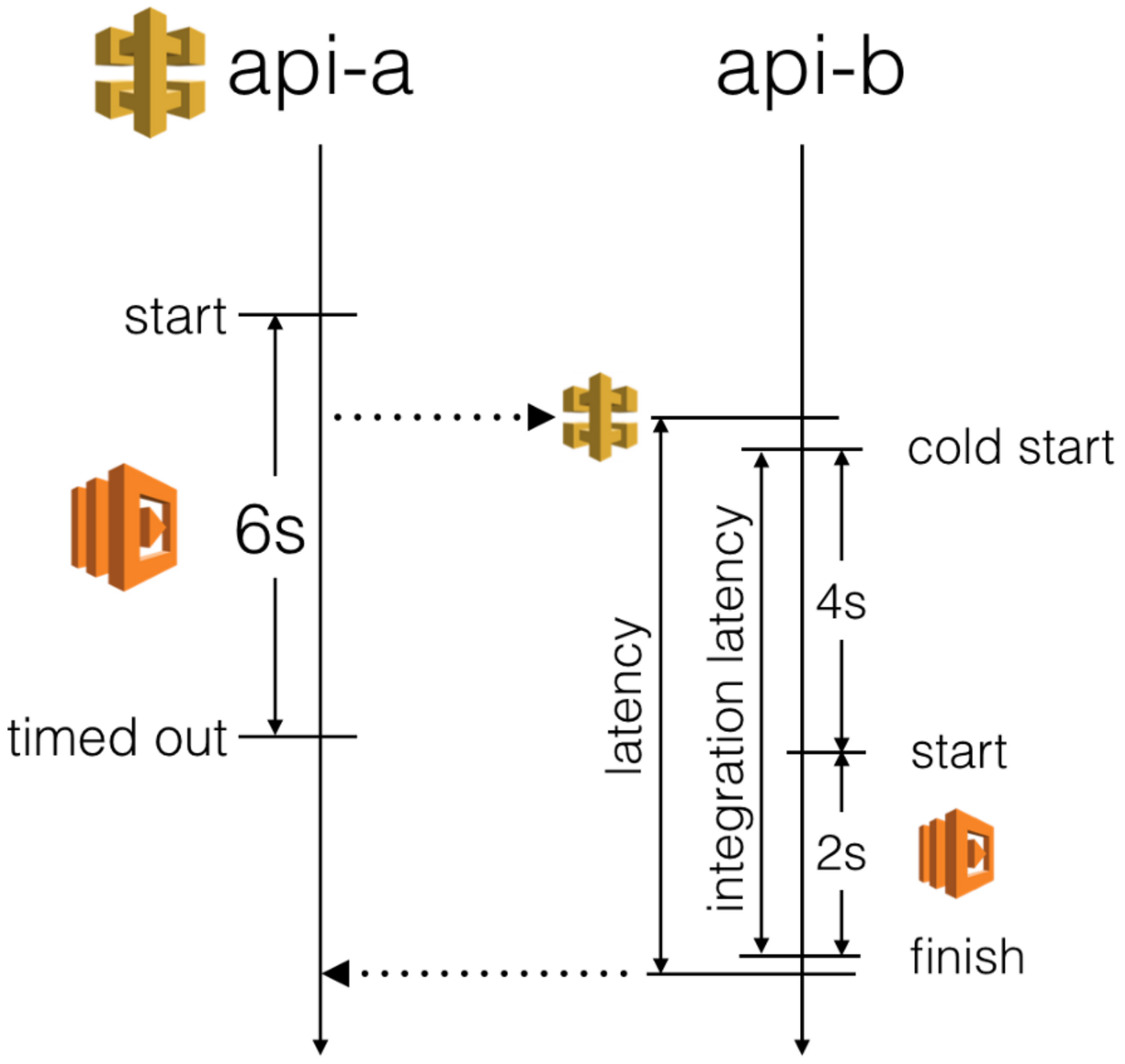
hypothesize steady state will continue in both control group & the experiment group

ie. you should have a reasonable degree of confidence the system would handle the failure before you proceed with the experiment

API Gateway

API caching TTL	300 seconds by default and configurable between 0 and 3600 by an API owner.	Not for the upper bound (3600)
Integration timeout	29 seconds for all integration types, including Lambda, Lambda proxy, HTTP, HTTP proxy, and AWS integrations.	No
Header value size	10240 Bytes	No
Payload size	10 MB	No

consider the effect of cold-starts
& API Gateway overhead

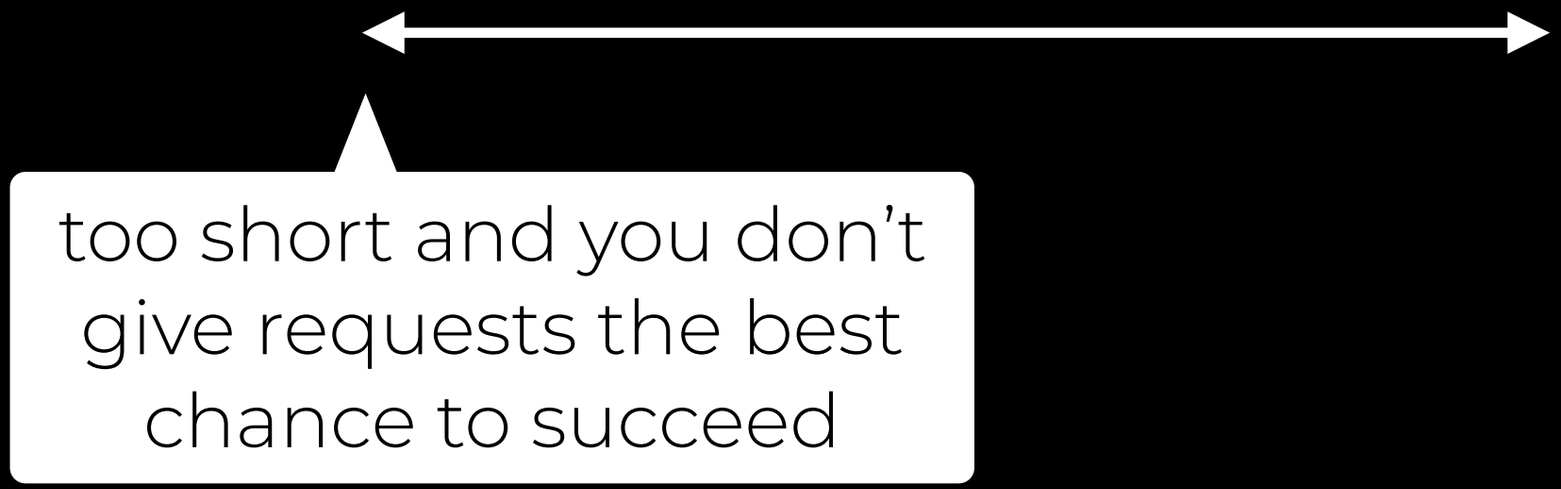


use short timeout for API calls

the goal of a timeout strategy is to give HTTP requests the **best chance to succeed**, provided that doing so does not cause the calling function itself to err

fixed timeout are tricky to get right...

fixed timeout are tricky to get right...



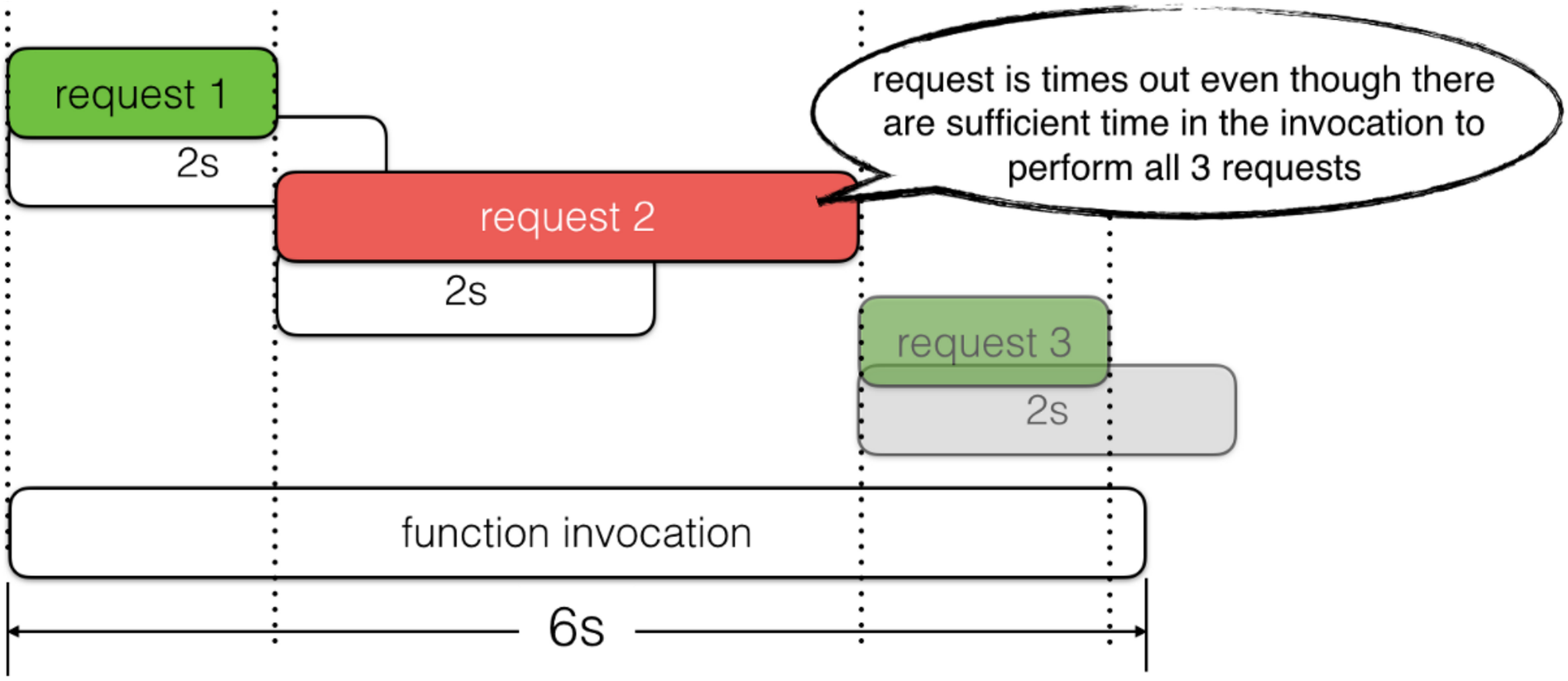
fixed timeout are tricky to get right...

too long and you run the
risk of letting the request
timeout the calling function

and it gets worse when you make multiple API calls in one function...

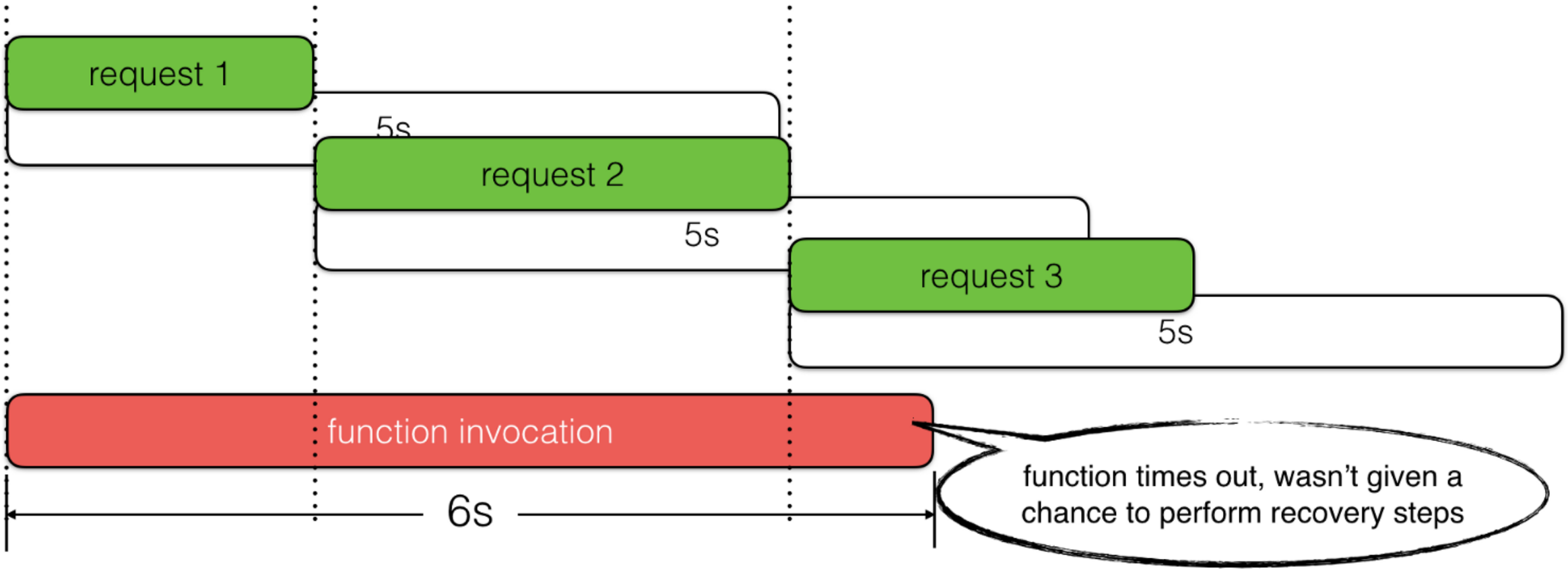
approach 1: split invocation time equally

(e.g. 3 requests, 6s function timeout = 2s timeout per request)



approach 2: every request is given nearly ALL the invocation time (free for all)

(e.g. 6s function timeout = 5s timeout per request)



set the request timeout based on the amount of invocation time left

The Context Object Methods (Node.js)

The context object provides the following methods.

context.getRemainingTimeInMillis()

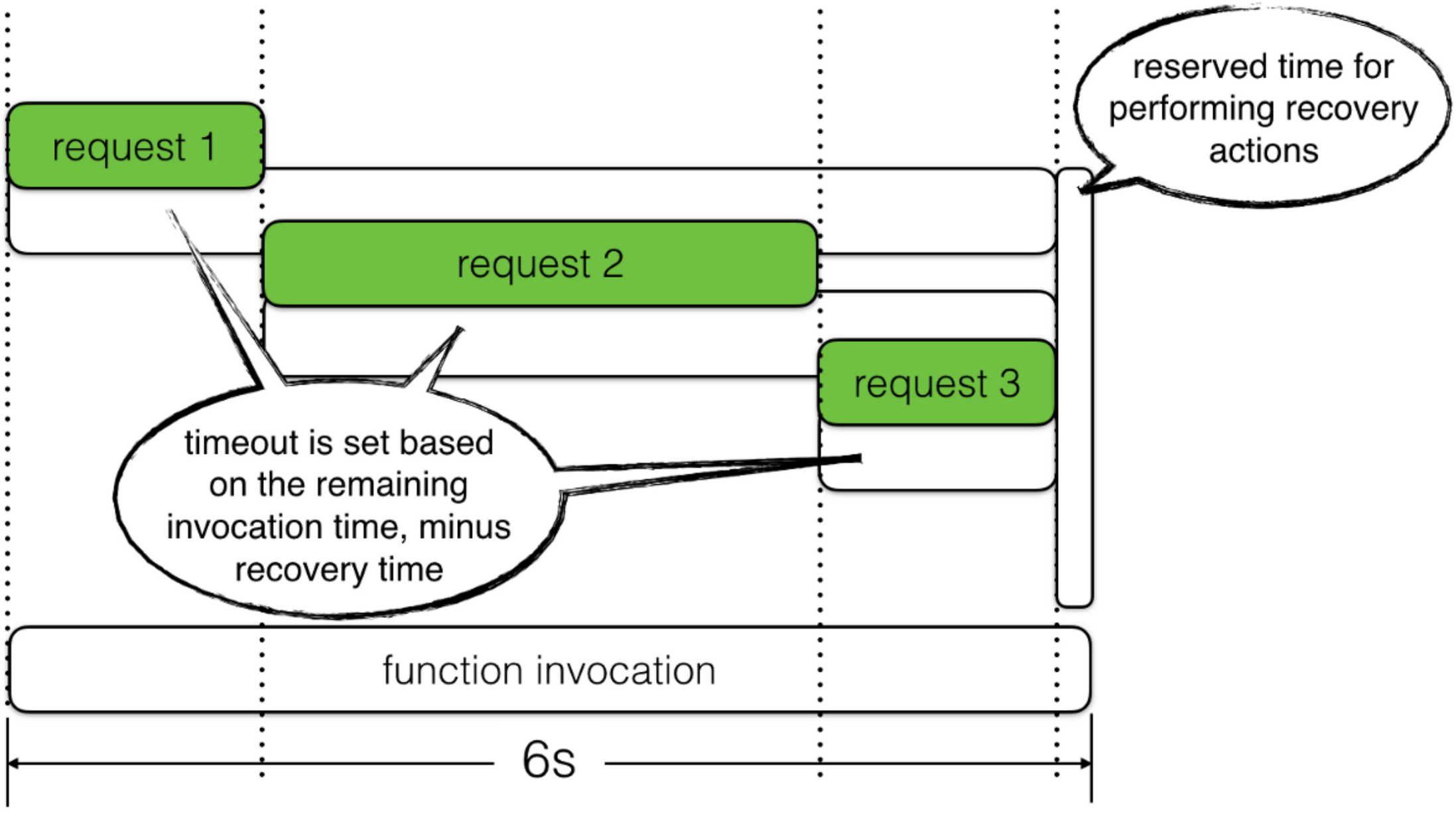
Returns the approximate remaining execution time (before timeout occurs) of the Lambda function that is currently executing. The timeout is one of the Lambda function configuration. When the timeout reaches, AWS Lambda terminates your Lambda function.

You can use this method to check the remaining time during your function execution and take appropriate corrective action at run time.

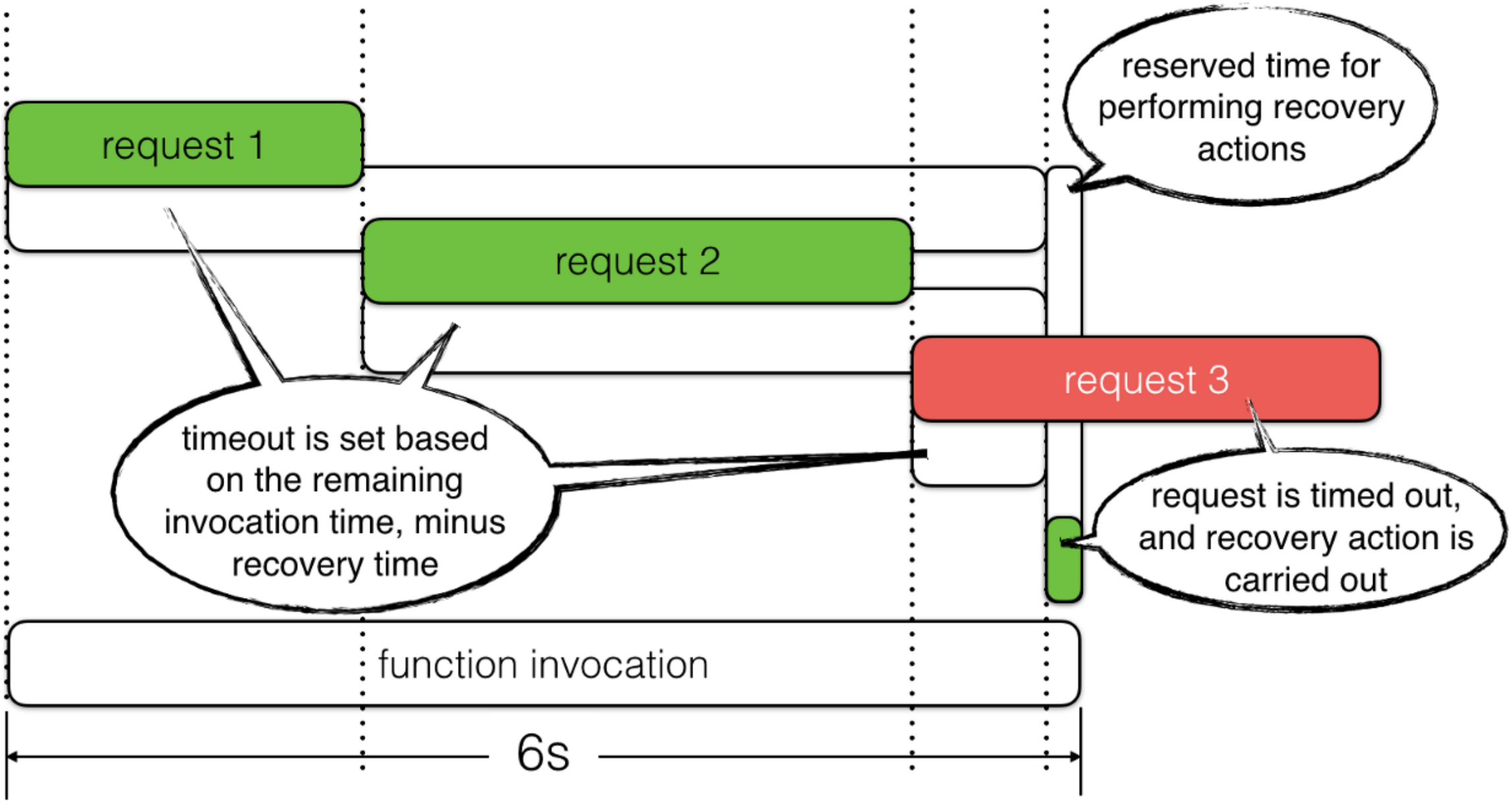
The general syntax is:

```
context.getRemainingTimeInMillis();
```

set timeout based on remaining invocation time



set timeout based on remaining invocation time




RECOVERY

log the timeout incident with as much context as possible

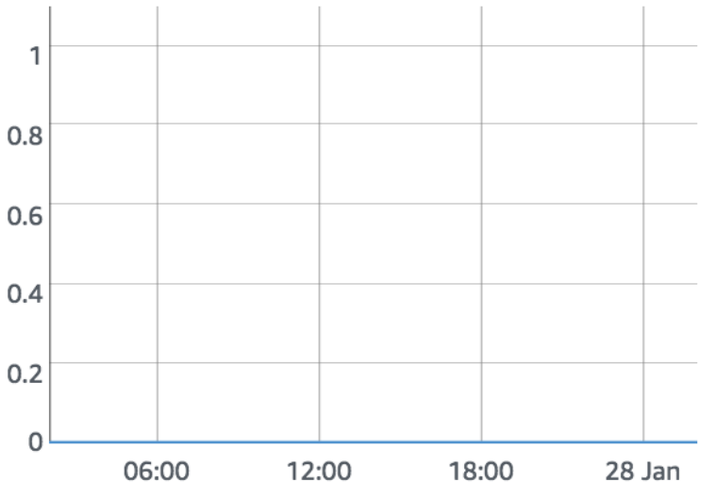
e.g. timeout value, correlation IDs, request object, ...

apig-iscoldstart-dev-hello

Qualifiers Actions Select a test event..

Invocation count Last 24 hours

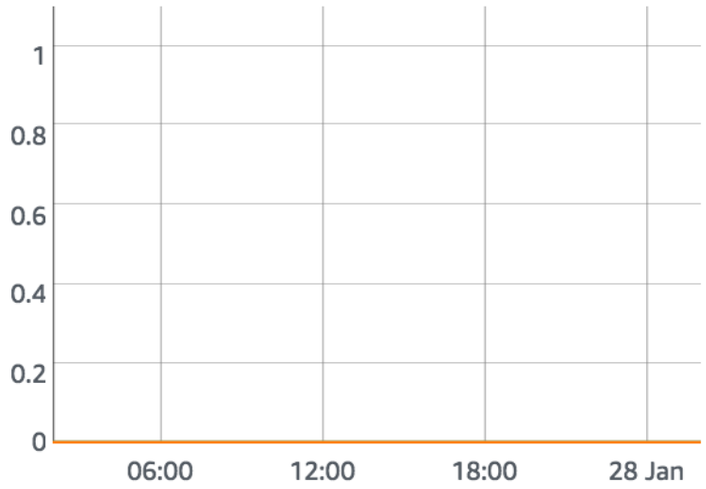
Jump to Metrics Jump to Logs



Count

Invocation duration Last 24 hours

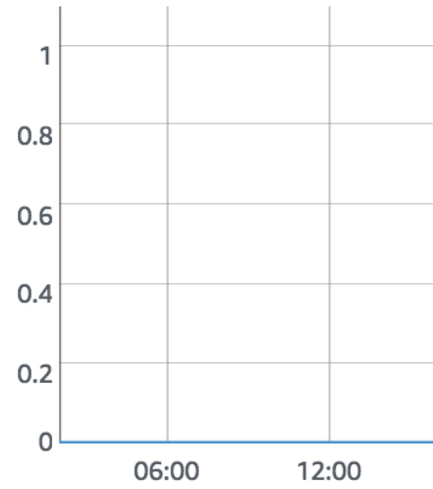
Jump to Metrics Jump to Logs



Max Milliseconds
Avg Milliseconds
Min Milliseconds

Invocation errors

Jump to Metrics

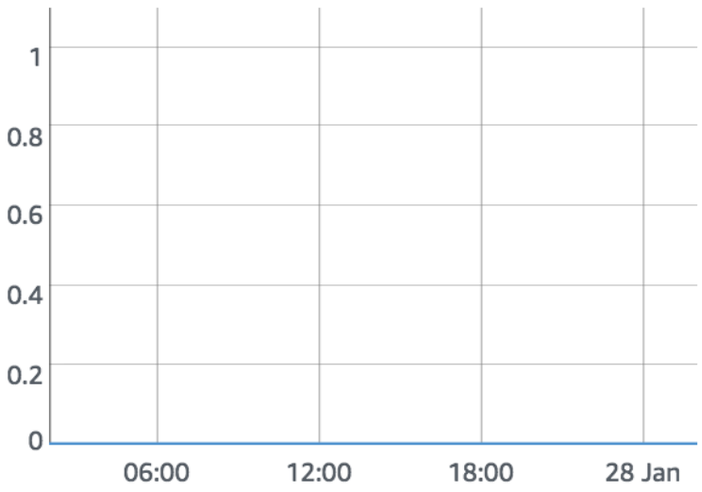


Count

report custom metrics

Throttled invocations Last 24 hours

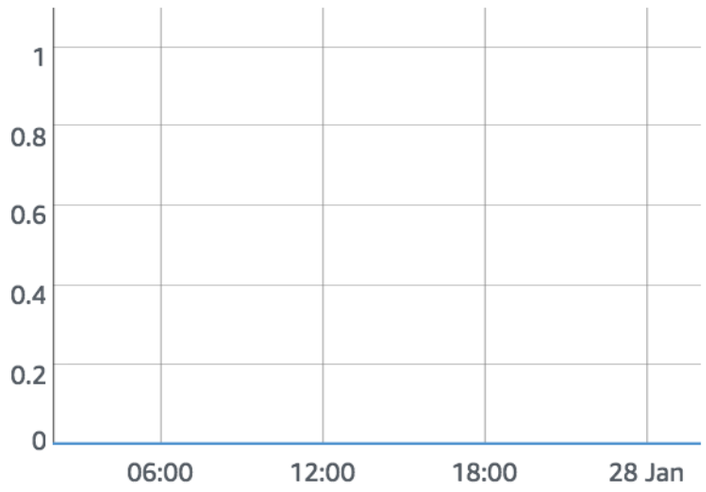
Jump to Metrics Jump to Logs



Count

Iterator age Last 24 hours

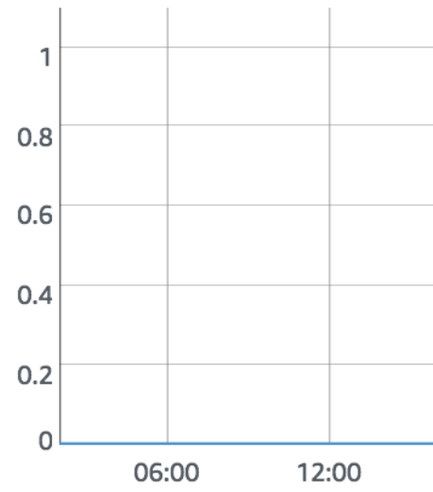
Jump to Metrics Jump to Logs



Max Milliseconds

DLQ errors

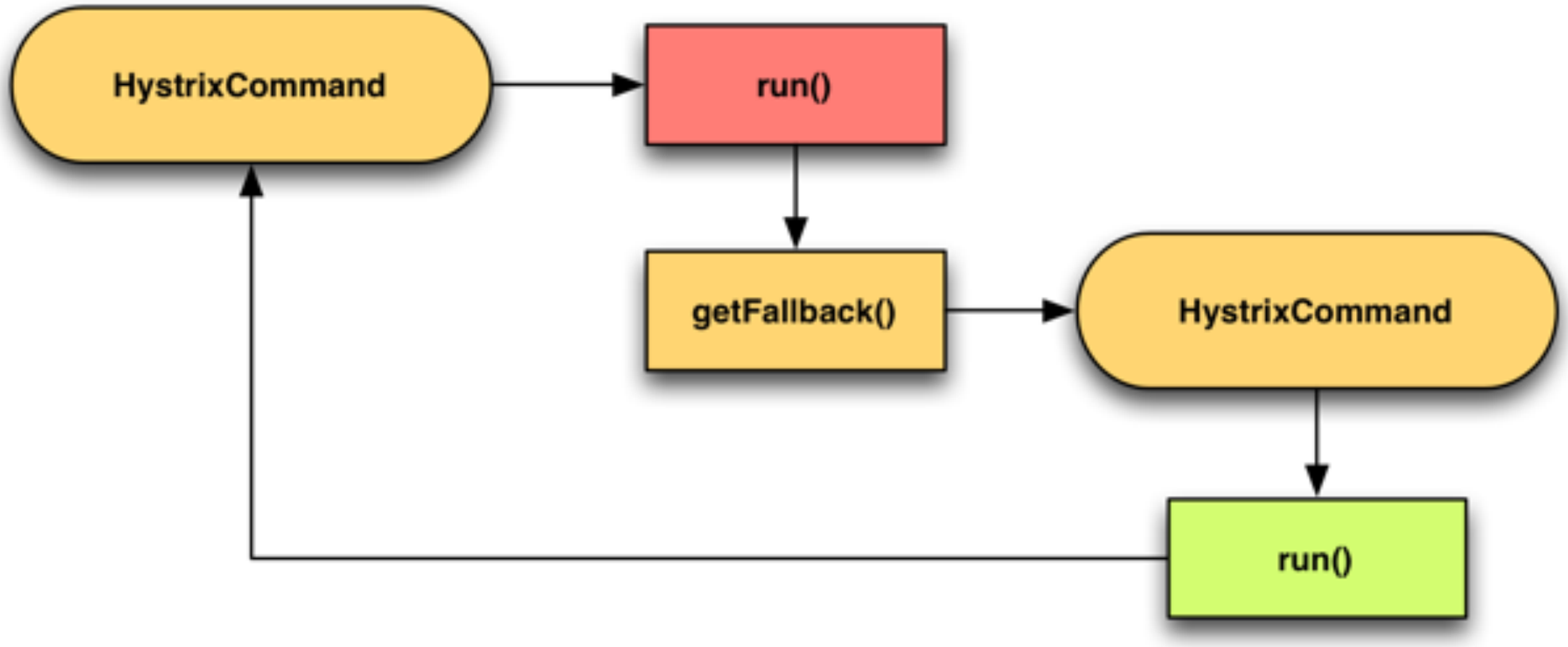
Jump to Metrics



Count

Hystrix

Defend Your App



NETFLIX

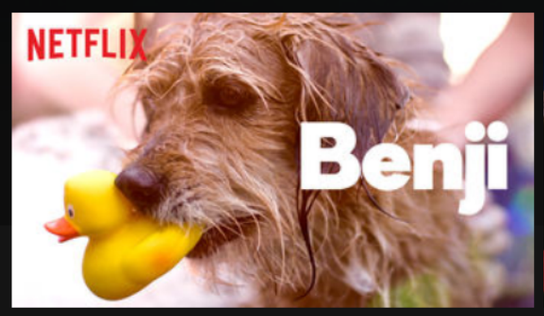
Home TV Programmes Films Originals Recently Added My List

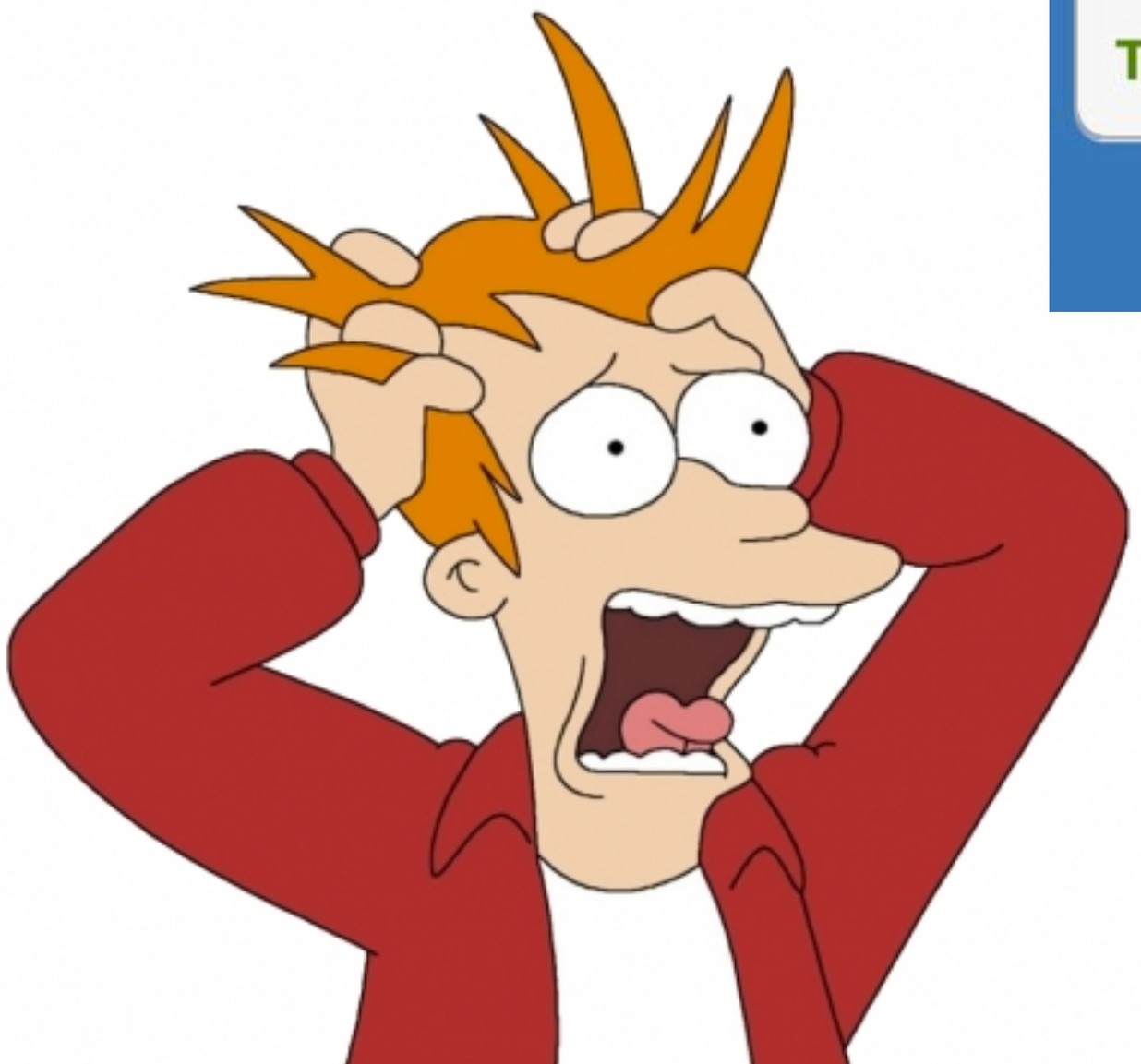
NETFLIX ORIGINAL

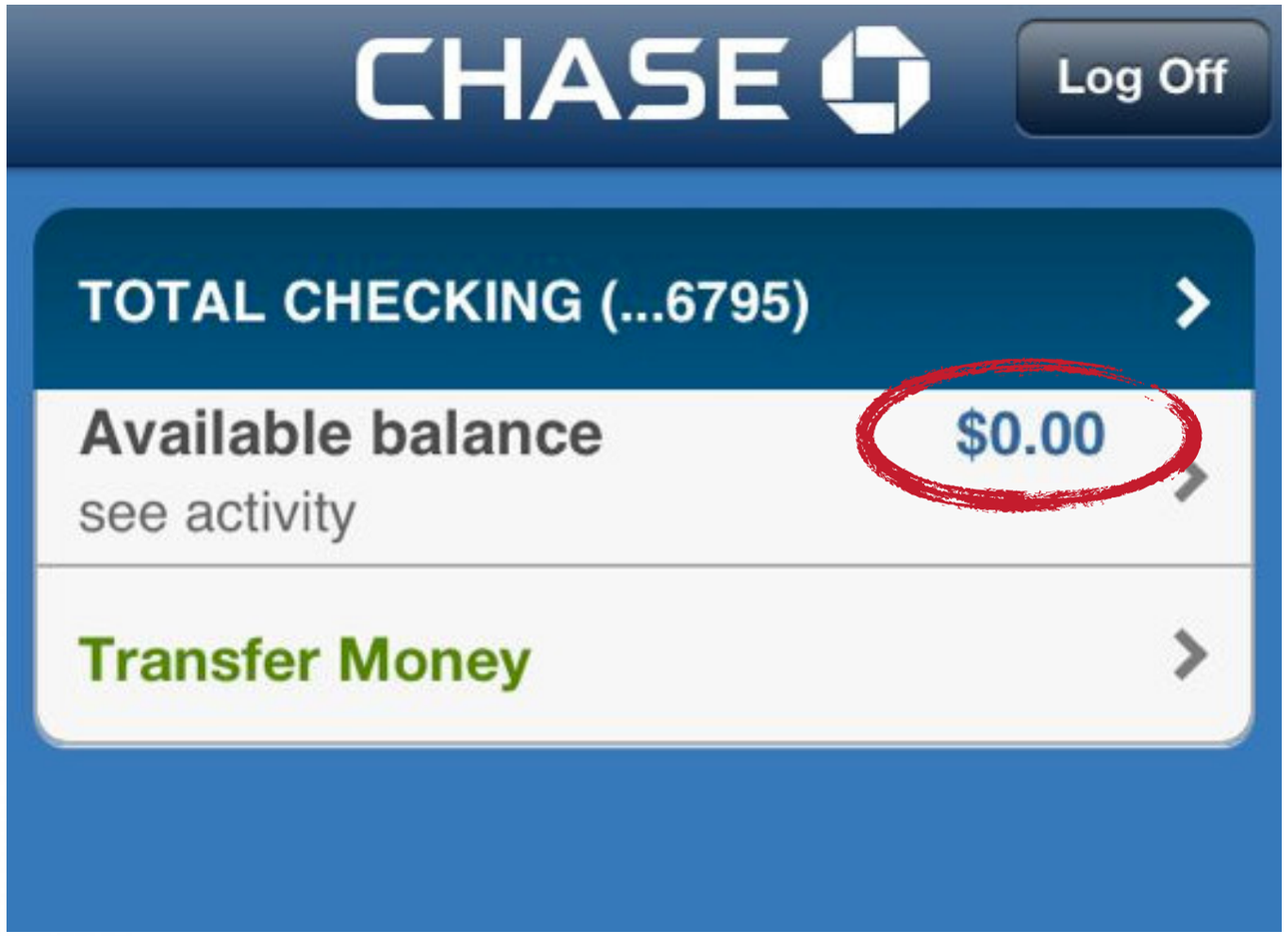
RICKY GERVAIS HUMANITY

Play + My List

Continue Watching for Yan






CHASE
Log Off
TOTAL CHECKING (...6795)
Available balance
$0.00
see activity
Transfer Money

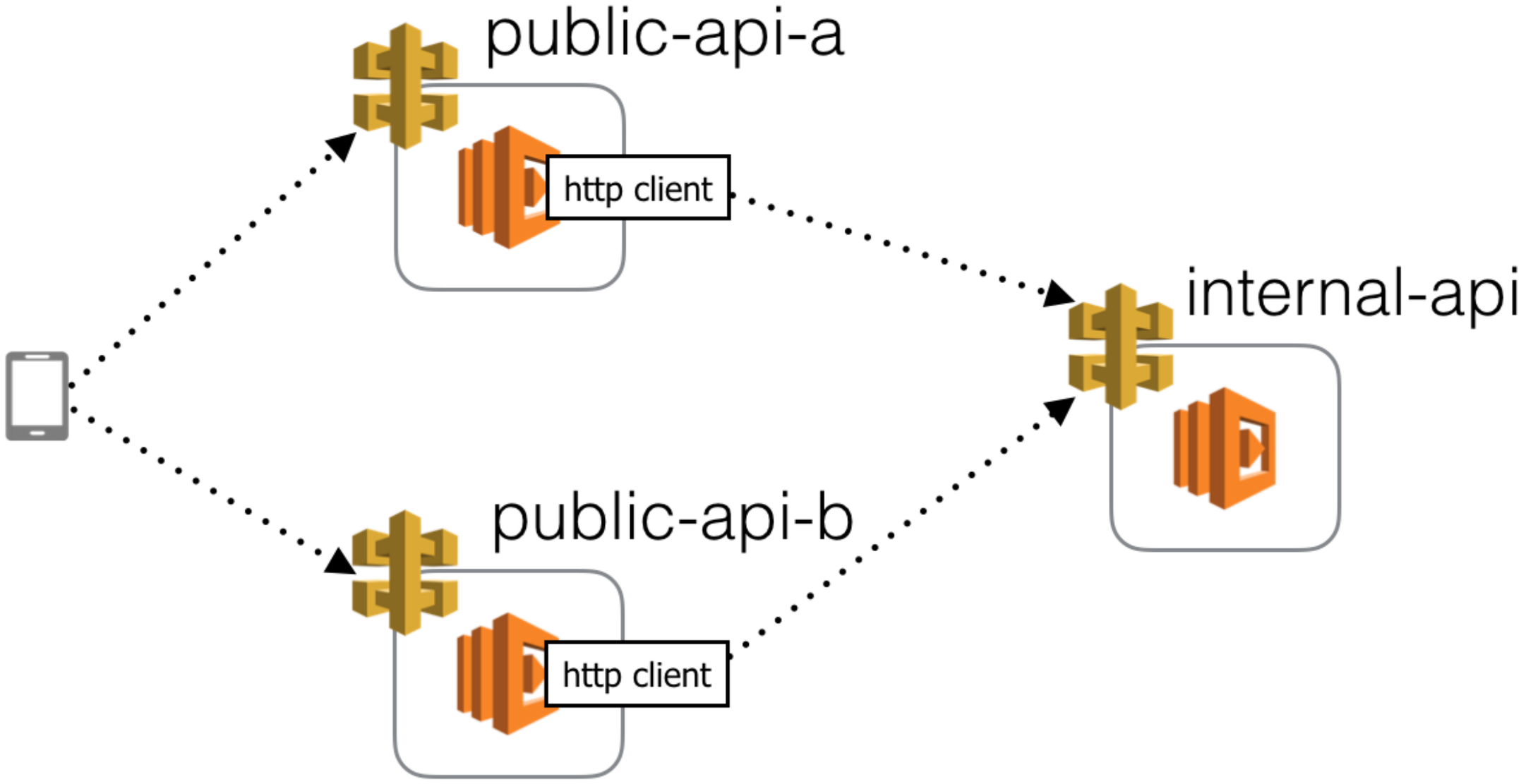
be mindful when you sacrifice precision for availability, **user experience is the king**

STEP 3.

inject realistic failures

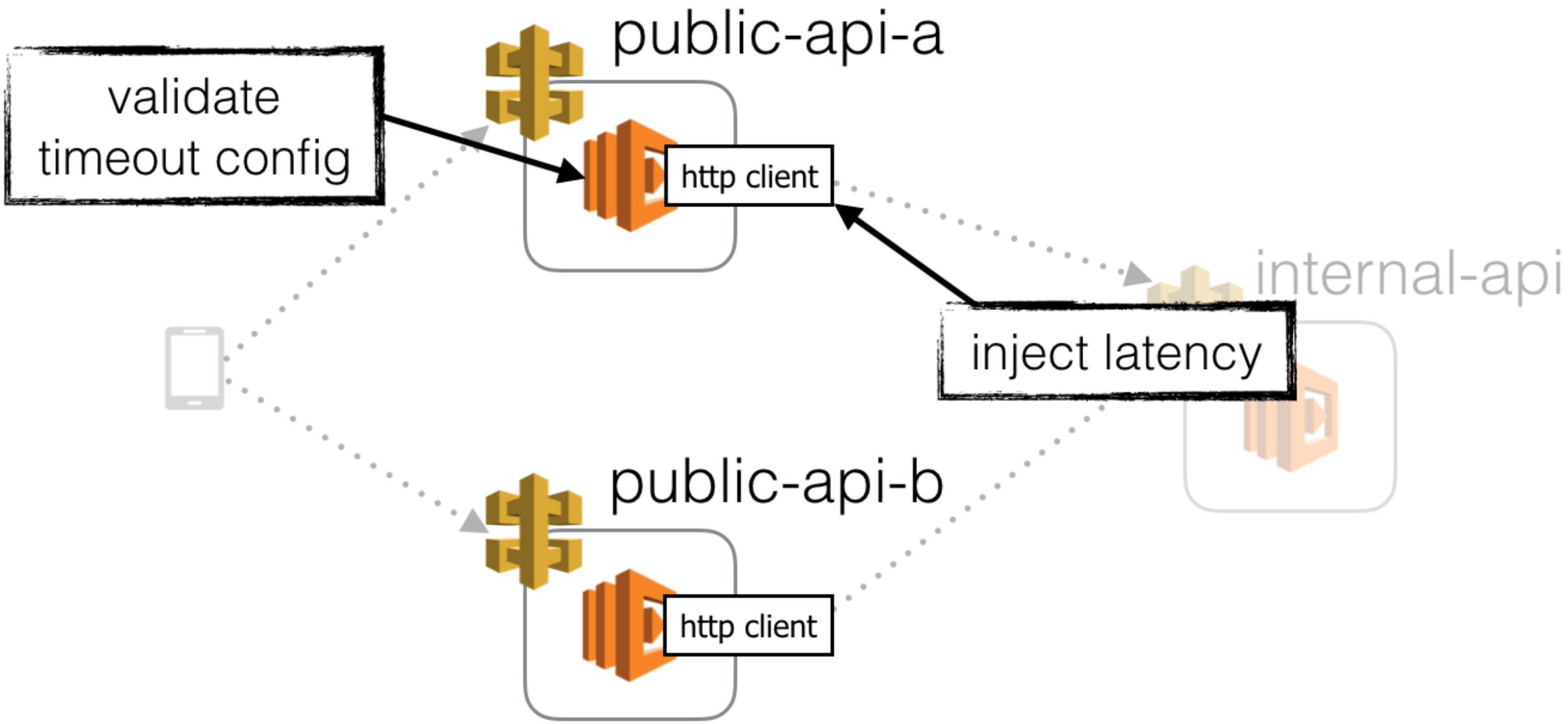
e.g. server crash, network error,
HD malfunction, etc.

where to inject latency?

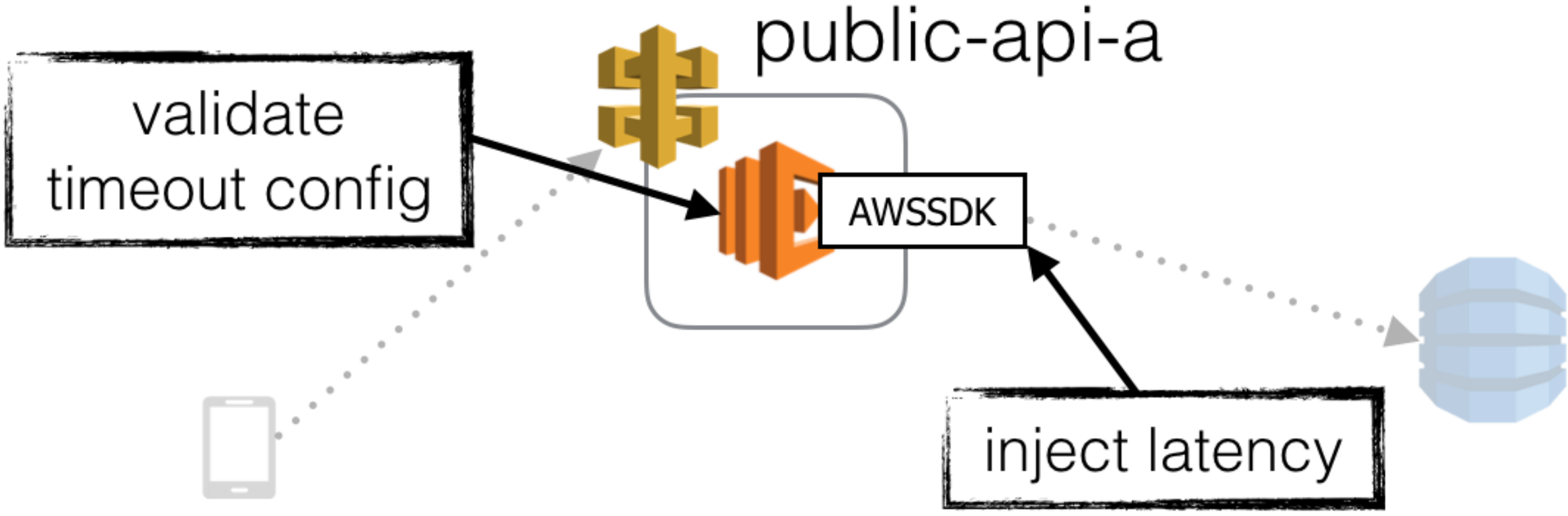


hypothesis:

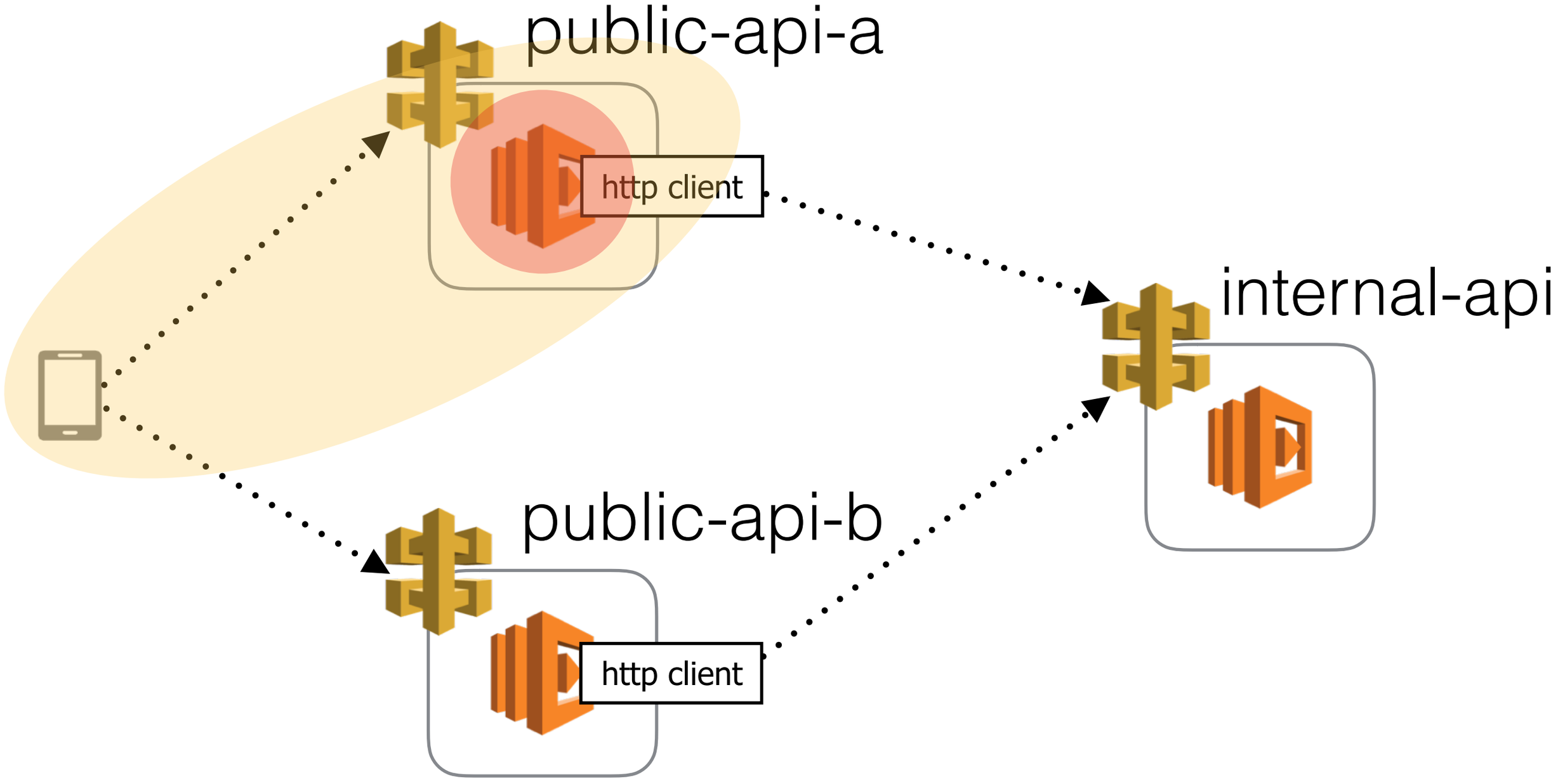
function has appropriate timeout on its HTTP communications and can degrade gracefully when these requests time out


public-api-a
validate
timeout config
http client
inject latency
internal-api
public-api-b
http client

should also be applied to 3rd parties
services we depend on, e.g. DynamoDB

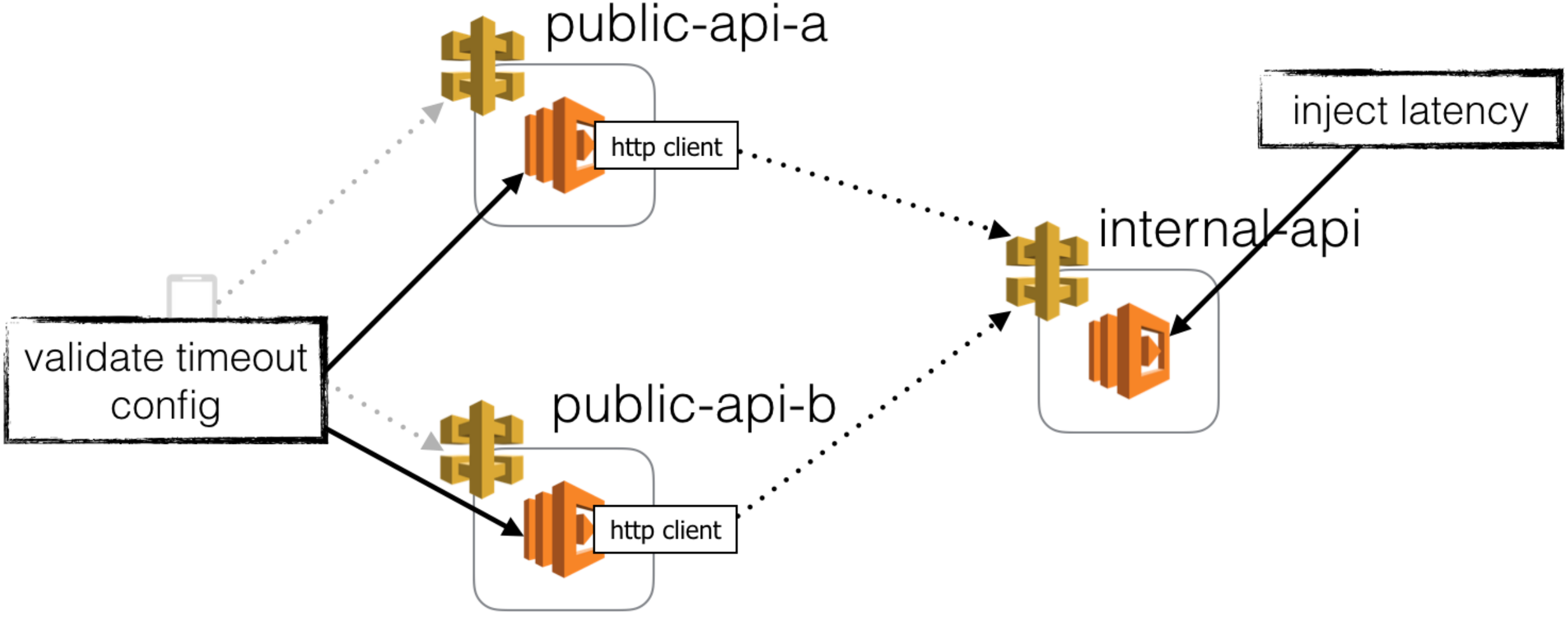

public-api-a
validate
timeout config
AWSSDK
inject latency

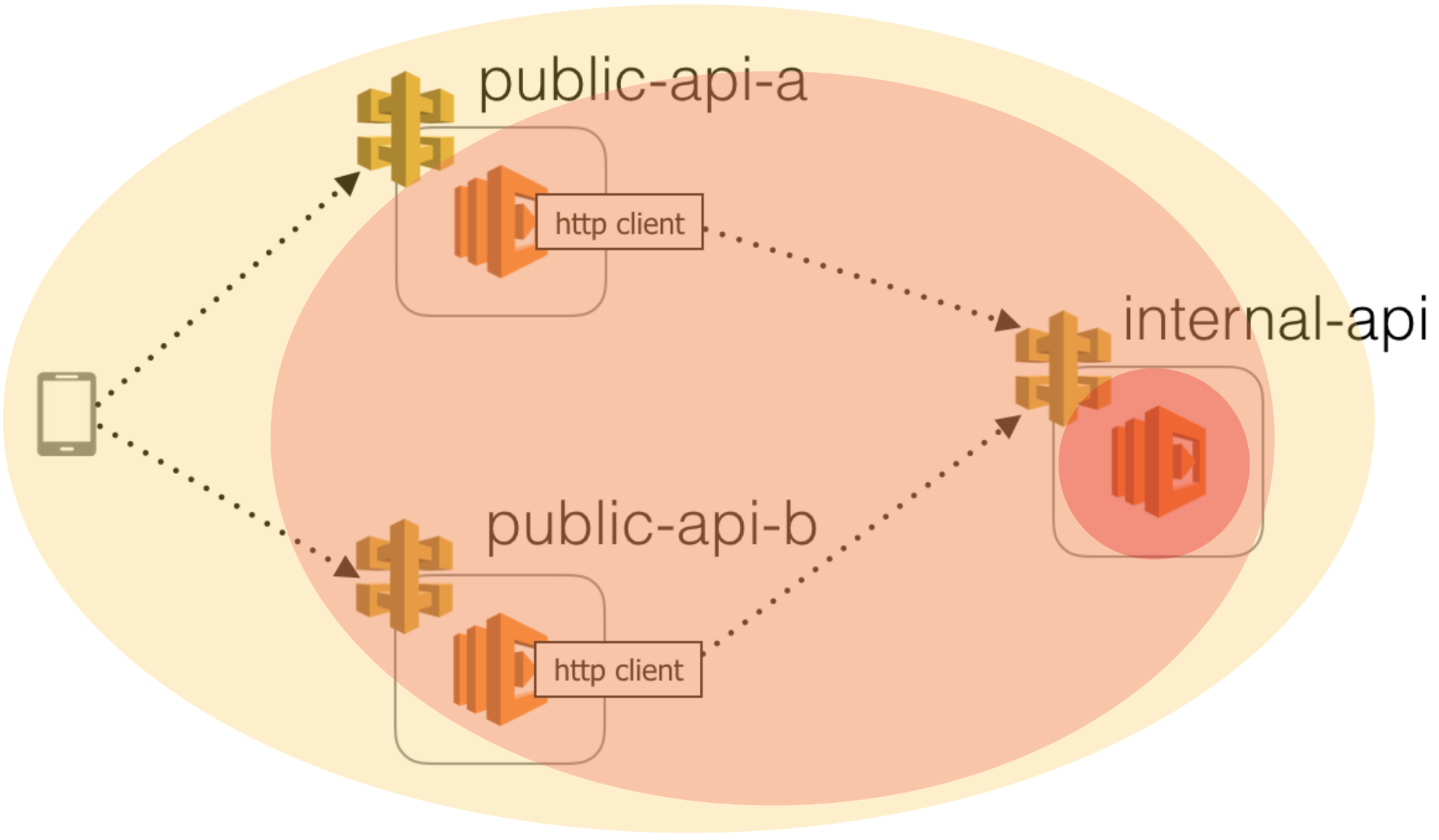
what’s the blast radius?


public-api-a
http client
internal-api
public-api-b
http client

hypothesis:

all functions have appropriate timeout on their HTTP communications to this internal API, and can degrade gracefully when requests are timed out


public-api-a
http client
inject latency
internal-api
validate timeout config
public-api-b
http client


public-api-a
http client
internal-api
public-api-b
http client

large blast radius, risky..

could be effective when used **away from production environment**, to weed out weaknesses quickly

not priming developers to
build more resilient systems


development

development

production

Priming (psychology):

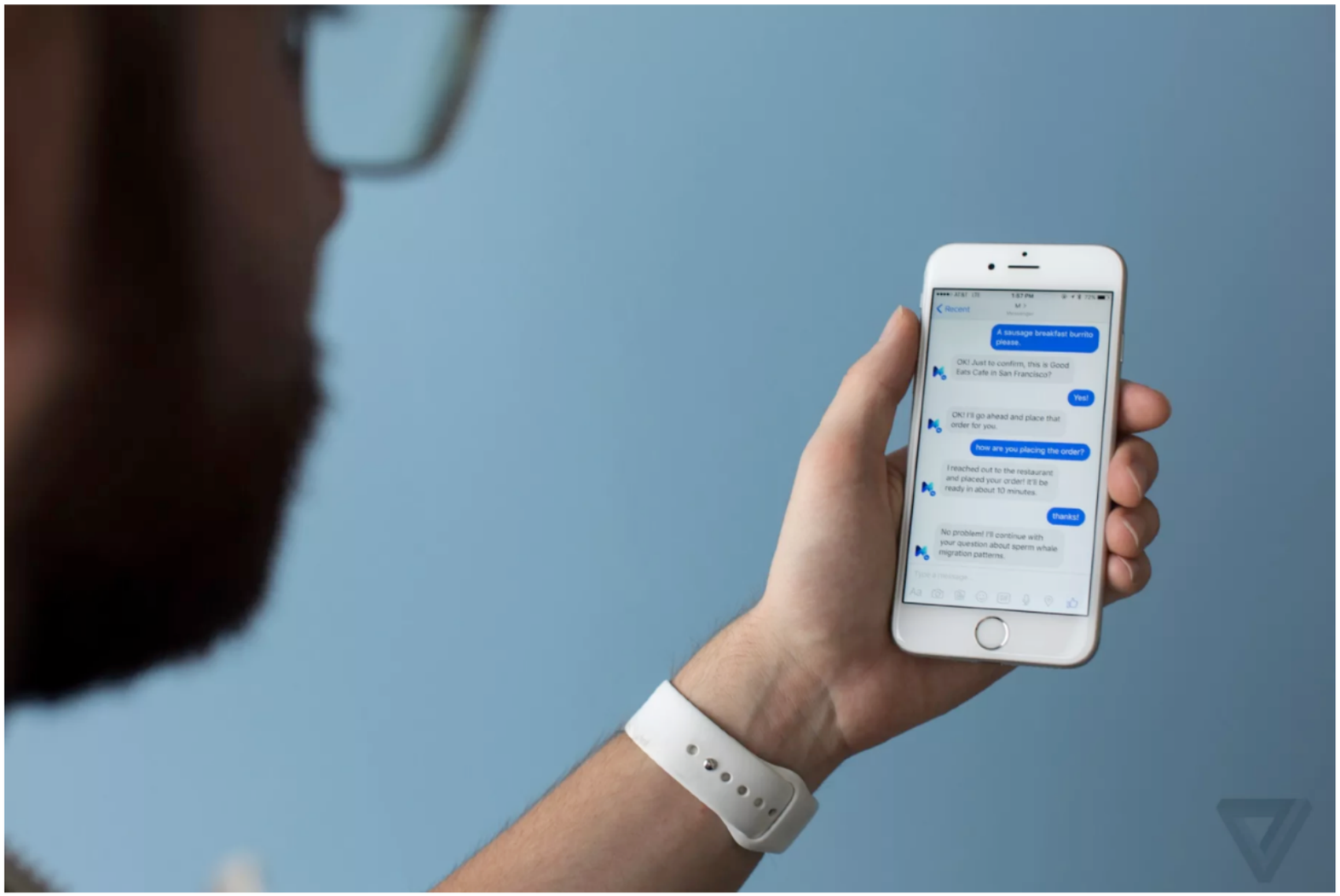
Priming is a technique whereby exposure to one stimulus influences a response to a subsequent stimulus, without conscious guidance or intention.

It is a technique in psychology used to train a person's memory both in positive and negative ways.

Facebook's '2G Tuesdays' simulate super slow internet in the developing world

By Rich McCormick | Oct 28, 2015, 5:58am EDT

SHARE

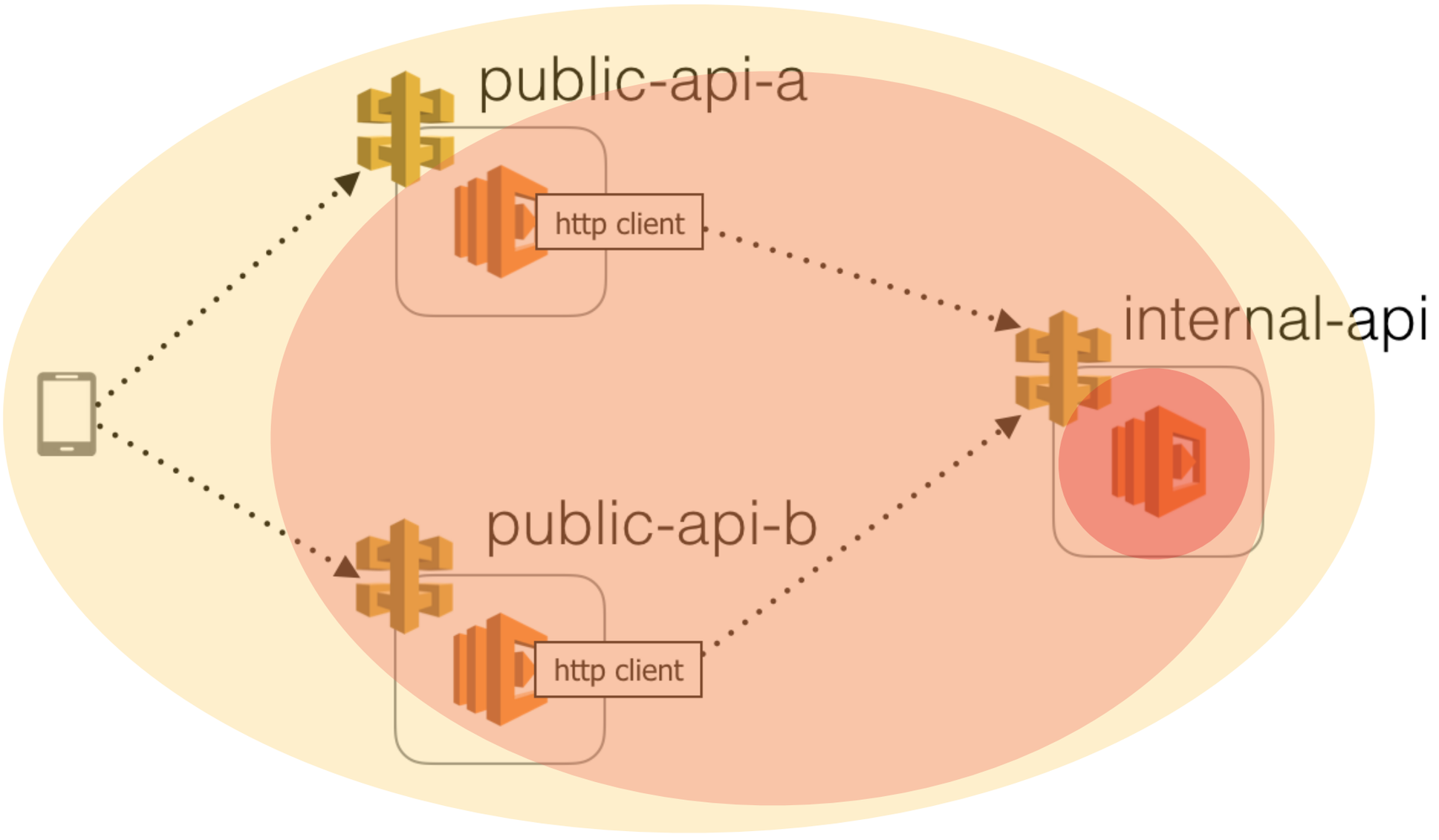


In an attempt to understand the millions of people in emerging markets who only have access to slow internet connections, Facebook is adopting a new opt-in initiative called "2G

NOW TRENDING

China will ban people with poor 'social credit' from planes and trains

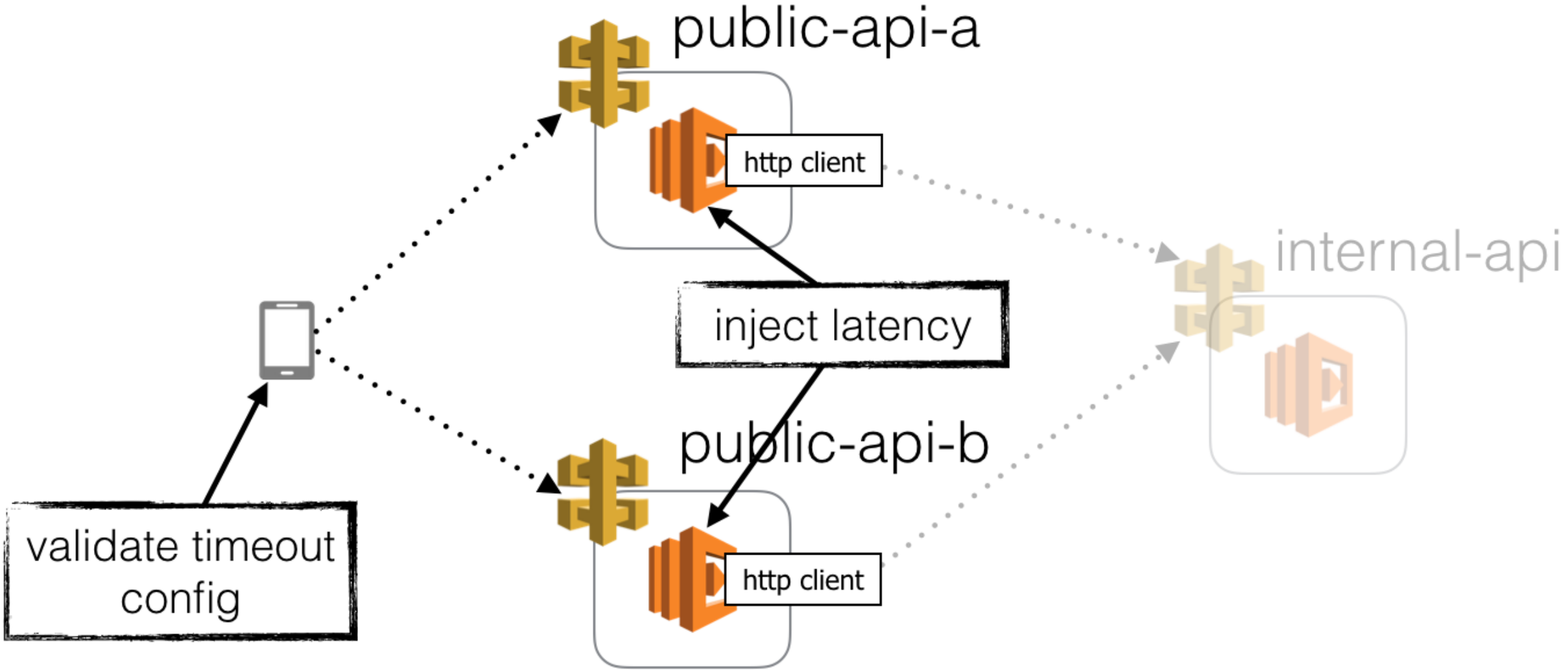
The final trailer for Avengers: Infinity War is here

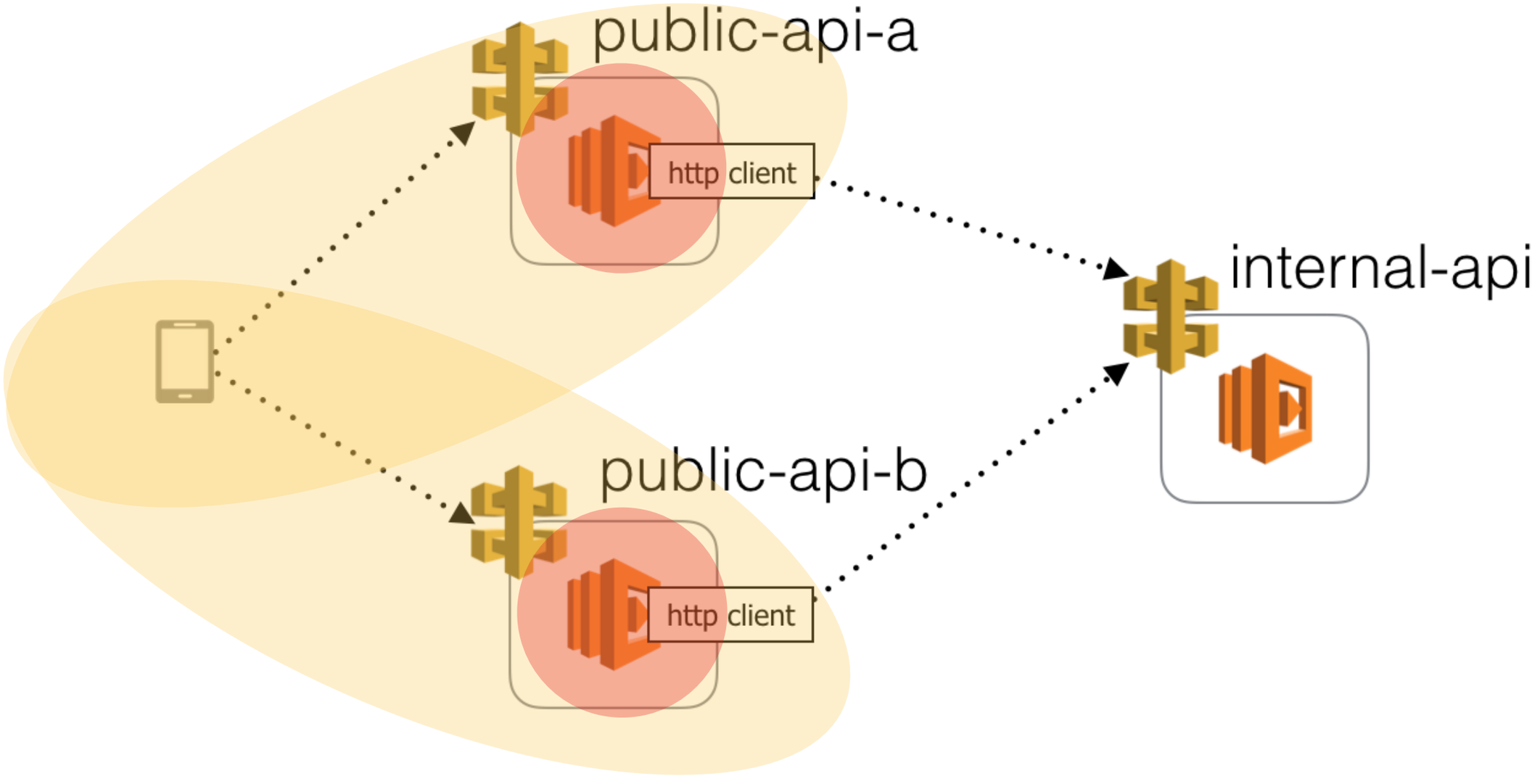

public-api-a
http client
internal-api
public-api-b
http client

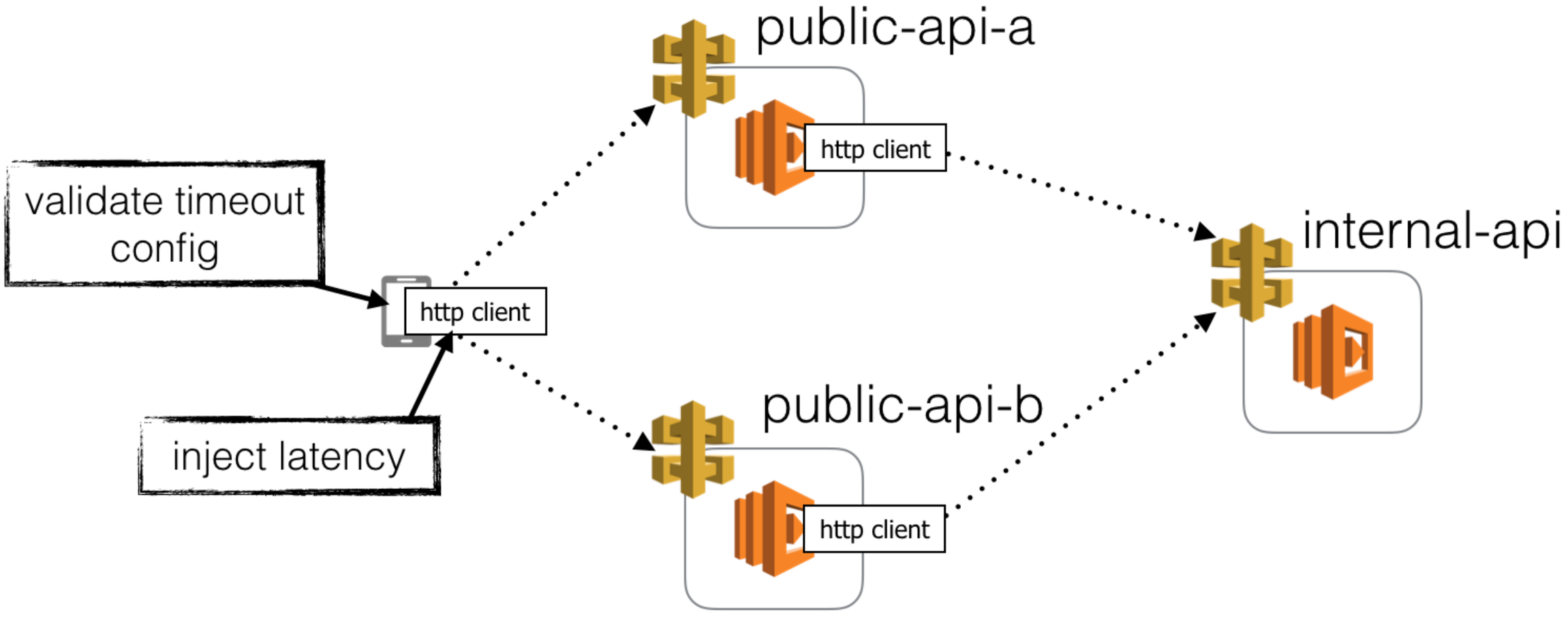
make dev environments better resemble the turbulent conditions you should realistically expect your system to survive in production

hypothesis:

the client app has appropriate timeout on their HTTP communication with the server, and can degrade gracefully when requests are timed out


public-api-a
http client
inject latency
internal-api
public-api-b
http client
validate timeout config


public-api-a
http client
internal-api
public-api-b
http client


public-api-a
validate timeout config
http client
http client
internal-api
public-api-b
inject latency
http client

STEP 4.

disprove hypothesis

i.e. look for difference with steady state

how to inject latency?

static weaver (e.g. AspectJ, PostSharp),
or dynamic proxies

manually crafted wrapper library

```
const co      = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function* (config) {
  if (config.isEnabled === true && Math.random() < config.probability) {
    let delayRange = config.maxDelay - config.minDelay;
    let delay = Math.floor(config.minDelay + Math.random() * delayRange);

    console.log(`injecting [${delay}ms] latency to HTTP request...`);
    yield Promise.delay(delay);
  }
});
```

```
const co      = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function* (config) {
  if (config.isEnabled === true && Math.random() < config.probability) {
    let delayRange = config.maxDelay - config.minDelay;
    let delay = Math.floor(config.minDelay + Math.random() * delayRange);

    console.log(`injecting [${delay}ms] latency to HTTP request...`);
    yield Promise.delay(delay);
  }
});
```

```
const co      = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function* (config) {
  if (config.isEnabled === true && Math.random() < config.probability) {
    let delayRange = config.maxDelay - config.minDelay;
    let delay = Math.floor(config.minDelay + Math.random() * delayRange);

    console.log(`injecting [${delay}ms] latency to HTTP request...`);
    yield Promise.delay(delay);
  }
});
```

```
let Req = function* (options) {
  let request = getRequest(options);
  let fullResponse = options.resolveWithFullResponse === true;

  let latencyInjectionConfig = options.latencyInjectionConfig;
  yield injectLatency(latencyInjectionConfig);  // <- this is the import bit

  try {
    let resp = yield exec(request);
    return fullResponse ? resp : resp.body;
  } catch (e) {
    if (e.response && e.response.error) {
      throw e.response.error;
    }

    throw e;
  }
};
```

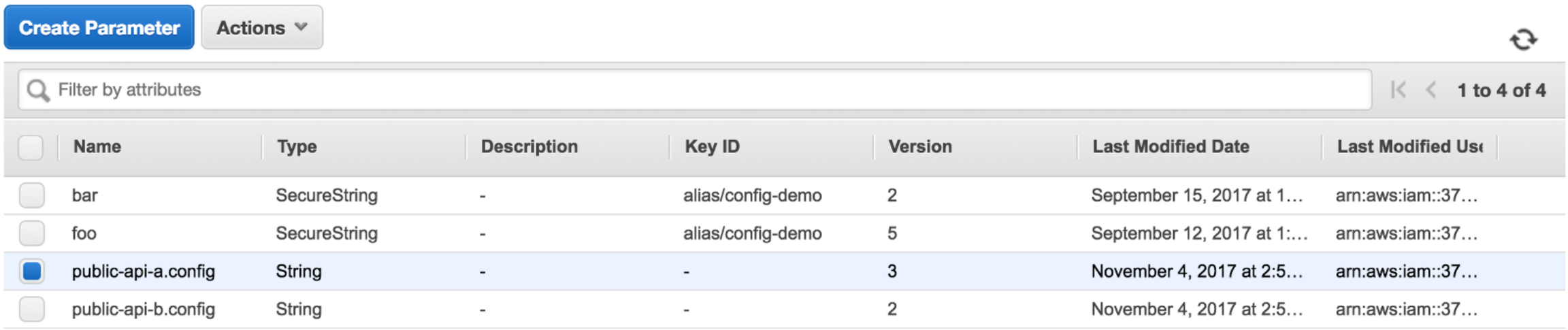
```
let Req = function* (options) {
  let request = getRequest(options);
  let fullResponse = options.resolveWithFullResponse === true;

  let latencyInjectionConfig = options.latencyInjectionConfig;
  yield injectLatency(latencyInjectionConfig);  // <- this is the import bit

  try {
    let resp = yield exec(request);
    return fullResponse ? resp : resp.body;
  } catch (e) {
    if (e.response && e.response.error) {
      throw e.response.error;
    }

    throw e;
  }
};
```

configured in SSM Parameter Store




```
{
  "internalApi": "https://xx.amazonaws.com/dev/internal",
  "chaosConfig": {
    "httpClientLatencyInjectionConfig": {
      "isEnabled": true,
      "probability": 0.5,
      "minDelay": 100,
      "maxDelay": 5000
    }
  }
}
```

```
{
  "internalApi": "https://xx.amazonaws.com/dev/i
  "chaosConfig": {
    "httpClientLatencyInjectionConfig": {
      "isEnabled": true,
      "probability": 0.5,
      "minDelay": 100,
      "maxDelay": 5000
    }
  }
}
```

no injected latency

Timeline | Raw data

Method	Response	Duration	Age	ID
--	200	81.0 ms	8.6 sec (2017-11-04 15:54:25 UTC)	1-59fde2b1-6e08594037ba472dfeca7a3e

Name	Res.	Duration	Status	0.0ms	10ms	20ms	30ms	40ms	50ms	60ms	70ms	80ms	90ms
latency-injection-demo-dev-public-api-a AWS::Lambda													
latency-injection-demo-dev-public-api-a	200	81.0 ms	✓										
latency-injection-demo-dev-public-api-a AWS::Lambda::Function													
latency-injection-demo-dev-public-api-a	-	74.0 ms	✓										
HTTP client	-	59.0 ms	✓										
## exec request	-	58.0 ms	✓										

Timeline | Raw data

Method	Response	Duration	Age	ID
--	200	81.0 ms	8.6 sec (2017-11-04 15:54:25 UTC)	1-59fde2b1

Name	Res.	Duration	Status	0.0ms	10ms	20ms
latency-injection-demo-dev-public-api-a AWS::Lambda						
latency-injection-demo-dev-public-api-a	200	81.0 ms	✓			
latency-injection-demo-dev-public-api-a AWS::Lambda::Function						
latency-injection-demo-dev-public-api-a	-	74.0 ms	✓			
HTTP client	-	59.0 ms	✓			
## exec request	-	58.0 ms	✓			

with injected latency

Timeline | Raw data

Method	Response	Duration	Age	ID
--	200	3.5 sec	10.5 sec (2017-11-04 15:57:08 UTC)	1-59fde354-6b0784e851c3c43729368ae1

Name	Res.	Duration	Status	0.0ms	500ms	1.0s	1.5s	2.0s	2.5s	3.0s	3.5s
latency-injection-demo-dev-public-api-a AWS::Lambda											
latency-injection-demo-dev-public-api-a	200	3.5 sec	✓								
latency-injection-demo-dev-public-api-a AWS::Lambda::Function											
latency-injection-demo-dev-public-api-a	-	3.5 sec	✓								
HTTP client	-	3.5 sec	✓								
## latency injection	-	3.4 sec	✓								
## exec request	-	63.0 ms	✓								

Timeline | Raw data

Method	Response	Duration	Age	ID
--	200	3.5 sec	10.5 sec (2017-11-04 15:57:08 UTC)	1-59fde

Name	Res.	Duration	Status	0.0ms	500ms
▼ **latency-injection-demo-dev-public-api-a** AWS::Lambda					
latency-injection-demo-dev-public-api-a	200	3.5 sec	✓		
▼ **latency-injection-demo-dev-public-api-a** AWS::Lambda::Function					
latency-injection-demo-dev-public-api-a	-	3.5 sec	✓		
HTTP client	-	3.5 sec	✓		
## latency injection	-	3.4 sec	✓		
## exec request	-	63.0 ms	✓		

factory wrapper function

(think bluebird's *promisifyAll* function)

```
const co          = require('co');
const apiHandler = require('../lib/apiHandler');
const injectable = require('../lib/injectable');
const Promise     = require('bluebird');
const AWS         = require('aws-sdk');
const asyncDDB    = Promise.promisifyAll(new AWS.DynamoDB.DocumentClient());
const dynamodb    = injectable.injectableAll(asyncDDB);
```

```
let req = {
  TableName : 'latency-injection-demo-dev',
  Key: { id: 'foo' }
};

// the wrapped (and promisified) DocumentClient's async functions can
// take an optional argument (the last argument) which is expect to be
// the format of the latencyInjectionConfig, ie. of the shape
// { isEnabled, probability, minDelay, maxDelay }
let item = yield dynamodb.getAsync(req, latencyInjectionConfig);
```

▶	22:00:56	2017-11-04T22:00:56.715Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_DAEMON_ADDRESS is
▶	22:00:56	2017-11-04T22:00:56.719Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_CONTEXT_MISSING is
▶	22:00:56	2017-11-04T22:00:56.731Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f Subsegment streaming threshold se
▶	22:00:56	START RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f Version: $LATEST
▶	22:00:56	2017-11-04T22:00:56.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f {"resource":"/public-b","path":"/pub
▶	22:00:56	2017-11-04T22:00:56.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f loading cache keys: public-api-b.co
▶	22:00:56	2017-11-04T22:00:56.875Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f successfully loaded cache keys: pul
▼	22:00:56	2017-11-04T22:00:56.920Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f injecting [1366ms] latency to wrapp

```
2017-11-04T22:00:56.920Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f injecting [1366ms] latency to wrapped function...
```

▶	22:00:58	2017-11-04T22:00:58.294Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f SUCCESS {"message":"everything i
▶	22:00:58	END RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f
▶	22:00:58	REPORT RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f Duration: 1534.60 ms Billed Duration: 1600

.715Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_DAEMON_ADDRESS is
.719Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_CONTEXT_MISSING is
.731Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f Subsegment streaming threshold se
ca570a-c1ab-11e7-9c5a-7fad5899050f Version: $LATEST
.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f {"resource":"/public-b","path":"/pub
.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f loading cache keys: public-api-b.co
.875Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f successfully loaded cache keys: pu
.920Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f injecting [1366ms] latency to wrapp

a-7fad5899050f injecting [1366ms] latency to wrapped function...

.294Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f SUCCESS {"message":"everything
570a-c1ab-11e7-9c5a-7fad5899050f
a1ca570a-c1ab-11e7-9c5a-7fad5899050f Duration: 1534.60 ms Billed Duration: 1600

theburningmonk / **lambda-latency-injection-demo**

Unwatch 1 | Star 1 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Demo for how to apply latency injection to Lambda functions

Edit

Add topics

6 commits | **1** branch | **0** releases | **1** contributor | MIT

Branch: **master** | New pull request | Create new file | Upload files | Find file | Clone or download

theburningmonk - it works! — Latest commit ea92e33 on 4 Nov 2017

.vscode	- added injectable helper for AWSSDK, etc.	4 months ago
examples	- added injectable helper for AWSSDK, etc.	4 months ago
functions	- it works!	4 months ago
lib	- it works!	4 months ago
.gitignore	- init commit	4 months ago
LICENSE	Initial commit	5 months ago
README.md	Initial commit	5 months ago
package-lock.json	- init commit	4 months ago
package.json	- init commit	4 months ago
serverless.yml	- added injectable helper for AWSSDK, etc.	4 months ago


ERROR INJECTION

failures are **INEVITABLE**

the only way to truly know your system's resilience against failures is to test it through **controlled** experiments

WHAT DOESN’T KILL YOU,
MAKES YOU STRONGER.

WHAT DOESN'T KILL YOU MAKES YOU STRONGER EXCEPT BEARS

BEARS WILL KILL YOU

vaccinate your serverless architecture against failures

Yan Cui

http://theburningmonk.com

@theburningmonk

DAZN

"Netflix for sports"

offices in London, Leeds, Katowice and Amsterdam