# How a scientist would improve serverless functions

Gero Vermaas, Jochem Schulenklopper

*O'Reilly Software Architecture*
*Berlin, Germany, November 7th, 2019*

Xebia


Jochem
@jschulenklopper
Gero
@gerove

# Agenda

- What was our problem?
- Why were 'traditional' QA methods less applicable?
- Investigating a scientific approach to solve it
- Introducing a (serverless) Scientist
- Experiences using Serverless Scientist
- What’s cooking in the lab today?

/whereis #everybody? · Home

https://whereis-everybody.com

# #?

A friendly Slack app that remembers where you said you were

- **Home**
- About
- Locations
- Buzz
- Technology
- Installation
- Pricing
- Support
- Privacy Policy
- Terms of Service



# /whereis #everybody?

**/whereis #everybody?** is a simple and friendly Slack app that keeps track of user-submitted locations. With that, Slack can be used to register where you are, and provides a convenient way to discover past and current locations of your Slack team members. It makes it so easy to get to know your co-worker's whereabouts...

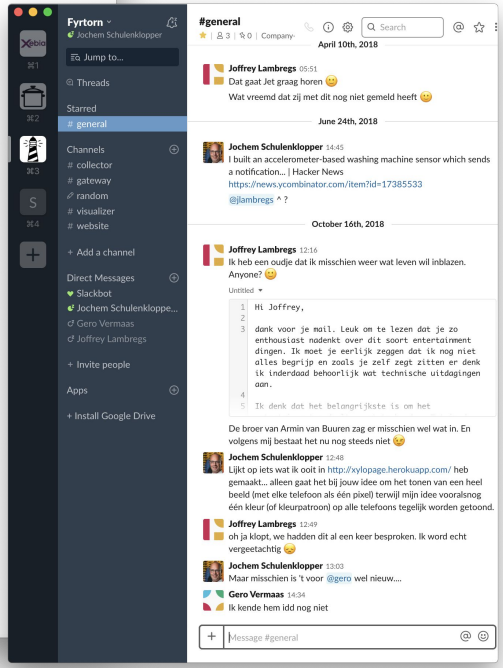
**/whereis #everybody?** can be found in the Slack App Directory. It can also be installed directly using this button:

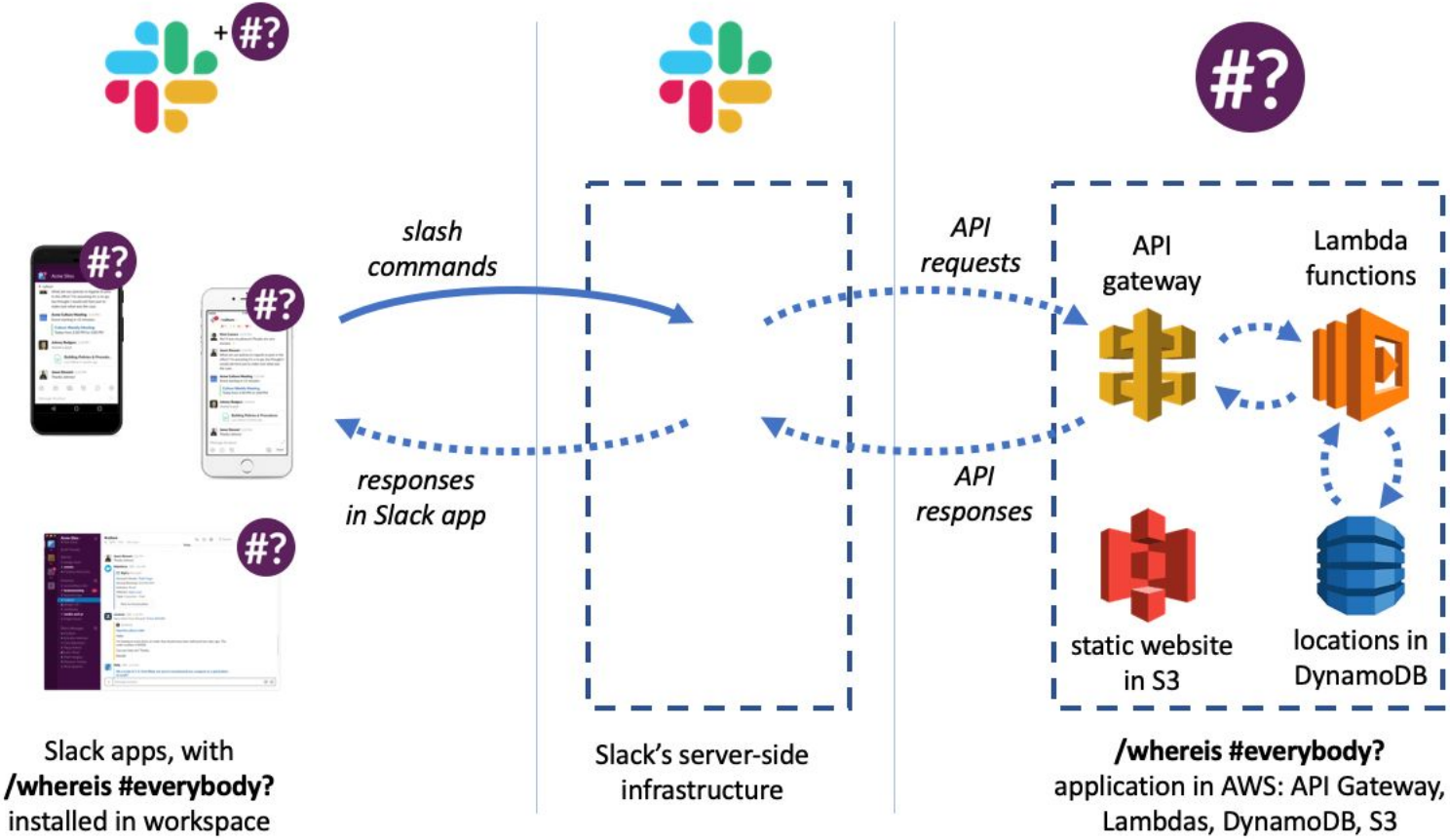
Add to Slack

After installation, the app works via a collection of 'slash commands'.

## The basics: registering your location

For example, you can inform your team where you're working:


slash commands
responses in Slack app
API requests
API responses
API gateway
Lambda functions
static website in S3
locations in DynamoDB
Slack apps, with /whereis #everybody? installed in workspace
Slack's server-side infrastructure
/whereis #everybody? application in AWS: API Gateway, Lambdas, DynamoDB, S3

# Which QA method is best for testing refactored functions in production?

# Requirements for QA of refactored software

Test a refactored implementation of something that's already in production

We can't (or don't want to) specify all test cases for unit/integration tests

It's a hassle to direct (historic) production traffic towards a new implementation

Don't activate a new implementation before we're really confident that it's better

Don't change software to enable testing


Tests in production
QA
Tests not in production

# Two groups of software QA methods

Division is made by *"with what do you compare the software?"*

- **compare software against specification or tester expectations**

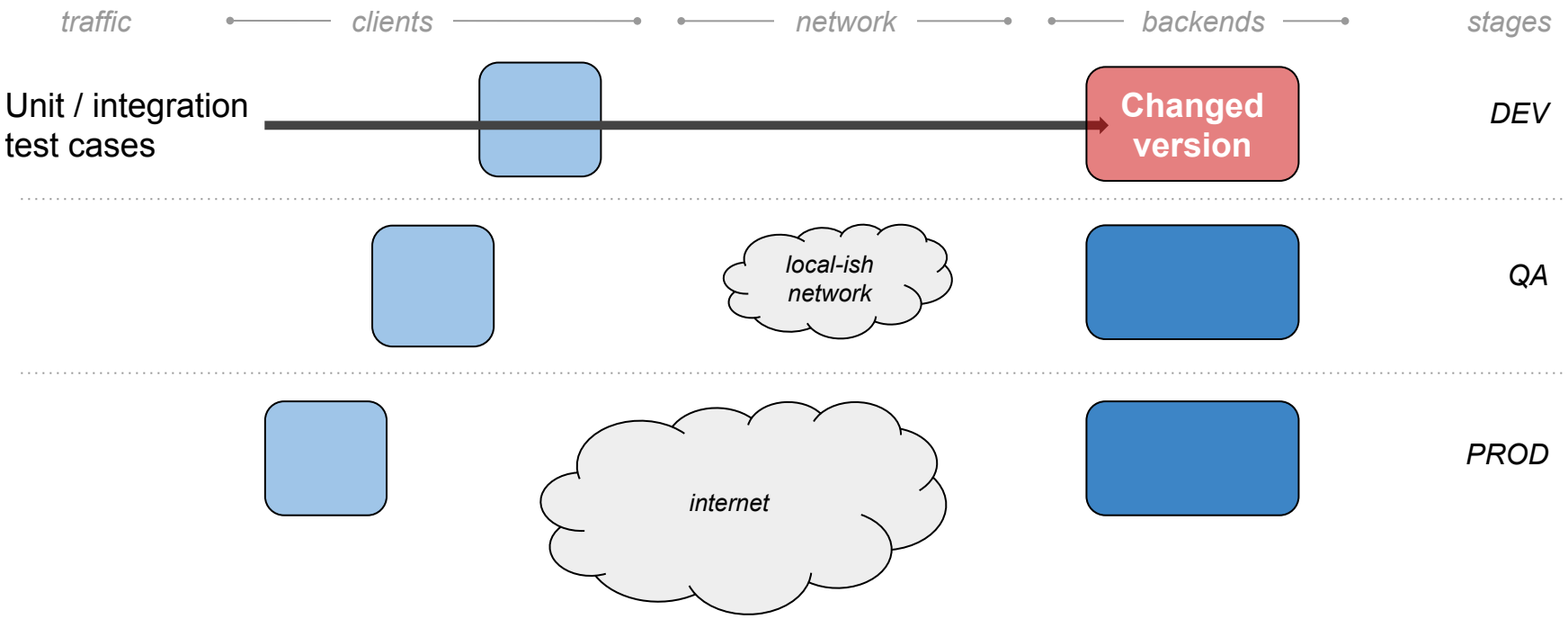
  Unit testing, Integration testing, Performance testing, Acceptance testing (typically, before new or changed software lands in production)

- **compare new version with earlier version**

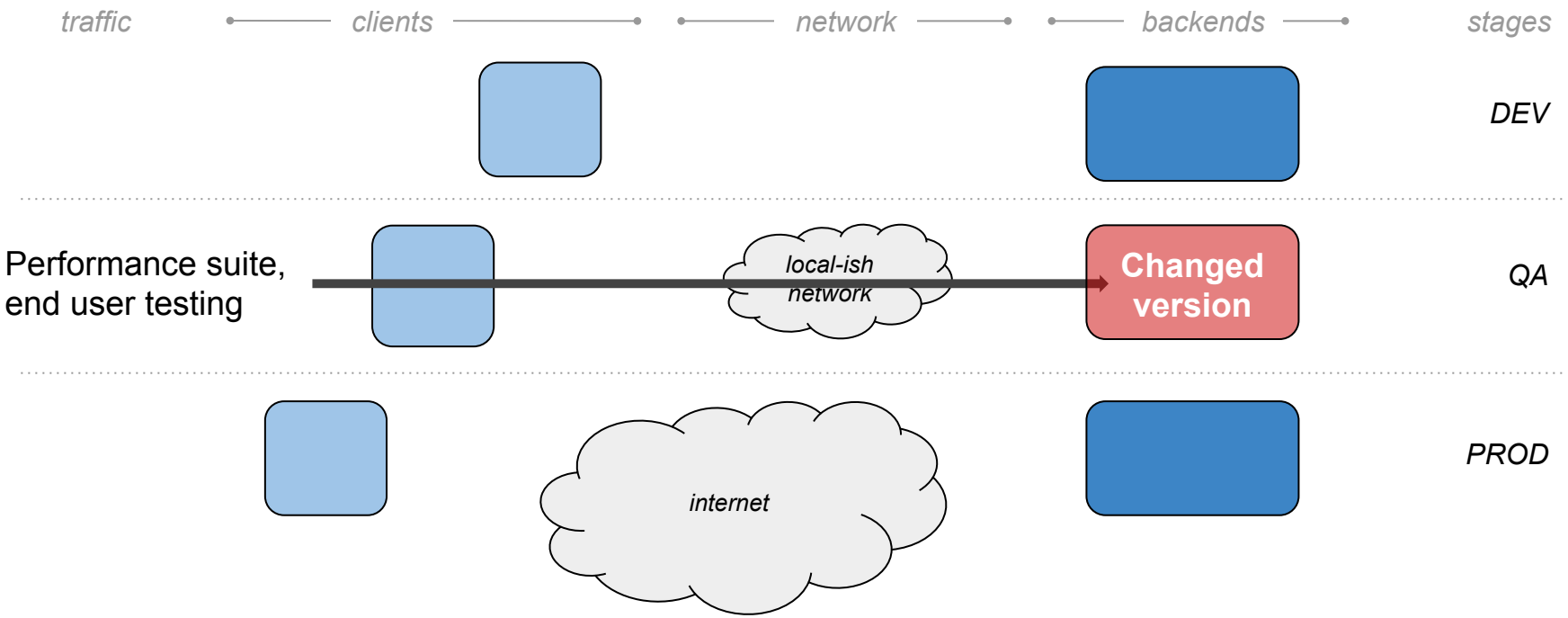
  Feature flags, blue/green deployments, Canary releases, A/B-testing

| *QA method* | **Test against** | **Phase** | **How to get test data** |
|---|---|---|---|
| *Unit testing* | Test spec | Dev | Manual / test suite |
| *Integration testing* | Test spec | Dev | Manual / test suite |
| *Performance testing* | Test spec | Tst | Dump production traffic /simulation |
| *Acceptance testing* | User spec | Acc | Manual |
| *Feature flags* | User expectations | Prd | Segment of production traffic |
| *A/B-testing* | Comparing options | Prd | Segment of production traffic |
| *Blue/green deployments* | User expectations | Prd | All production traffic |
| *Canary releases* | User expectations | Prd | Early segment of production traffic |

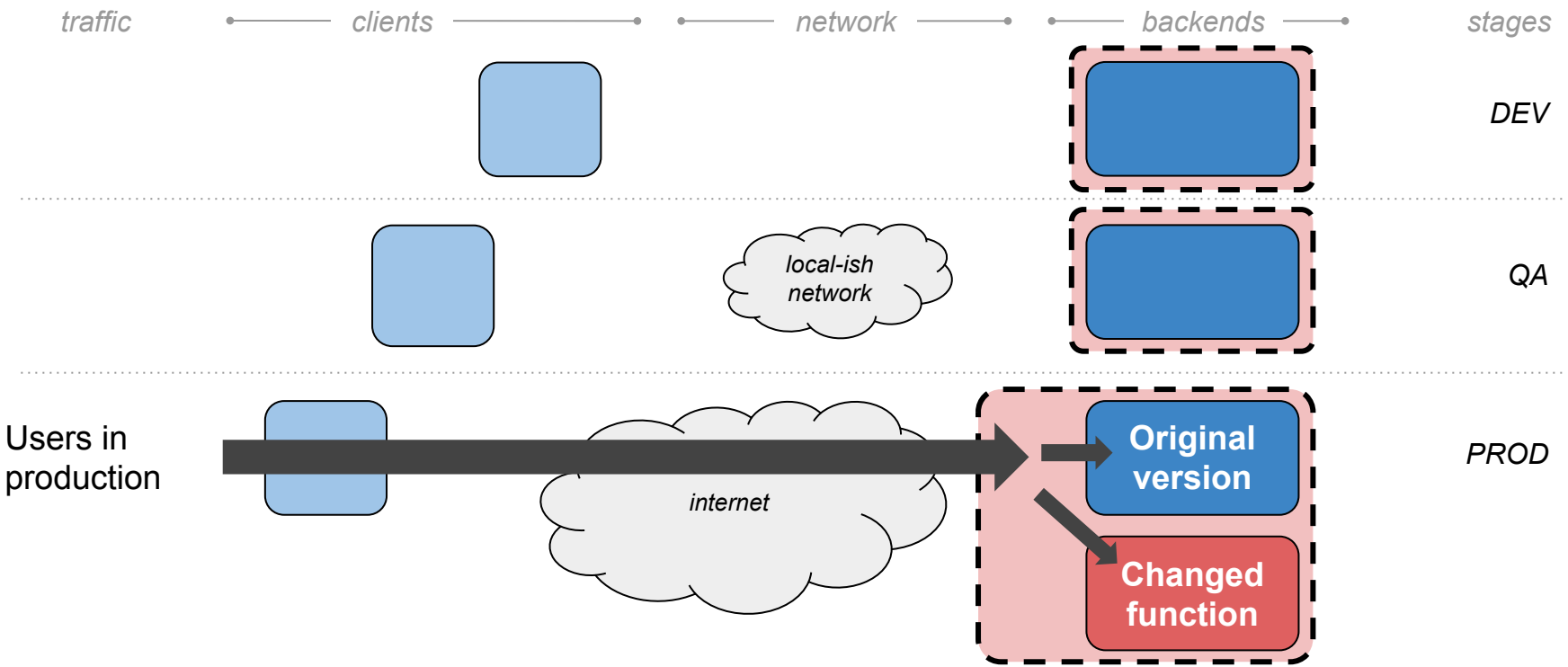
# QA method: unit / integration testing



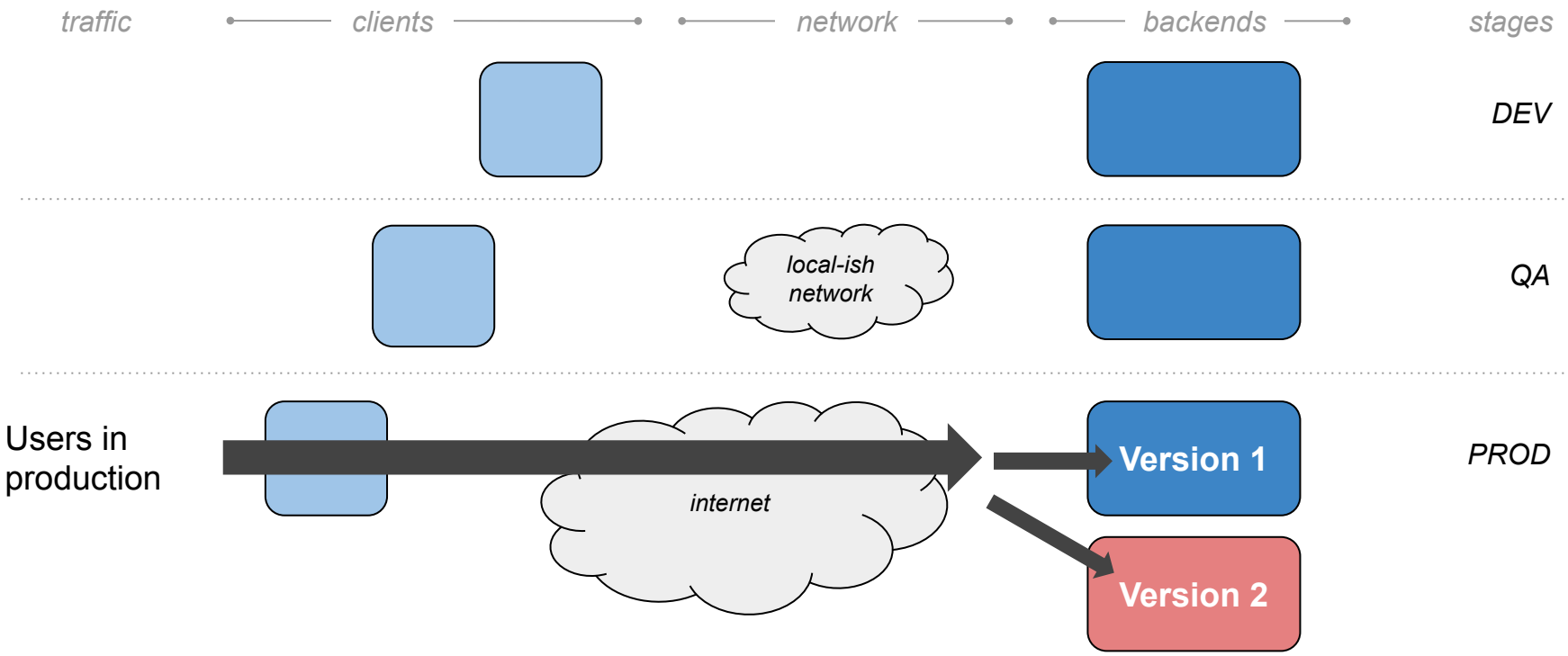
# QA method: performance / acceptance testing



# QA method: feature flags, A/B testing



# QA method: deployments, canary testing



Move Fast and Fix Things - The

https://github.blog/2015-12-15-move-fast/

December 15, 2015

Engineering

# Move Fast and Fix Things

Vicent Martí

Anyone who has worked on a large enough codebase knows that technical debt is an inescapable reality: The more rapidly an application grows in size and complexity, the more technical debt is accrued. With GitHub's growth over the last 7 years, we have found plenty of nooks and crannies in our codebase that are inevitably below our very best engineering standards. But we've also found effective and efficient ways of paying down that technical debt, even in the most active parts of our systems.

At GitHub we try not to brag about the "shortcuts" we've taken over the years to scale our web application to more than 12 million users. In fact, we do quite the opposite: we make a conscious effort to study our codebase looking for systems that can be rewritten to be cleaner, simpler and more efficient, and we develop tools and workflows that allow us to perform these rewrites efficiently and reliably.

As an example, two weeks ago we replaced one of the most critical code paths in our infrastructure: the code that performs merges when you press the Merge Button in a Pull Request. Although we routinely perform these kind of refactorings throughout our web app, the importance of the merge code makes it an interesting story to demonstrate our workflow.

## Merges in Git

We've talked at length in the past about the storage model that GitHub uses for repositories platform and our Enterprise offerings. There are many implementation details that make th efficient in both performance and disk usage, but the most relevant one here is the fact t repositories are always stored "bare".

in the repository (the ones that you would see on your actually available on disk in our infra

Scientist: Measure Twice, Cut

https://github.blog/2016-02-03-scientist/

February 3, 2016

Engineering

# Scientist: Measure Twice, Cut Over Once

Jesse Toth

Today we're releasing Scientist 1.0 to help you rewrite critical code with confidence.

As codebases mature and requirements change, it is inevitable that you will need to replace or rewrite a part of your system. At GitHub, we've been lucky to have many systems that have scaled far beyond their original design, but eventually there comes a point when performance or extensibility break down and we have to rewrite or replace a large component of our application.

## Problem

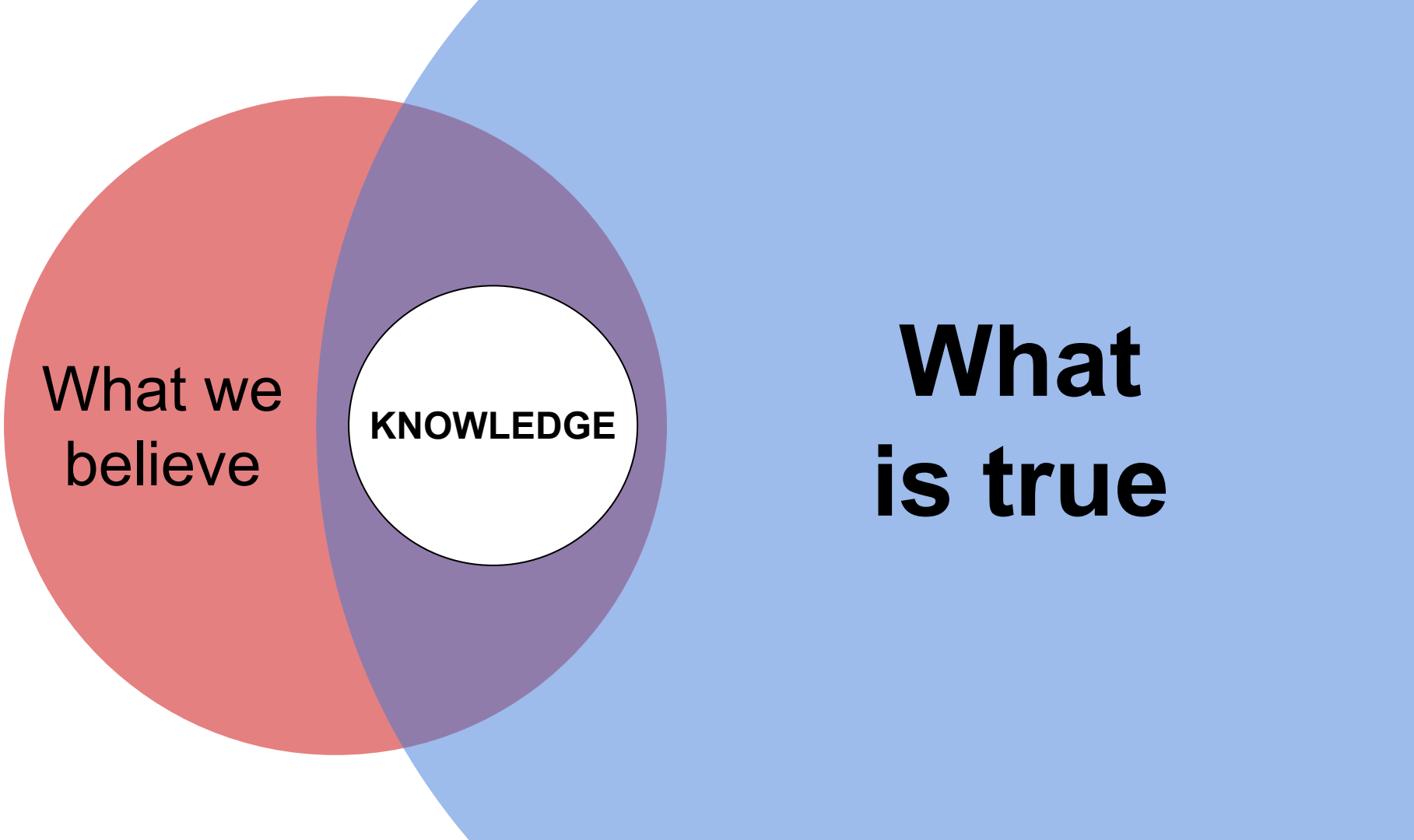
A few years ago when we were faced with the task of rewriting one of the most critical systems in our application — the permissions code that controls access and membership to repositories, teams, and organizations — we began looking for a way to make such a large change and have confidence in its correctness.

There is a fairly common architectural pattern for making large-scale changes known as Branch by Abstraction. It works by inserting an abstraction layer around the code you plan to change. The abstraction simply delegates to the existing code to begin with. Once you have the new code in place, you can flip a switch in the abstraction to begin substituting the new code for the old.

Using abstractions in this way is a great way to create a chokepoi path, making it easy to switch over to the new code when the ti ensure that the behavior of the new system will match the old s be called in all places where the old system was called. For suc architecture, this pattern only fulfilled half of the requirements. W the new system would be used in all places that the old system was, but also that it be correct and match what the old system did.

## Why tests aren't en

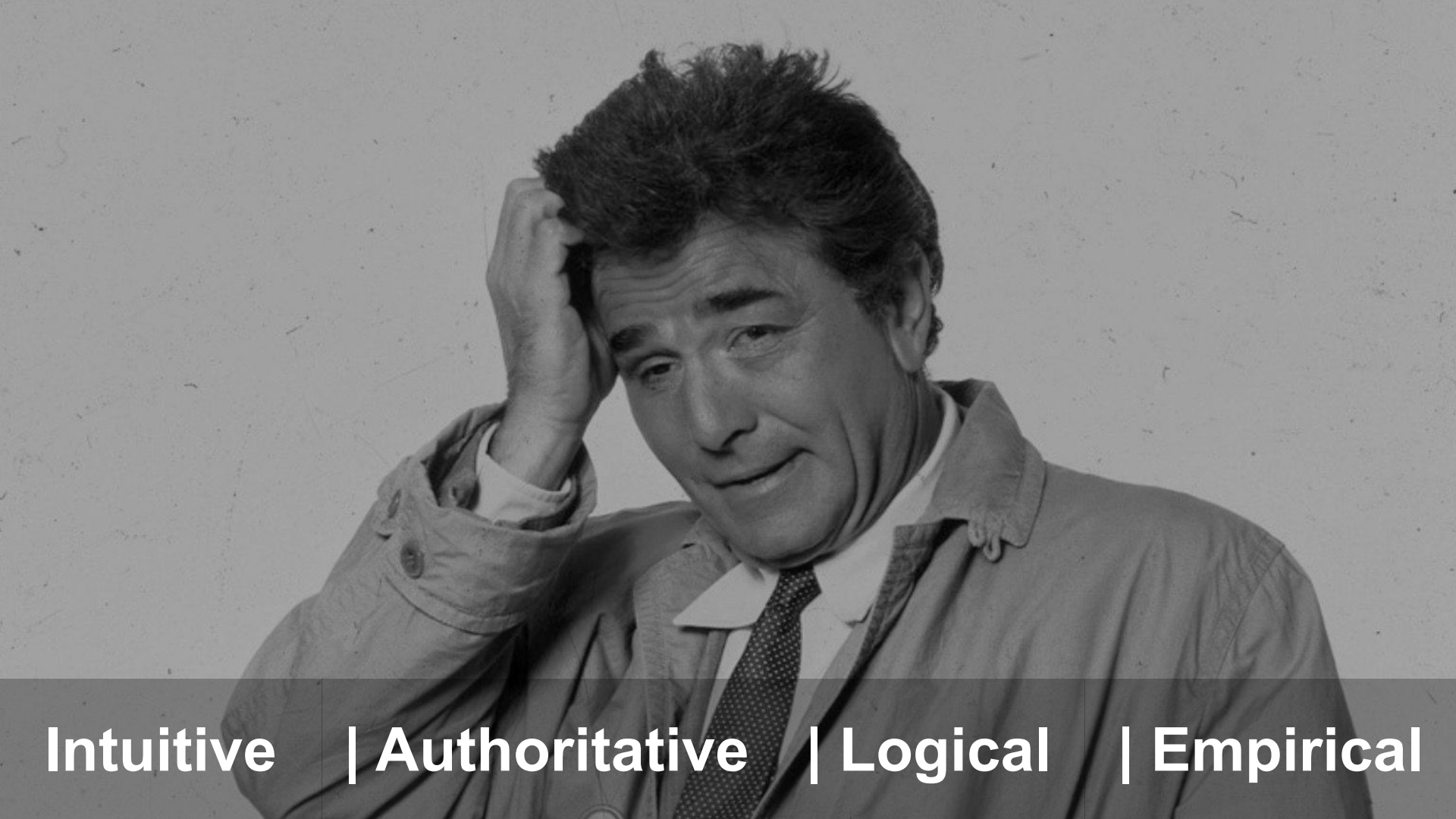
GitHub


What we believe
KNOWLEDGE
What is true

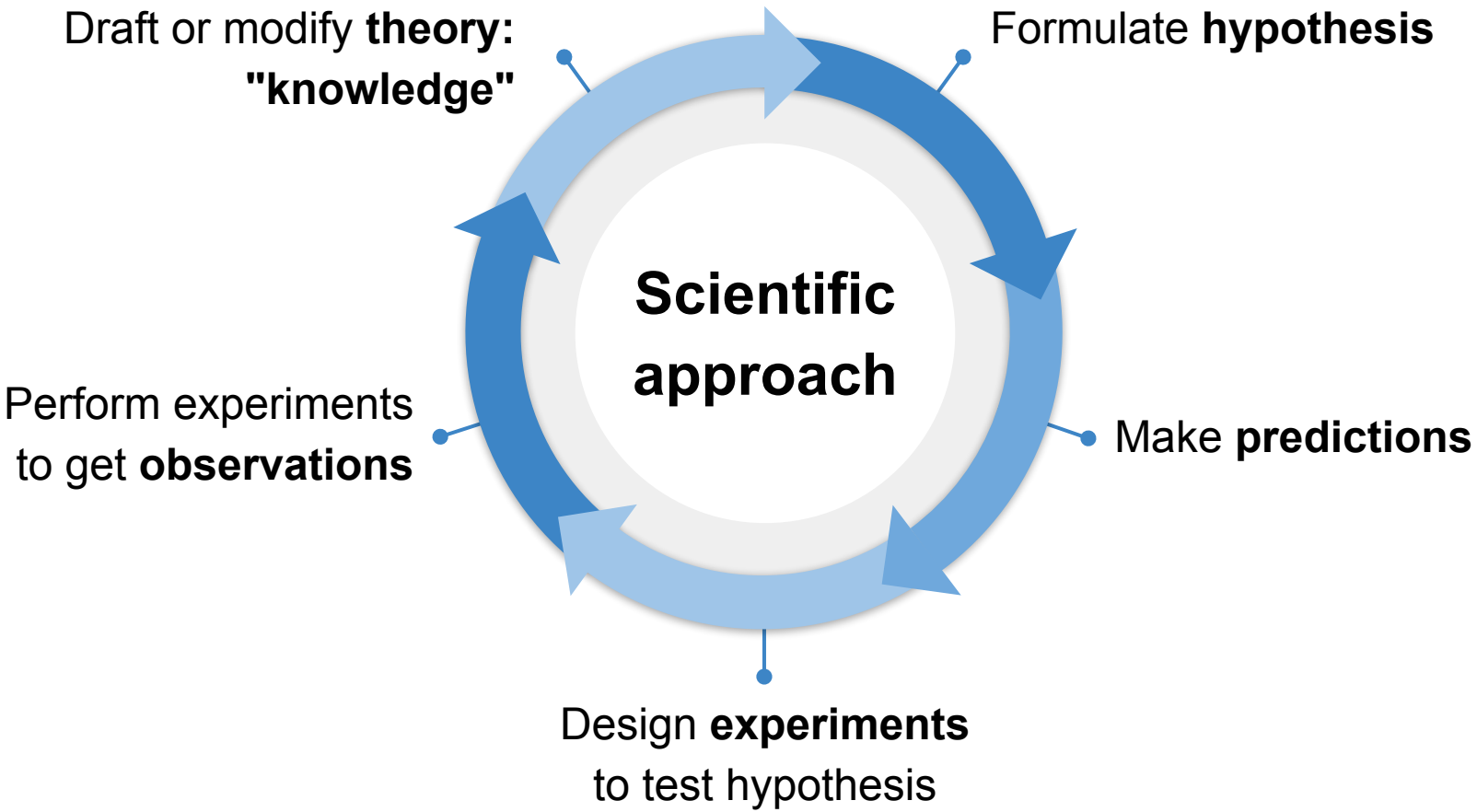
# Epistemology: knowledge, truth, and belief

Different 'sources' or types of knowledge:

- **Intuïtive knowledge**
  based on beliefs, feelings and thoughts, rather than facts
- **Authoritative knowledge**
  based on information from people, books, or any higher being
- **Logical knowledge**
  arrived at by reasoning from a generally accepted point
- **Empirical knowledge**
  based on demonstrable, objective facts, determined through observation and/or experimentation


Intuitive | Authoritative | Logical | Empirical


Intuitive | Authoritative | Logical | Empirical


Scientific approach
Formulate **hypothesis**
Make **predictions**
Design **experiments** to test hypothesis
Perform experiments to get **observations**
Draft or modify **theory: "knowledge"**

# Proposal: new software QA method, "Scientist"

Situation:

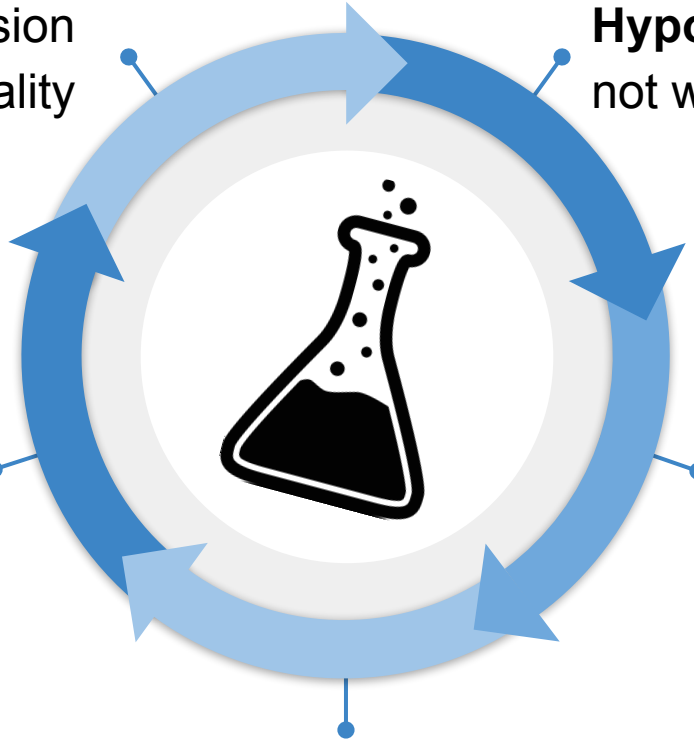
- We have an existing software component running in production: "control"
- We have an alternative (and hopefully better) implementation: "candidate"

Questions to be answered by an experiment:

- Is the candidate **behaving correctly** (or just as control) in all cases? (functionality)
- Is the candidate **performing qualitatively better** than the control? (response time, stability, memory use, resource usage stability, ...)

**Theory:** draw conclusion about software quality

**Hypothesis:** "candidate is not worse than control"



**Experiment:** process PROD traffic for sufficient amount of time

**Prediction:** "candidates performs better than control in production"

**Design experiment:** direct production traffic to candidates as well, compare results with control

# Requirements for such a Scientist in software

Ability to

- Experiment: test controls and (multiple) candidates with production traffic
- Observe: compare results of controls and candidates

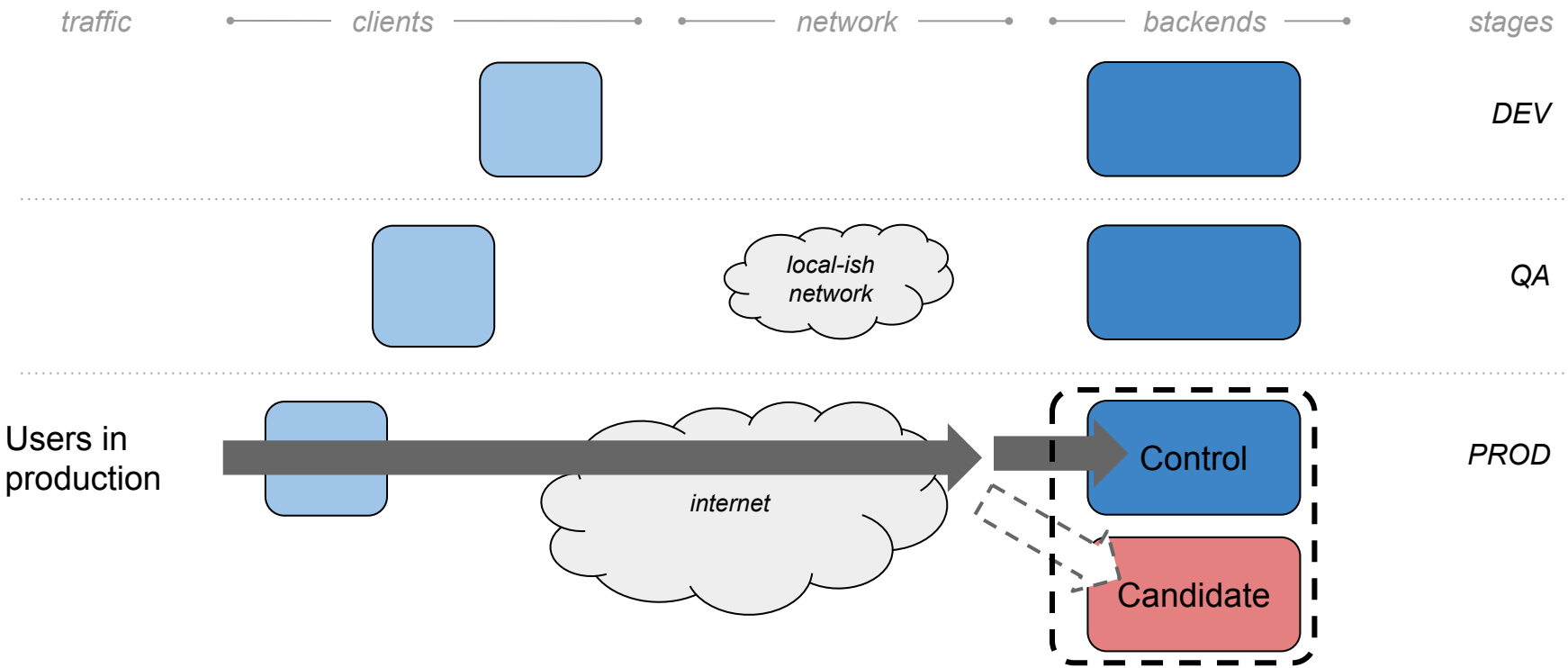
Additionally, for practical reasons in performing experiments

- Easily route traffic to single or multiple candidates
- Increase sample size once more confident of candidates
- No impact for end-consumer
- No change required in control – *where some miss the mark, IMHO*
- No persistent effect from candidates in production

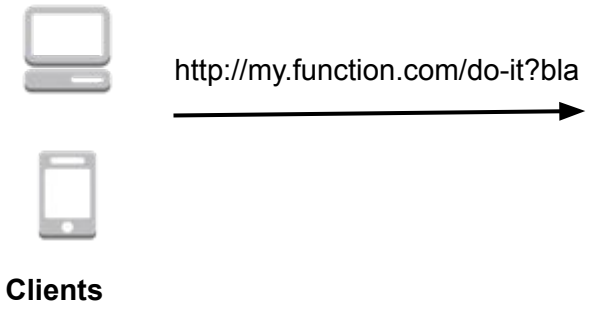
# Extra requirements for a *serverless* Scientist

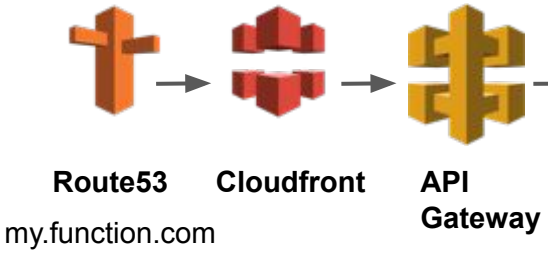
- Don't introduce complex 'plumbing' to get traffic to control and experiment
- Don't change software code of control in order to conduct experiments
- Don't add (too much) latency by introducing candidates in path
- Make it easy to define and enable experiments: routing traffic to candidates
- Make it effortless to deploy and activate candidates
- Store results and run-time data for both control and candidates
- Make it easy to compare control and candidates in experiments
- Make it easy to end experiments, leaving no trace in production

# QA method: Scientist



# Typical setup for serverless functions on AWS

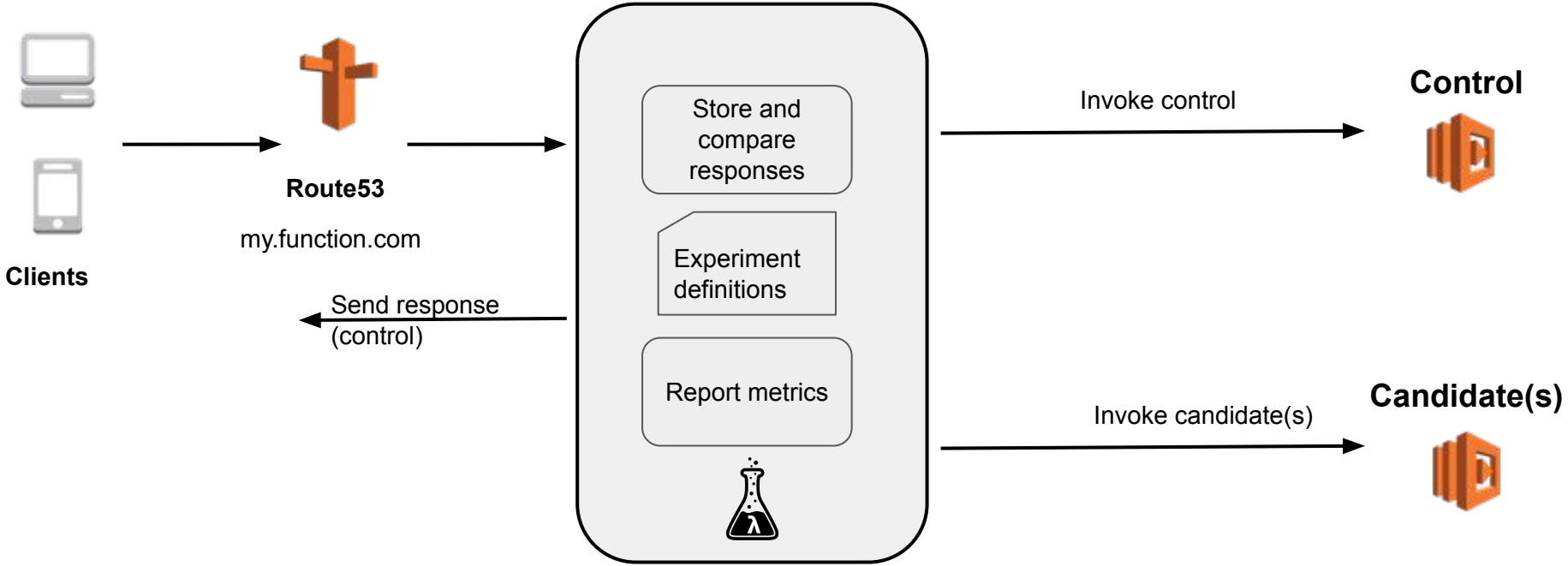


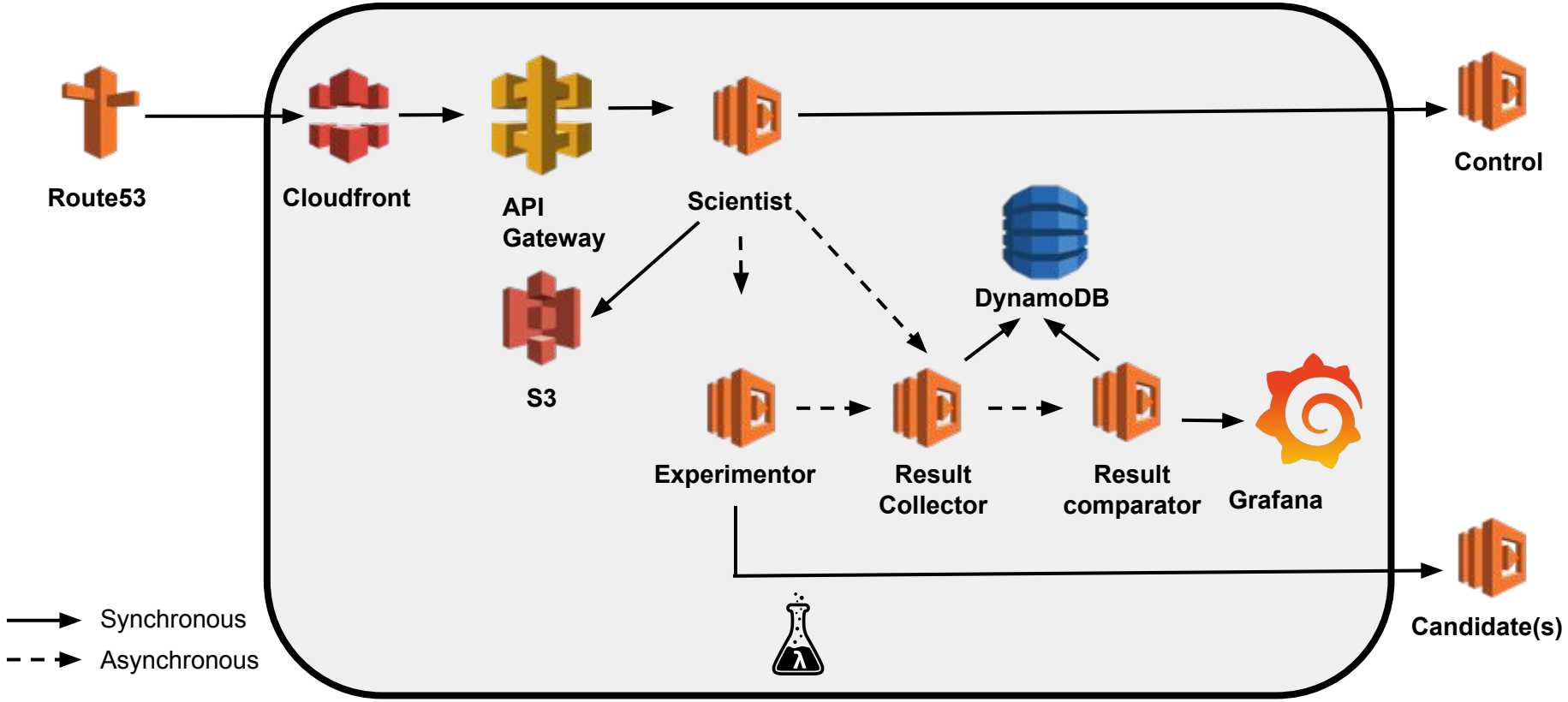



**Question: How do we compare the candidate against the control in production?**

# Serverless Scientist



# Serverless Scientist under the hood



# Example: rounding

```yaml
experiments:
  rounding-float:
    comparators:
      - body:
      - statuscode:
      - headers:
        - content-type
    path: round
    control:
      name: Round Node8.10
      arn: arn:aws:lambda:{AWSREGION}:{AWSACCOUNT_ID}:function:control-round
    candidates:
      candidate-1:
        name: Round Python3-math
        arn: arn:aws:lambda:{AWSREGION}:{AWSACCOUNT_ID}:function:candidate-round-python3-math
      candidate-2:
        name: Round python-3-round
        arn: arn:aws:lambda:{AWSREGION}:{AWSACCOUNT_ID}:function:candidate-round-python3-round
```

https://api.serverlessscientist.com/**round**?number=62.5

# Example of Serverless Scientist at work

Round: Simply round a number

Control request:

```
curl https://rounding-service.com/round?number=10.23
{"number":10.23,"rounded_number":10}
```

Serverless Scientist request:

```
curl https://api.serverlessscientist.com/round?number=10.23
{"number":10.23,"rounded_number":10}
```


## Control

```
1. {
2.     "body": "{\"number\":332.5,\"rounded_number\":333}",
3.     "headers": {
4.         "Content-Type": "application/json"
5.     },
6.     "statusCode": "200"
7. }
8.
```

## Round python-3-round

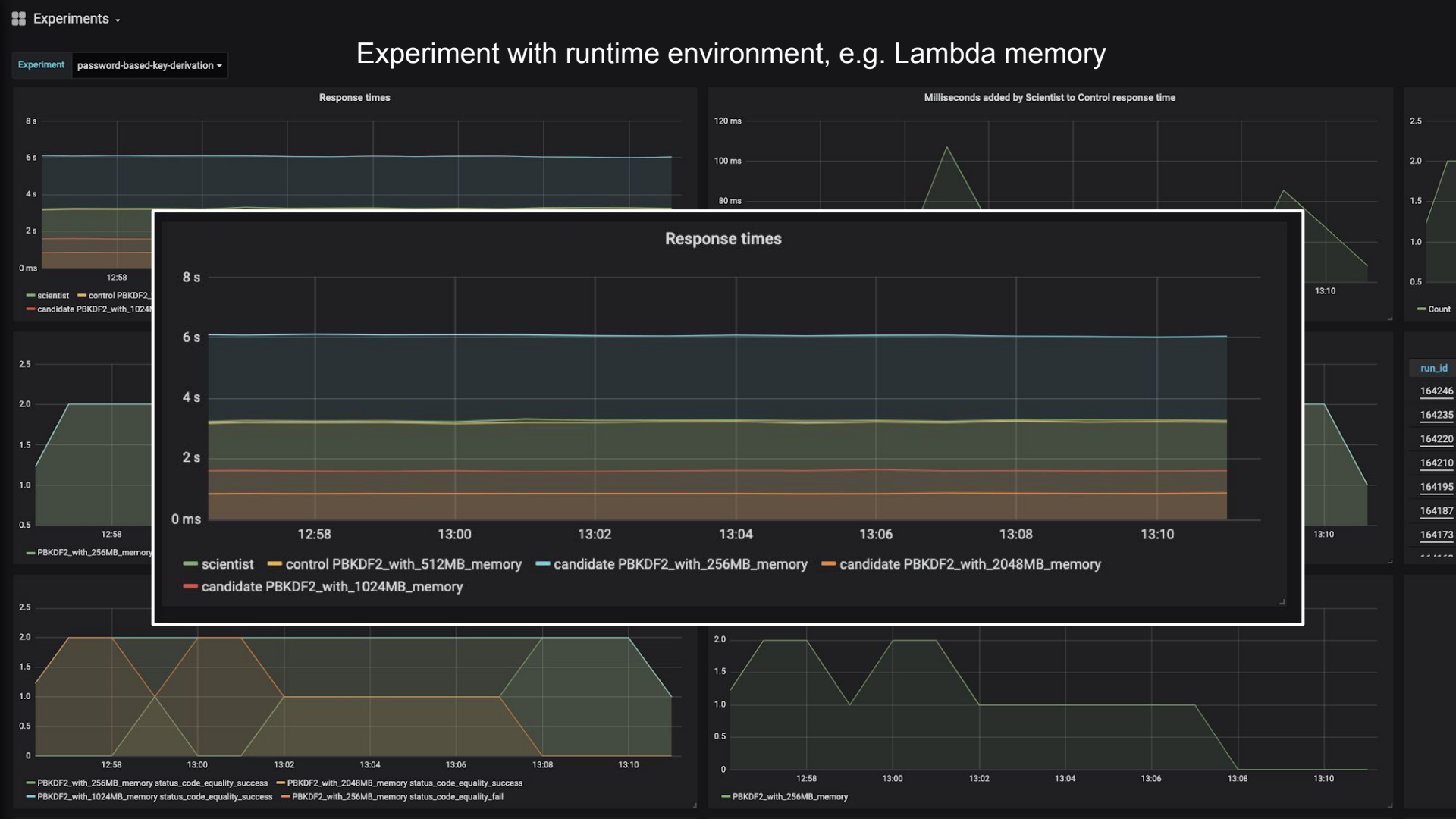
```
1. {
2.     "body": "{\"number\": 332.5, \"rounded_number\": 332}",
3.     "headers": {
4.         "Content-Type": "application/json"
5.     },
6.     "statusCode": "200"
7. }
8.
```

# Learnings: Compare on intended result (semantics) not on literal response

**https://qrcode?text=https://www.serverlessscientist.com**

Control | Candidate 1 | Candidate 2

# Experiment with runtime environment, e.g. Lambda memory



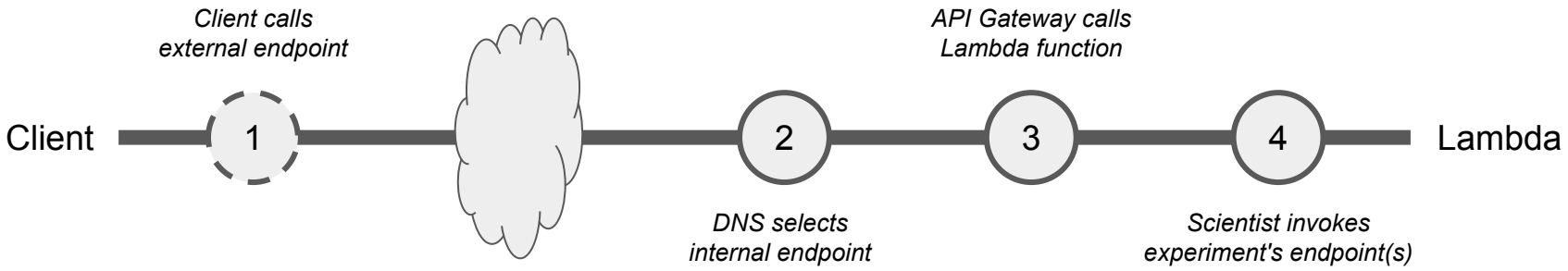
# Learnings from Serverless Scientist

- Detected unexpected differences between programming language (versions)
  - `Round()` in Python 2.7 `round(20.5)` returns 21.
  - `Round()` in Python 3: `round(20.5)` returns 20, not 21.
  - `Round()` in JavaScript: `round(20.5)` returns 21
- Compare on intended result (semantics) not on literal response (syntactically):
  - `{"first": 1, "second": 2}` versus `{"second": 2, "first": 1}`
  - Identical looking PNGs, but different binaries
- Easy to experiment and quick learning
  - adding/removing/updating candidates on the fly without impacting client
  - Instant feedback via the dashboard

# The route of client's request to Lambda function

Four major configuration points that determines which Lambda function is called:

1. (Client's request to an API endpoint - client decides which endpoint is called)
2. Proxy or DNS server - routing an external endpoint to an internal endpoint
3. API Gateway configuration - mapping a request to a Lambda function
4. Serverless Scientist - invoking functions for experiment's endpoints



# Options to promote candidate as new control

**2. Change the route for an external endpoint to another internal endpoint**

On load balancer, proxy function or DNS configuration,
direct traffic from old control to new candidate -> becomes new control

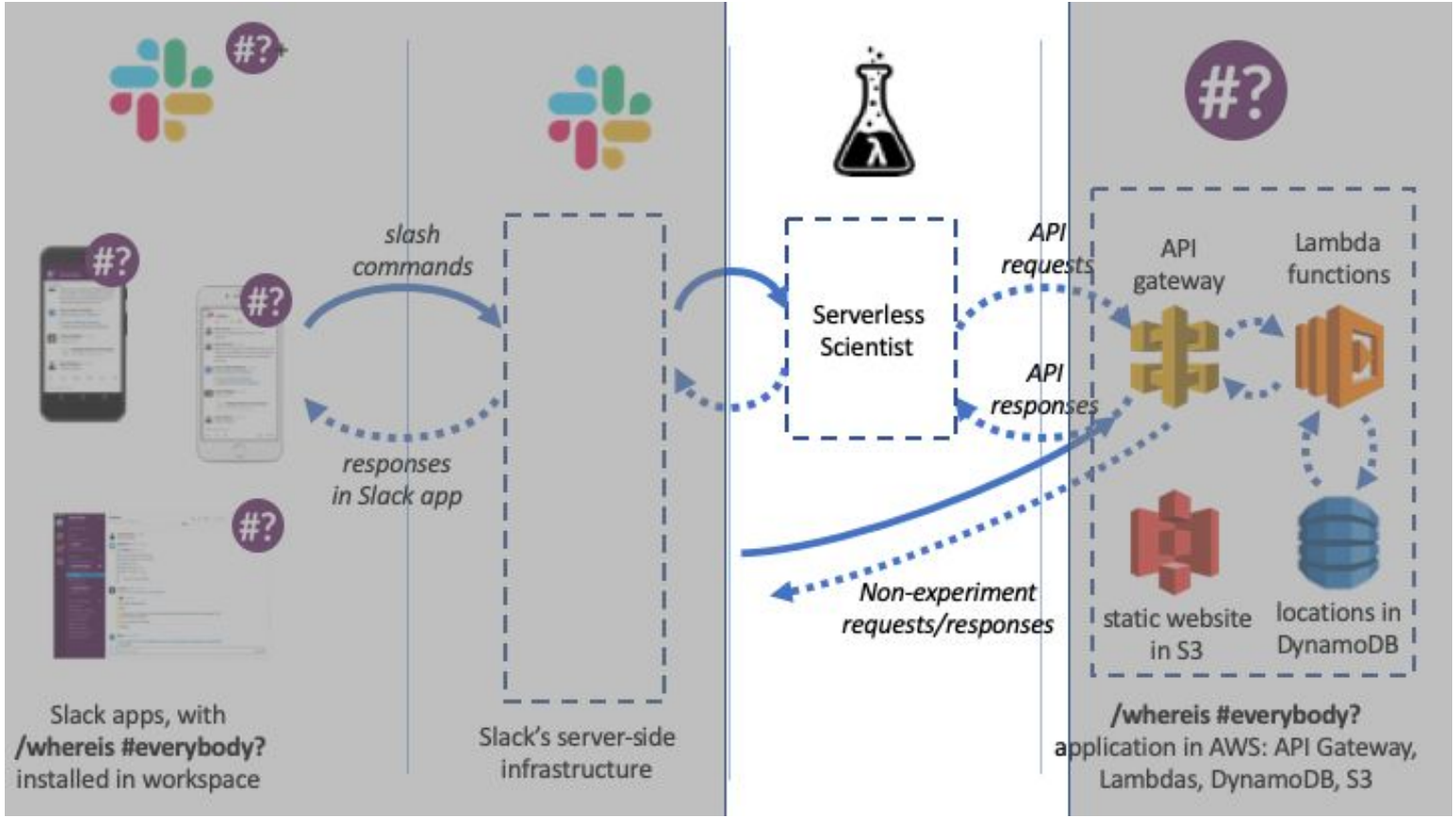
**3. Change API Gateway configuration: associate other Lambda function**

Change the existing production Lambda function to a new implementation: a Lambda function previously a candidate in an experiment

**4. Change setup of experiment: inject candidate as new control**

Change ARNs of control to the previous candidate in the experiment
(and possibly specify the old control as a new candidate)

# Serverless Scientist for */whereis #everybody?*

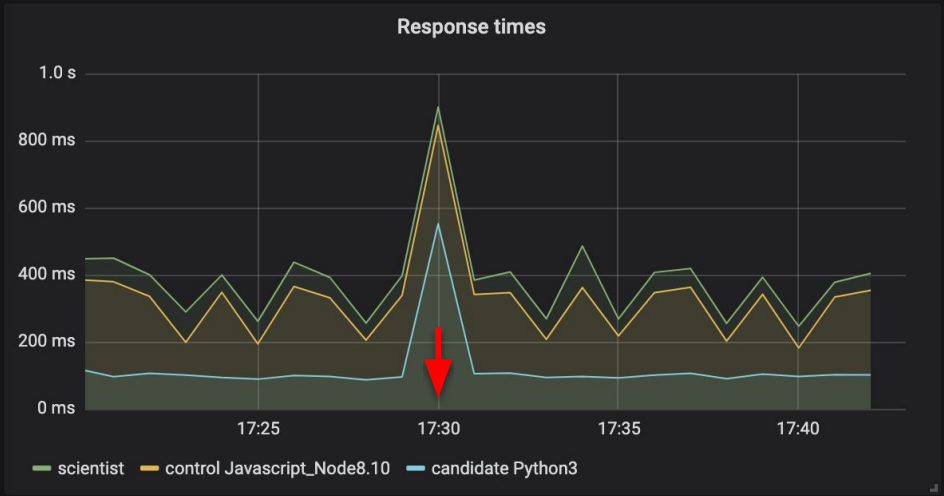


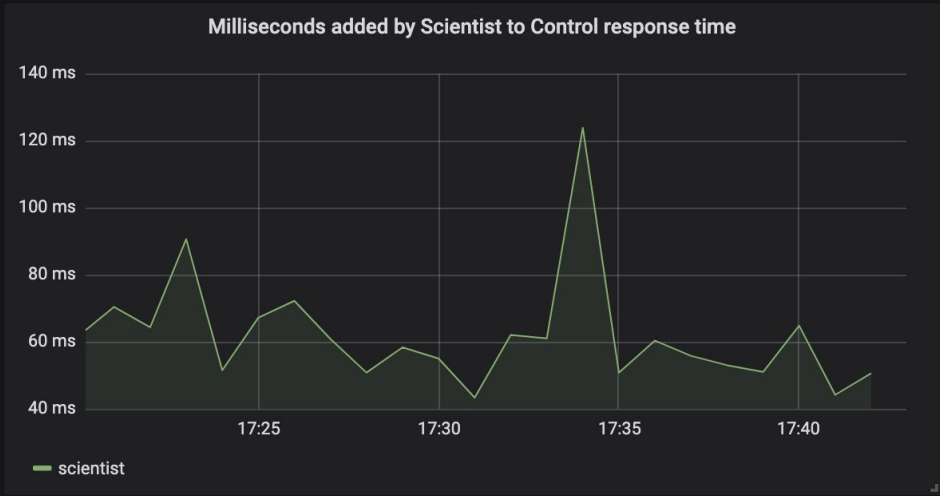
# Set up the experiment

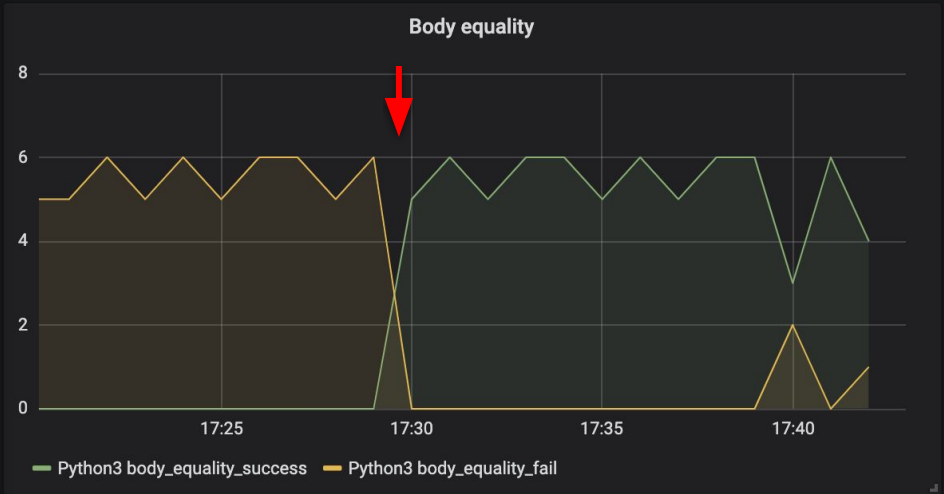
```yaml
experiments:
 wewhowasat:
   comparators:
     - body:
     - statuscode:
     - headers:
       - content-type
   path: whowasat
   control:
     name: Javascript Node8.10
     arn: arn:aws:lambda:{AWSREGION}:{AWSACCOUNT_ID}:function:whereis-everybody-prod-slackwhowasat
   candidates:
     candidate-1:
       name: Python3
       arn: arn:aws:lambda:{AWSREGION}:{AWSACCOUNT_ID}:function:whereis-everybody-prod-p_whowasat
```

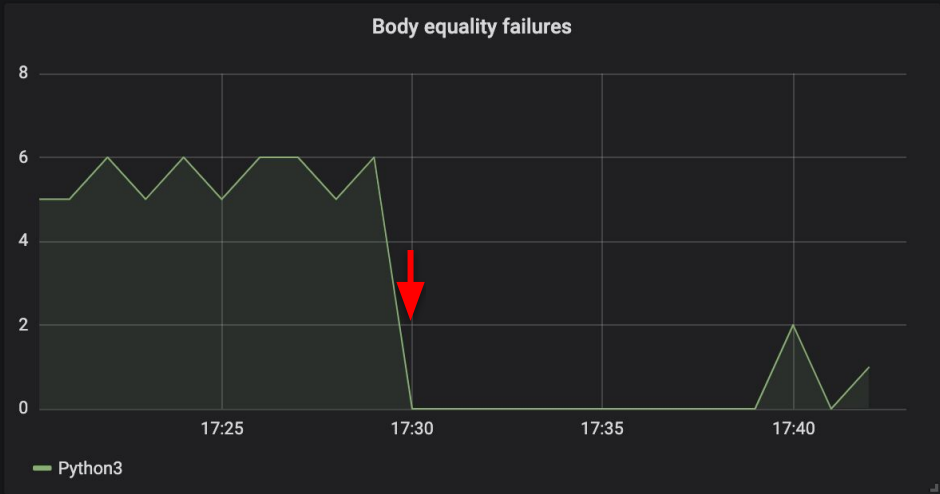
Experiment wewhowasatprod

New version deployed



# Advantages of (serverless) Scientist approach

Drop-in QA without changing code

No need to generate test traffic

No separate test suite

Iteratively improve candidates

Slowly increase traffic to candidates

Quick feedback with very limited risks

# Drawbacks of (serverless) Scientist approach

Degraded control response time

Additional latency

"Equal" == "equal" == "EQUAL"?

More function calls ➔ $

Syncing persistent changes
by control with candidates

Handling persistent changes in candidates

# When is a (serverless) Scientist less applicable?

When interface of service changes

- Requests to control cannot simply be duplicated to candidates
- Candidate responses not always comparable with control responses

When no production traffic is available, or is too limited

- Scientist shines with real-time, live production traffic
- Production traffic needs to have high code coverage, not neglecting parts

When a control is not (yet) available

- You need a control to compare a candidate against

# What's cooking in our lab?

- ~~Open-sourcing the code~~ https://gitlab.com/practicalarchitecture/serverless-scientist
- More fine-grained compare functions
- Distribute traffic over candidates
- Better management of experiments
- Support generic API testing
- Metrics reporting endpoints
- Better experiments dashboard
- Better UI for comparing results
- Support for other FaaS platforms


Xebia