How a scientist would improve serverless functions

Gero Vermaas, Jochem Schulenklopper

O'Reilly Software Architecture
Berlin, Germany, November 7th, 2019
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?
A friendly Slack app that remembers where you said you were

/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](https://whereis-everybody.com/add.png)

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:

```
/iamat location
```

Store current location(s)

```
+ /iamat
```

"bold", "italic", "strike", "code", "preformatted", "quote"
Slack apps, with `/whereis #everybody?` installed in workspace

Slack’s server-side infrastructure

API gateway

Lambda functions

static website in S3

locations in DynamoDB

`/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 not in production

Tests in production

QA
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
<table>
<thead>
<tr>
<th>QA method</th>
<th>Test against</th>
<th>Phase</th>
<th>How to get test data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit testing</strong></td>
<td>Test spec</td>
<td>Dev</td>
<td>Manual / test suite</td>
</tr>
<tr>
<td><strong>Integration testing</strong></td>
<td>Test spec</td>
<td>Dev</td>
<td>Manual / test suite</td>
</tr>
<tr>
<td><strong>Performance testing</strong></td>
<td>Test spec</td>
<td>Tst</td>
<td>Dump production traffic / simulation</td>
</tr>
<tr>
<td><strong>Acceptance testing</strong></td>
<td>User spec</td>
<td>Acc</td>
<td>Manual</td>
</tr>
<tr>
<td><strong>Feature flags</strong></td>
<td>User expectations</td>
<td>Prd</td>
<td>Segment of production traffic</td>
</tr>
<tr>
<td><strong>A/B-testing</strong></td>
<td>Comparing options</td>
<td>Prd</td>
<td>Segment of production traffic</td>
</tr>
<tr>
<td><strong>Blue/green deployments</strong></td>
<td>User expectations</td>
<td>Prd</td>
<td>All production traffic</td>
</tr>
<tr>
<td><strong>Canary releases</strong></td>
<td>User expectations</td>
<td>Prd</td>
<td>Early segment of production traffic</td>
</tr>
</tbody>
</table>
QA method: unit / integration testing

traffic — clients — network — backends — stages

Unit / integration test cases — Changed version

local-ish network — internet

DEV

QA

PROD
QA method: performance / acceptance testing

traffic ← clients → network ← backends → stages

DEV

Performance suite, end user testing

local-ish network

QA

Changed version

QA

PROD

internet
QA method: feature flags, A/B testing
QA method: deployments, canary testing

Users in production | Version 1 | Version 2

internet

local-ish network

traffic — clients — network — backends

stages

DEV

QA

PROD

Version 1

Version 2
Move Fast and Fix Things

Vineet Marri

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 it incurs. With GitHub’s growth over the last 7 years, we have found plenty of corners and cracks in our codebase that are inevitably below our very best engineering standards. But we’ve also found our codebase at GitHub to be 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 workarounds that allow us to perform these rewritings more efficiently and reliably.

As an example, two weeks ago we replaced one of the most critical code paths in our infrastructure. Although the code that performs merges when you press the Merge Button in a Pull Request, the importance of the merge routine performance is 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. This model has two parts: platform and our Enterprise offerings. These are many implementation details that make up the platform and our Enterprise offerings, but most important are the facts that the platforms are always stored as “bare”, with the repository (the one you would see on your local machine) being available on disk in our infrastructure.

Today, we’re releasing Scientist 1.0, a tool to help you rewrite critical code with confidence.

As codebases mature and requirements change, it is inevitable that you will need to rewrite or refactor a part of your system. At GitHub, we’ve found that many systems have scaled far beyond their original design, but eventually there comes a point when performance or extendibility breaks 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 in 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 using the new code for the old.

Why tests aren’t enough

Using abstractions in this way is a great way to create a changepath, making it easy to switch over to the new code when the old one ceases to be needed. It also can be called in places where the old system was called. For example, the new system would be used in all places that the old system was, but also those parts of a system that are not only that are not used for the old one. When the old system is removed, the new one can be used in its place.

Why tests aren’t enough

But tests aren’t enough. They only provide confidence in correctness when the system is in a stable state. For example, if you have a function that returns a result, and the result is always the same, then the tests are enough. But if the function changes at some point, the tests won’t catch it.

Instead, we need a way to make a change in a way that ensures that the behavior of the new system is the same as the old system. This is where Scientist comes in. It allows you to rewrite a part of the system in a way that makes it easier to switch over to the new code when the old one ceases to be needed. It also can be called in places where the old system was called. For example, the new system would be used in all places that the old system was, but also those parts of a system that are not only that are not used for the old one. When the old system is removed, the new one can be used in its place.

Scientist: Measure Twice, Cut Over

February 3, 2016

Engineering

Jason Toth
What we believe

KNOWLEDGE

What is true
Epistemology: knowledge, truth, and belief

Different 'sources' or types of knowledge:

- **Intuitive 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
Draft or modify theory: "knowledge"

Formulate hypothesis

Make predictions

Perform experiments to get observations

Design experiments to test hypothesis
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"

Prediction: "candidates performs better than control in production"

Experiment: process PROD traffic for sufficient amount of time

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

traffic -- clients -- network -- backends -- stages

DEV

QA

Users in production

PROD

internet

local-ish network

Control

Candidate
Typical setup for serverless functions on AWS

Clients

http://my.function.com/do-it?bla

Route53
Cloudfront
my.function.com
API Gateway

do-it Lambda

Control

Candidate

do-it better Lambda

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

- **Clients**
  - my.function.com

- **Route53**
  - my.function.com

- **Store and compare responses**
- **Experiment definitions**
- **Report metrics**

- **Invoke control**
- **Invoke candidate(s)**

- **Control**
- **Candidate(s)**
Serverless Scientist under the hood
Example: rounding

```
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

```
{
    "body": "{\"number\":332.5, \"rounded_number\":332}\",
    "headers": {
        "Content-Type": "application/json"
    },
    "statusCode": "200"
}
```

### Round python-3-round

```
{
    "body": "{\"number\": 332.5, \"rounded_number\": 332}\",
    "headers": {
        "Content-Type": "application/json"
    },
    "statusCode": "200"
}
```
Learnings: Compare on intended result (semantics) not on literal response

Experiment with runtime environment, e.g. Lambda memory
Learnings from Serverless Scientist

- Detected unexpected differences between programming language (versions)
  - `round(20.5)` returns 21 in Python 2.7
  - `round(20.5)` returns 20, not 21, in Python 3
  - `round(20.5)` returns 21 in JavaScript
- 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 determine 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 \( \rightarrow \) 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

```json
experiments:
  wehhowasat:
    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
```
New version deployed
Advantages of (serverless) Scientist approach

- Drop-in QA without changing code
- No need to generate test traffic
- No separate test suite
- Slowly increase traffic to candidates
- Iteratively improve candidates
- Quick feedback with very limited risks
Drawbacks of (serverless) Scientist approach

Additional latency

More function calls $\rightarrow$

Degraded control response time

"Equal" $\Rightarrow$ "equal" $\Rightarrow$ "EQUAL"?

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