QCon SF 2019

Observability in the Software Supply Chain

Seeing into your build system
Intro

Ben Hartshorne

honeycomb.io

@maplebed

ben@honeycomb.io
What’s this talk about

- Part story
- Part pulling back the curtain
- Part encouragement
Part 1 - Story Time
Part 1

The #1 Programmer Excuse for Legitimately Slacking Off:

"My code's compiling."

Hey! Get back to work!

Compiling!

Oh. Carry on.

https://imgs.xkcd.com/comics/compiling.png
Part 1: Story Time

- Our builds were slow
- Well, they *felt* slow
- Were they actually?
- We had no idea
Part 1: Story Time

... but we believe in Observability!!!
<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>poodle-yarn</td>
<td>34.233s</td>
</tr>
<tr>
<td>doodle-yarn</td>
<td>2.205s</td>
</tr>
<tr>
<td>honeycomb-migrations</td>
<td>4.050s</td>
</tr>
<tr>
<td>retriever-migrations</td>
<td>81.00ms</td>
</tr>
<tr>
<td>poodle-build</td>
<td>88.671s</td>
</tr>
<tr>
<td>go-test</td>
<td>111.294s</td>
</tr>
<tr>
<td>poodle-test</td>
<td>35.803s</td>
</tr>
<tr>
<td>doodle-build</td>
<td>435.0ms</td>
</tr>
<tr>
<td>doodle-test</td>
<td>1.589s</td>
</tr>
<tr>
<td>go-install</td>
<td>31.216s</td>
</tr>
</tbody>
</table>
go test
yarn
Part 1: Story Time

● Instrumentation
  ○ Changed our model
  ○ Guided work
Part 1: Story Time
Overall build times
Part 1: Story Time

critical path
Part 1: Story Time

● Instrumentation
  ○ Changed our model
  ○ Guided work
  ○ Proved results
  ○ Uncovered unknown variance
Part 1: Story Time

- Using production-focused tools
  - Developers unfamiliar with build use the same tools they know to understand a different service
Part 2 - How does it work?
Part 2: How did we do it?

- Go back to the beginning of the story
  - Between “observability” and “so we instrumented it”
  - We need to define the problem
Part 2: How did we do it?

- **Vocab: Build System**
  - Really Build and Test
  - On a trigger (usually a commit), do some stuff
  - Can Succeed or Fail
  - Runs many shell commands
Part 2: How did we do it?

- Core duties of a build system
  - Set up an isolated environment
  - Run a bunch of commands
  - Stop when one fails
  - Record the result
Part 2: How did we do it?

- Core instrumentation for builds
  - How long did it take
  - What commands were run
    - How long did each take
  - Did it succeed
    - Did each command succeed
Part 2: How did we do it?

- **How long did it take?**
  - 398.452s

- **What commands were run and how long did each take?**
  - poodle-yarn: 34.233s
  - honeycomb-migrations: 2.205s
  - retriever-migrations: 4.050s
  - poodle-build: 88.671s
  - go-test: 111.294s
  - poodle-test: 35.803s
  - doodle-build: 435.008s
  - doodle-test: 1.589s
  - go-install: 31.216s

- **Did they succeed?**
  - success
  - success
  - success
  - success
  - failure
Part 2: How did we do it?

- Sample build config

```go
language: go

addons:
  artifacts:
    paths:
      - $GOPATH/bin

script: go test github.com/honeycombio/honeyvent/...

after_success:
  - rm $GOPATH/bin/honeyvent
  - go install -ldflags "-X main.BuildID=1.$(TRAVIS_BUILD_NUMBER)" github.com/honeycombio/honeyvent/...```
Part 2: How did we do it?

- Let’s write a small wrapper that
  - Takes a name and a command
  - Measures how long it takes to run
  - Records its exit status
  - and passes it along
  - Outputs the resulting data
```bash
#!/bin/bash

### the first argument of our script is the name
### everything else is the command to run
name=$1
shift

### time the thing and record its exit code
start=$(date +%s)
eval $*
exitcode=$?
end=$(date +%s)

### send our instrumentation
dur=$((end - start))
echo name=$name dur=$dur exitcode=$exitcode

### exit with the same status code as the command we ran
exit $exitcode
```
Part 2: How did we do it?

```
$ ~/bin/buildeventbash.sh fancysleep sleep 4
    name=fancysleep dur=4 exitcode=0

$ echo $?  
    0
```

```
$ ~/bin/buildeventbash.sh fancyfail false
    name=fancyfail dur=0 exitcode=1

$ echo $?  
    1
```
#!/bin/bash

### the first argument of our script is the name
### everything else is the command to run
name=$1
shift

### time the thing and record its exit code
start=$(date +%s)
eval $*
exitcode=$?
end=$(date +%s)

### send our instrumentation
dur=$(((end - $start))
curl https://hny.co -d "name=$name dur=$dur exitcode=$exitcode"

### exit with the same status code as the command we ran
exit $exitcode
Part 2: How did we do it?

- Our new build config

```yaml
language: go

addons:
  artifacts:
    paths:
      - $GOPATH/bin

# our script buildeventbash.sh
# is checked in at tools/beb
script: tools/beb go-test go test github.com/honeycombio/honeyvent/...

after_success:
  - rm $GOPATH/bin/honeyvent
  - tools/beb go-install go install -ldflags "-X main.BuildID=1.$(TRAVIS_BUILD_NUMBER)" \
    github.com/honeycombio/honeyvent/...
```
Part 2: How did we do it?

- What gets this beyond prototype?
  - Switch languages bash -> go
  - Link commands together
  - Improve our data model
  - Collect additional context
Part 2: How did we do it?

- Link our commands together
  - Tie all the spans in to a trace
  - Use the Build ID from the build system
Part 2: How did we do it?

script:
- STEP_START=$(date +%s)
- STEP_SPAN_ID=$(echo script | sum | cut -f 1 -d \ )
- ./buildevents cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID go-tests -- go test ./...
- ./buildevents cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID js-tests -- yarn test
- ./buildevents step $TRAVIS_BUILD_ID $STEP_SPAN_ID $STEP_START script
Part 2: How did we do it?

● Improve our data model
  ○ Group commands into steps
    ■ Time the whole group
  ○ build is the whole thing
  ○ step is a group of commands
  ○ cmd is one specific command
Part 2: How did we do it?

```bash
script:
- STEP_START=$(date +%s)
- STEP_SPAN_ID=$(echo script | sum | cut -f 1 -d \ )
- ./buildevets cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID no-tests -- go test ./...
- ./buildevets cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID js tests -- yarn test
- ./buildevets step $TRAVIS_BUILD_ID $STEP_SPAN_ID $STEP_SPAN_ID $STEP_START script
```

*Group these two commands together*
Part 2: How did we do it?

```bash
script:
  STEP_START=$(date +%s)
  STEP_SPAN_ID=$(echo script | sum | cut -f 1 -d )
  ./buildevens cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID go-tests -- go test ./...
  ./buildevens cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID js-tests -- yarn test
  ./buildevens step $TRAVIS_BUILD_ID $STEP_SPAN_ID $STEP_START script
```
Part 2: How did we do it?

```bash
- STEP_START=$(date +%s)
- STEP_SPAN_ID=$(echo script | sum | cut -f 1 -d \ )
- ./buildevens cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID go-tests -- go test ./...
- ./buildevens cmd $TRAVIS_BUILD_ID $STEP_SPAN_ID js-tests -- yarn test
- ./buildevens step $TRAVIS_BUILD_ID $STEP_SPAN_ID $STEP_START script
```
Part 2: How did we do it?
Part 2: How did we do it?

- Collect additional context
  - Branch name, PR#, CI system, etc.
  - Custom fields from the developer
    - Artifact size
    - Build depth
    - Links to other data sources
    - Test results, etc.
Part 2: How did we do it?

- Collect additional custom context
  - Put `name=value` pairs in a file
  - Put the name of that file in the env
  - `BuildEvents` includes those fields
Part 2: How did we do it?
Part 2: How did we do it?

- One more trick
  - Interrogate the build system API
  - CircleCI has live API endpoints
  - Reveals the *real* start time
  - Other juicy data
Part 2: How did we do it?

- Speaking of open source ...

github / honeycombio / buildevents

Travis-CI  GitLab  JenkinsX  Google Cloud Build  CircleCI
Part 2: How did we do it?

- Build systems run shell commands
- By wrapping shell commands we hook in instrumentation
- Use the env or fs for IPC
- Pull extra context from the env
- Pull extra context from APIs
- Send that data to visualization
Part 3 - What’s next?
Part 3: What’s next?

- Build systems are part of the SSC
  - and traces easily represent builds
- What other parts would benefit from new visualizations and tracking?
  - Commit to deploy lifecycle
  - ???
Part 3: What’s next?

- Commit to deploy lifecycle
  - Many commits to one PR
  - Many review cycles before merge
  - Many builds along the way
  - Many environments in deploy
  - Many PRs in one deploy?
  - Revisions?
Part 3: What’s next?

- Commit to deploy lifecycle challenge
  - Represent individual run well
    - Show commit-forward
    - Show deploy-backward
    - Represent cycles, delays
Part 3: What’s next?

- Commit to deploy lifecycle challenge
  - Represent runs in aggregate
    - Overall lead time
    - Trends in PR review delay
    - Time from merge to deploy
Part 3: What’s next?

- Commit to deploy lifecycle challenge
  - Ease of integration into toolchain
    - Source code repositories
    - Build systems
    - Deploy systems
    - etc.
Part 3: What’s next?

Homework!!!!!
Summary

- Instrumentation and visualization lead to smarter work
- Build systems are an easy first step
Summary

● The tools we use for prod can be used in the SSC
● The SSC is ripe for new insights and visualizations
Summary

- The ideas of observability apply outside of production and bring value to the SSC
Thank you

Ben Hartshorne
maplebed
honeycomb.io