

High-performing engineering teams and the Holy Grail







Jeremy Meiss



Director, DevRel & Community

 **@IAmJerdog**



So back to the tech industry....





YOU SEEK THE HOLY GRAIL.



2016 State of DevOps Report

Presented by



2017 State of DevOps Report

Presented by:



puppet



DEVOPS RESEARCH & ASSESSMENT

Sponsored by:



splunk>



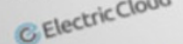
amazon



Atlassian



Deloitte



Electric Cloud

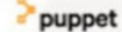


WAVEFRONT

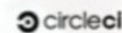
by VIMVOVO

2019 State of DevOps Report

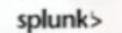
Presented by



puppet



circleci

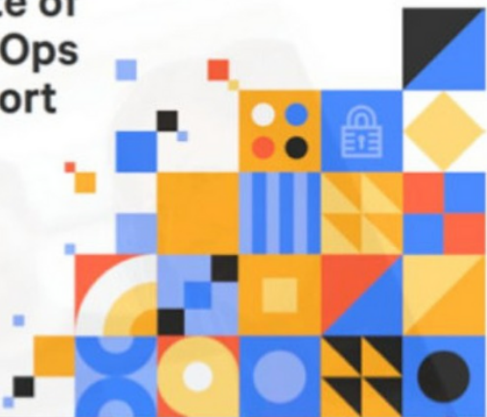


splunk>

Sponsored by

ANITIAN

ServiceNow



A white rectangular card is placed on a teal, textured background that resembles denim fabric. The card is slightly tilted and contains the text "ONE SIZE DOESN'T FIT ALL". The words "ONE SIZE" and "FIT ALL" are in black, while "DOESN'T" is in red.

**ONE SIZE
DOESN'T
FIT ALL**

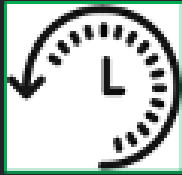


**THE HOLY HAND GRENADE FOR
HIGH-PERFORMING ENGINEERING TEAMS**

CI/CD Benchmarks for high-performing teams



Duration



Mean time
to recovery



Success
rate



Throughput





So what does the
data say?

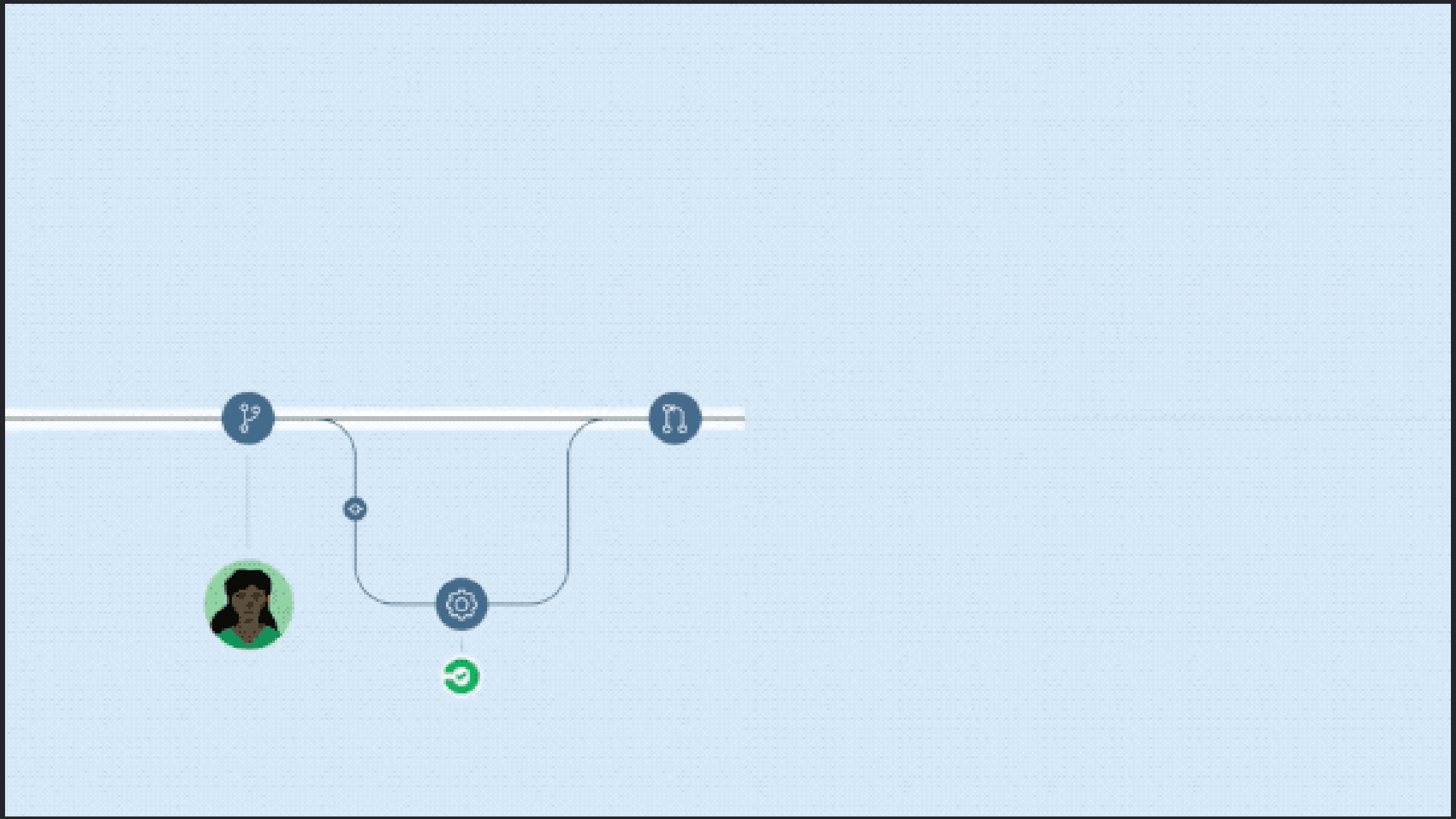
Duration

the foundation of software engineering velocity, measures the average time in minutes required to move a unit of work through your pipeline



And There Was Much Rejoicing





So what is an ideal Duration?

≤ 10 minute builds

"a good rule of thumb is to keep your builds to no more than ten minutes. Many developers who use CI follow the practice of not moving on to the next task until their most recent checkin integrates successfully. Therefore, builds taking longer than ten minutes can interrupt their flow."

-- Paul M. Duvall (2007). Continuous Integration: Improving Software Quality and Reducing Risk

Duration: What the data shows

Workflows	Duration
50%	≤ 3.3 mins
75%	< 9 mins
Avg	~ 11 mins
95th percentile	≥ 27 mins

Benchmark: 5-10mins

Improving test coverage

- Add unit, integration, UI, and end-to-end testing across all app layers
- Incorporate code coverage tools into pipelines to identify inadequate testing
- Include static and dynamic security scans to catch vulnerabilities
- Incorporate TDD practices by writing tests during design phase



Optimizing your pipelines

- Use **test splitting** and **parallelism** to execute multiple tests simultaneously
- Cache dependencies and other data to avoid rebuilding unchanged portions
- Use Docker images custom made for CI environments
- Choose the right machine size for your needs



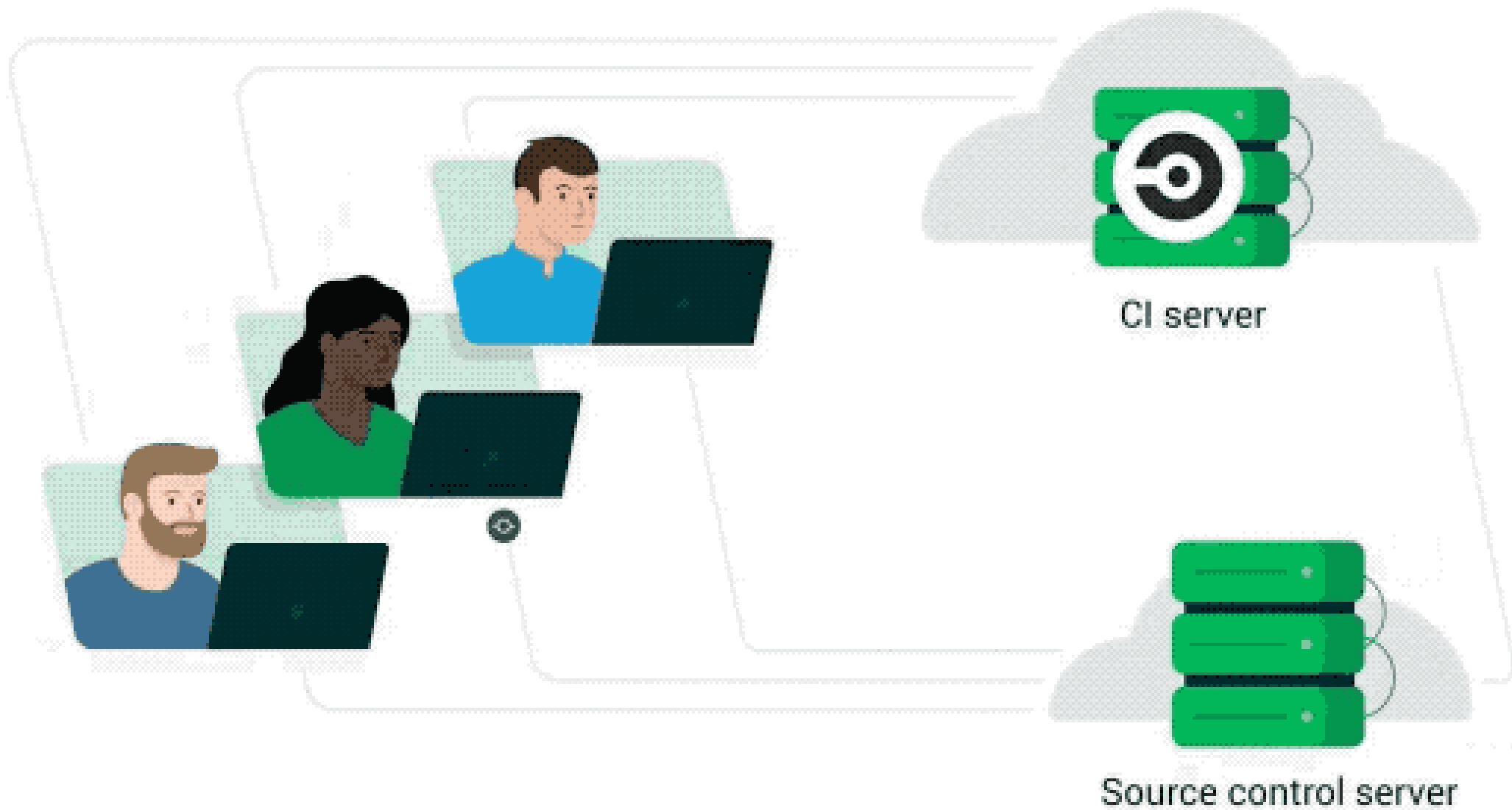
Duration and the Platform Team

- Identify and eliminate impediments to developer velocity
- Set guardrails and enforce quality standards across projects
- Standardize test suites and CI pipeline configs, i.e. shareable config templates and policies
- Welcome failed pipelines, i.e. fast failure
- Actively monitor, streamline, and parallelize pipelines across the org

Mean time to Recovery

*the average time required to go from a failed build
signal to a successful pipeline run*

Mean time to recovery is
indicative of resilience



"A key part of doing a continuous build is that if the mainline build fails, it needs to be fixed right away. The whole point of working with CI is that you're always developing on a known stable base."

-- Fowler, Martin. "Continuous Integration." Web blog post. MartinFowler.com. 1 May 2006. Web.



So what MTTR is ideal?

≤ 60 min MTTR on
default branches

MTTR: What the data shows

Workflows	TTR
50%	<=64 mins
top 25%	<=15 mins
top 5%	<=5 mins
75th percentile	<=22 hrs

Benchmark: 60mins

“10 minutes is a striking
improvement - what happened?”

Two factors impacting reduced MTTR

- Economic pressures in the macro environment + rising competition in the micro environment, forcing teams to prioritize product stability and reliability over growth
- High performers increasingly rely on platform teams to achieve steadier and more resilient development pipelines with built-in recovery mechanisms.

Treat your default branch as the
lifeblood of your project



Getting to faster recovery times

- Treat your default branch as the lifeblood of your project
- Set up instant alerts for failed builds using services like Slack, Twilio, or Pagerduty.
- Write clear, informative error messages for your tests that allow you to quickly diagnose the problem and focus your efforts in the right place.
- SSH into the failed build machine to debug in the remote test environment. Doing so gives you access to valuable troubleshooting resources, including log files, running processes, and directory paths.

MTTR and the Platform Team

- Emphasise the value of deploy-ready, default branches, with clear processes & expectations for failure recovery across all projects
- Set up effective monitoring and alerting systems, and track recovery time
- Limit frequency and severity of broken builds with role-based AC and config policies
- Config- and Infrastructure-as-Code tools limit potential for misconfig errors
- Actively monitor, streamline, and parallelize pipelines across the org

Success Rate

*number of passing runs divided by the total
number of runs over a period of time*

now go away...



...or i will
taunt you a
second time!

So what Success rate is ideal?

90%+ Success rate on
default branches

Success rate: What the data shows

Workflows	Success rate
avg on default	77%
avg on non-default	67%

Benchmark: 90%+ on default

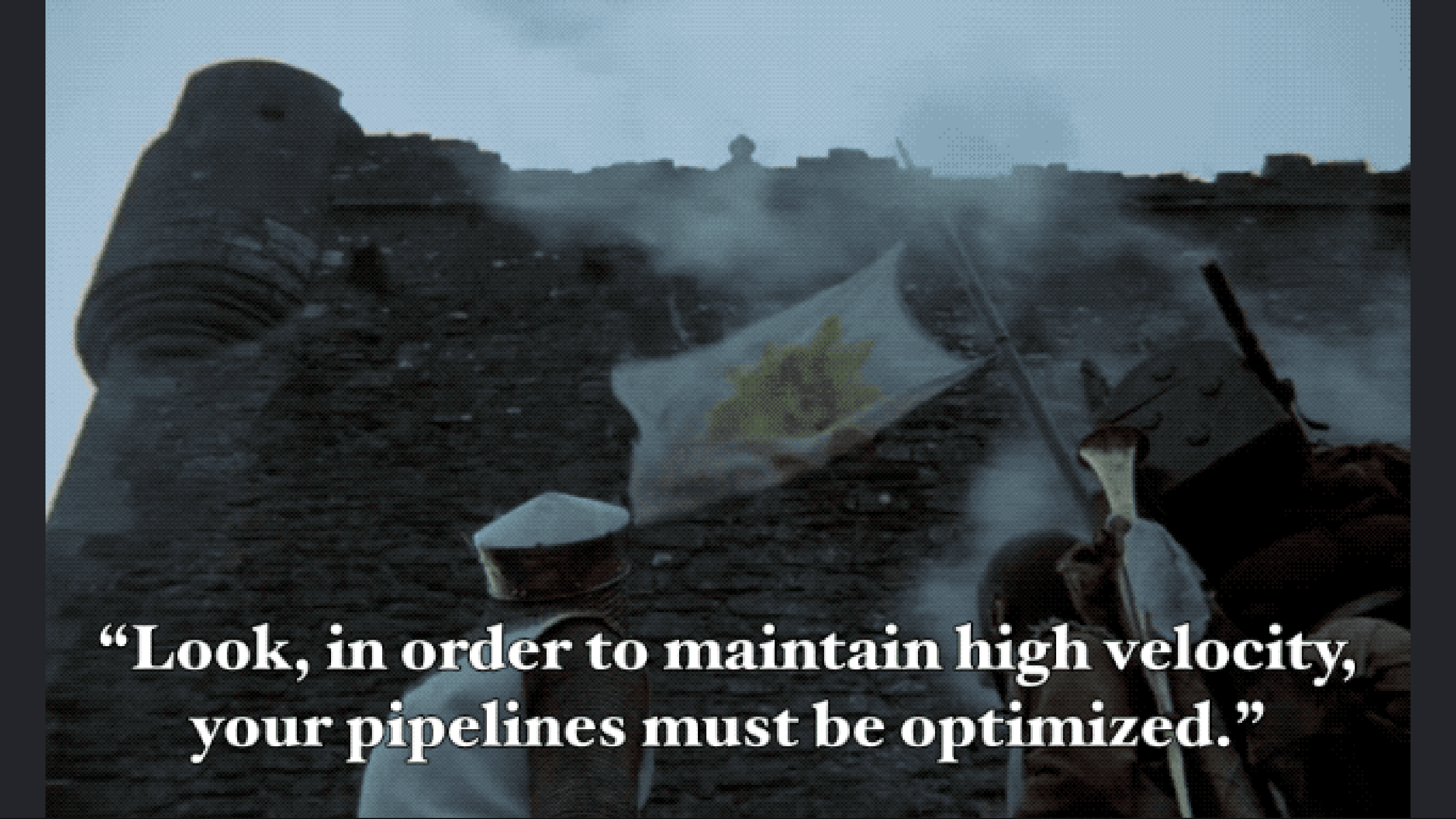


Success rate and the Platform Team

- With low success rates, look at your MTTR and shorten recovery time first
- Set a baseline success rate, then aim for continuous improvement, looking for flaky tests or gaps in test coverage
- Be mindful of patterns and influence of external factors, i.e. decline on Fridays, holidays, etc.

Throughput

*average number of workflow runs that an organization
completes on a given project per day*

A historical photograph of a battle scene. In the foreground, a flag with a yellow star on a white field is partially visible. To the right, a soldier's head is seen in profile, wearing a dark helmet. The background shows a cityscape with buildings and smoke rising from the ground, suggesting a recent battle. The overall tone is somber and historical.

**“Look, in order to maintain high velocity,
your pipelines must be optimized.”**



It's only a model.

So what Throughput is ideal?

It depends.

Throughput: What the data shows

Workflows	Throughput
median	1.54/day
top 5%	7/day
average	2.93/day

Benchmark: at the speed of your business



Throughput and the Platform Team

- Map goals to reality of internal and external business situations, i.e. customer expectations, competitive landscape, codebase complexity, etc.
- Capture a baseline, monitor for deviations
- Alleviate as much developer cognitive load from day-to-day work

High-Performing Teams in 2023

Metric	2020	2022	2023	Benchmark
Duration	4.0 minutes	3.7 minutes	3.3 minutes	10 minutes
TTR	72.9 minutes	73.6 minutes	64.3 minutes	<60 minutes
Success Rate	Avg 78% on default	Avg 77% on default	Avg 77% on default	Average >90% on default
Throughput	1.46 times per day	1.43 times per day	1.52 times per day	As often as your business requires - not a function of your tooling

*"Surely <insert programming language>
helps me achieve the "Holy Grail"!?"*

Top Language			
1	TypeScript	13	Jupyter Notebook
2	Python	14	C#
3	JavaScript	15	Scala
4	Ruby	16	Vue
5	Go	17	Elixir
6	Java	18	C++
7	PHP	19	Clojure
8	Kotlin	20	Rust
9	HCL	21	CSS
10	Shell	22	Gherkin
11	Swift	23	Makefile
12	HTML	24	Jsonnet
		25	Dart

Top Language by Duration			
1	Makefile	13	Liquid
2	LookML	14	VCL
3	Shell	15	EJS
4	HCL	16	Jinja
5	Mustache	17	PLSQL
6	Nix	18	PowerShell
7	SaltStack	19	SCSS
8	Open Policy Agent	20	Haml
9	Smarty	21	R
10	Dockerfile	22	CSS
11	Jsonnet	23	Python
12	Batchfile	24	C#
		25	Vue

Top Language by MTTR			
1	Gherkin	13	CSS
2	JavaScript	14	Elixir
3	PHP	15	Vue
4	HCL	16	Shell
5	Go	17	Kotlin
6	Ruby	18	C#
7	TypeScript	19	Rust
8	Perl	20	Dart
9	Python	21	Jupyter Notebook
10	HTML	22	Jinja
11	Java	23	PLSQL
12	Clojure	24	C
		25	C++

Top Language by Success Rate			
1	Mustache	13	Jupyter Notebook
2	Perl	14	Rust
3	Smarty	15	RobotFramework
4	Go	16	C#
5	PLSQL	17	Python
6	HCL	18	Clojure
7	Vue	19	TypeScript
8	Scala	20	Ruby
9	Makefile	21	Jinja
10	Elixir	22	C
11	Shell	23	PHP
12	HTML	24	Kotlin
		25	Dockerfile

Top Language by Throughput			
1	Hack	13	Haskell
2	Jsonnet	14	Starlark
3	Dart	15	PLSQL
4	Swift	16	Jinja
5	Elixir	17	Lua
6	Ruby	18	HTML
7	Mustache	19	Clojure
8	Jupyter Notebook	20	Apex
9	TypeScript	21	XSLT
10	Python	22	Perl
11	Elm	23	C++
12	Liquid	24	PureScript
		25	Gherkin

2020 Report



<https://circle.ci/ssd2020>

Full 2022 Report



<https://circle.ci/ssd2022>

Thank You.

For feedback and swag: circle.ci/jeremy



timeline.jerdog.me



[IAmJerdog](https://twitter.com/IAmJerdog)



[jerdog](https://dev.to/jerdog)



[/in/jeremy.me](https://in.jeremy.me)



@jerdog@hachyderm.io