

Dynamic Analysis

17-313 Fall 2026

Foundations of Software Engineering

<https://cmu-17313q.github.io>

Eduardo Feo Flushing

Administrivia

- Midterm grades will be posted tomorrow at noon.
- P3 will be posted tonight.
- Check your email: you will receive instructions to access a VM.

Learning Goals

- Explain why diagnosing a performance problem requires a workload, a baseline, repeatable measurements, and a way to locate the bottleneck.
- Interpret the essential features of a flame graph and choose between profiling, stress testing, and soak testing for a given symptom.
- Explain how telemetry supports reliability experiments and outline a bounded chaos experiment with a steady-state measure.
- Design a basic A/B test with random assignment, a primary outcome, and a guardrail; explain what its results would and would not establish.

Automated Analysis for Functional and Non-Functional Properties

Functional: does it do the right thing? (correctness). Non-functional: how well does it do it? (quality attributes).

■ Static Analysis

Formatting Linters ✓

Pattern-Based Linters ✓

Type-Based Analysis ✓

Taint Analysis ✓

■ Dynamic Analysis

Correctness – Testing ✓

Robustness – Fuzzing ✓

Performance – Profiling

Scalability – Stress testing

Stability – Soak testing

Reliability – Chaos Engineering

Usability – A/B testing

Outline

- **Performance under different conditions**
 - **Efficiency: Profiling and tracing**
 - **Scalability: Stress testing**
 - **Stability over time: Soak testing**
- Testing in production
 - Reliability and resilience: Chaos engineering
 - Usability: A/B testing

Performance under different conditions

Performance Testing

- Goal: identify **performance bugs**. What are these?

Unexpected bad performance on some subset of inputs.

Performance degradation over time.

Difference in performance across versions or platforms.

- Not as easy as functional testing. What's the baseline?

Fast = good, slow = bad ... but what's the threshold?

How do we get reliable measurements?

How do we debug where the issue lies?

A run took 1.8 seconds. Is that a bug?

Performance regression testing helps identify trends

- Measure execution time of critical components.
- Log execution times and compare over time.

Job 12e96643840000

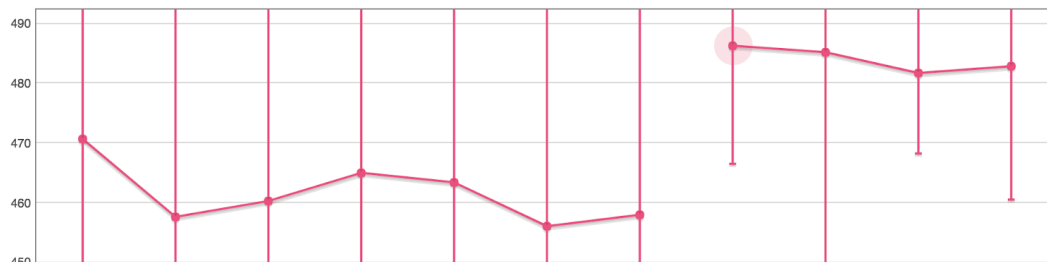
[Issue 808613](#) · [Analyze benchmark results](#) · 2.0 hours · 2/14/2018, 9:48:34 AM

Differences found after commits

[Re-record loading.desktop story set](#) by
ksakamoto@chromium.org

Job arguments

benchmark loading.desktop
chart cpuTimeToFirstMeaningfulPaint
configuration chromium-rel-mac11-pro
statistic avg
story Pantip
target telemetry_perf_tests
tir_label warm
trace Pantip



[Re-record loading.desktop story set](#) by ksakamoto@chromium.org

Build

Test

Values

builder	Mac Builder	task_id	3baea4beaa7f1710	trace	Pantip_2018-02-14_11-40-07_93865.html
isolate_hash	630b5fe7ae1b260e78db88233099249b5640517b	bot_id	build197-b4	trace	Pantip_2018-02-14_11-40-42_21734.html
		isolate_hash	146eb87de6d2594cc3a9ee9f3518f69fc3d0c2c3		

Performance under different conditions

Testing in production

Performance bugs are "bad" bugs

- Fixing performance bugs is usually more difficult than fixing non-performance bugs.
- Performance bugs usually don't generate incorrect results or crashes.
- Difficult to diagnose: system load, hardware configuration, network conditions, user-specific workflows, interactions with other systems.
- Big impact on user experience.

Discovering, Reporting, and Fixing Performance Bugs

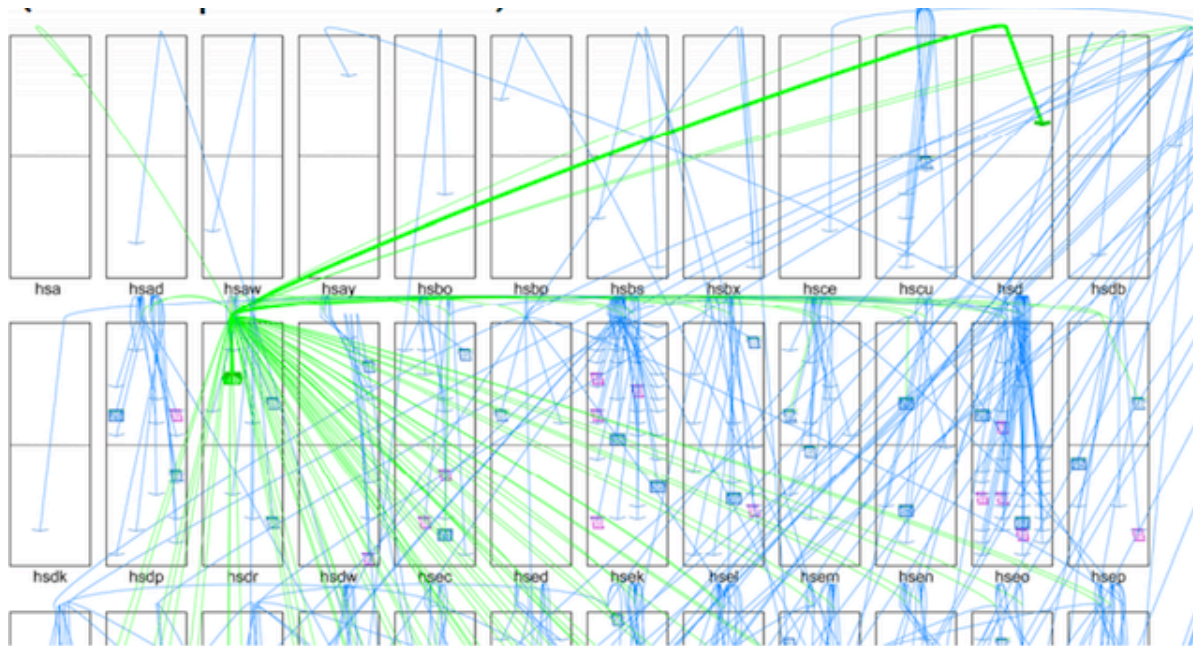
Adrian Nistor¹, Tian Jiang², and Lin Tan²

¹University of Illinois at Urbana-Champaign, ²University of Waterloo
nistor1@illinois.edu, {t2jiang, lintan}@uwaterloo.ca



Nistor, Jiang & Tan. "Discovering, Reporting, and Fixing Performance Bugs."

"Bad Bug" Story: A search query in Google Data Centers



Dick Sites, "Data Center Computers: Modern Challenges in CPU Design." A single search query fans out across many machines; a slowdown anywhere on this path can slow the whole response.

How to catch bad performance bugs

Performance analysis via instrumentation

- Embedding additional code to monitor the program's behavior.
- Usage:

Source code (**static**): additional instructions for data collection.

Binary files (**dynamic**): inserting monitoring code at runtime without altering the source.

- Applications:

Profiling: execution time, call frequency, resource usage.

Tracing: detailed execution flow, entries/exits, event sequences.

```
function factorial(n) {  
  // log function entry  
  console.log(  
    `Starting factorial calculation for ${n}`  
  );  
  let start = performance.now();  
  let result = 1;  
  for (let i = 1; i <= n; i++) {  
    result *= i;  
  }  
  // log execution time and function exit  
  let end = performance.now();  
  console.log(  
    `Factorial of ${n} calculated. Result: ${result}`  
  );  
  console.log(  
    `Time taken: ${end - start} milliseconds`  
  );  
  return result;  
}
```

Profiling: where does the time go?

- The program produces the correct result, but takes too long. Which part is responsible?
- **Profiling** measures a program while it runs to find where it spends time.
- **Goal:** Identify functions or execution paths that account for substantial time: potential bottlenecks.
- **One approach:** rather than timing every function call, a sampling profiler periodically records the active call stack.

What does the active call stack tell us?

Sampling call stacks

```
def f(): return 1
def d(): return 1

def e():
    return f()

def c():
    d()
    e()
    return f()

def b(): return c()

def a():
    b()
    return f()

while True:
    a()
```

Each sampled call stack is a snapshot of which function was executing, and who called it, at that moment. Call paths that appear in more samples tend to account for more execution time.

Flame Graphs

The Flame Graph

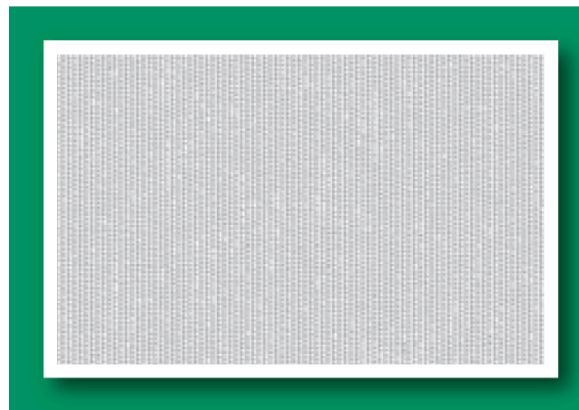
THIS
VISUALIZATION
OF SOFTWARE
EXECUTION
IS A NEW
NECESSITY FOR
PERFORMANCE

BRENDAN GREGG, NETFLIX



An everyday problem in our industry is understanding how software is consuming resources, particularly CPUs. What exactly is consuming how much, and how did this change since the last software version?

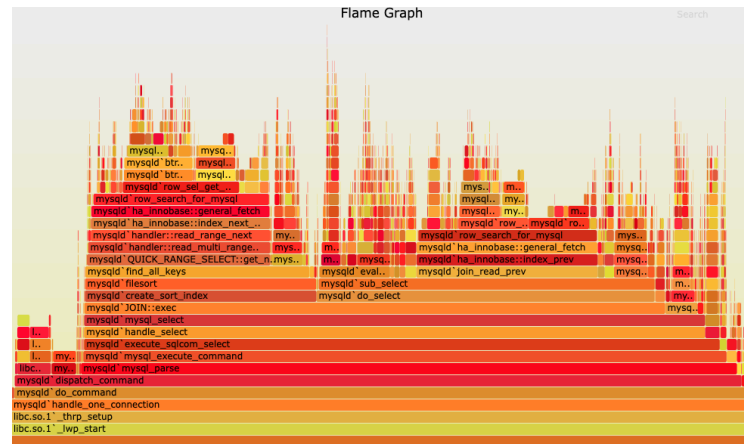
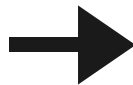
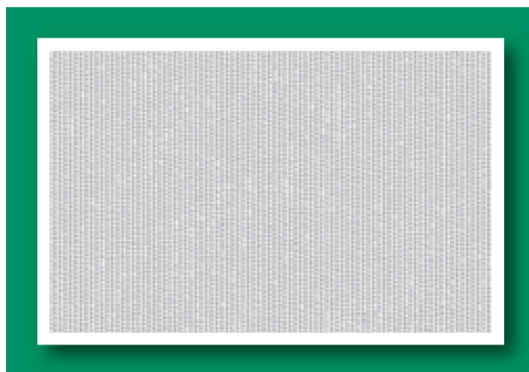
FIGURE 3: FULL MYSQL DTRACE PROFILE OUTPUT



Brendan Gregg, Netflix: "An everyday problem in our industry is understanding how software is consuming resources, particularly CPUs. What exactly is consuming how much, and how did this change since the last software version?" A raw dump of sampled stack traces (right) is unreadable at scale.

Flame Graphs

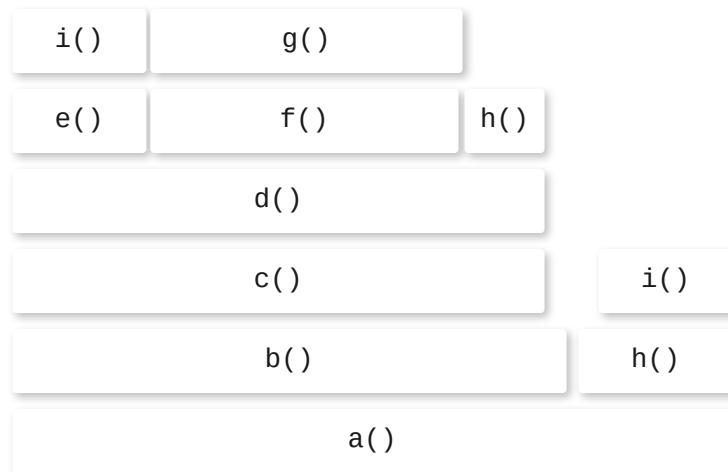
FIGURE 3: FULL MYSQL DTRACE PROFILE OUTPUT



Thousands of sampled stack traces, aggregated and visualized as one graph.

How to read a Flame Graph?

- Each box is a function call.
- **Top-down shows ancestry.** A box's children are the functions it called.
- **Box width is proportional to presence in the sampled stack traces...** not chronological duration or left-to-right order.
- The same function can appear in more than one box: `h()` and `i()` were each called from two different places.



Q. What does the flame graph for this code look like?

```
def f(): return 1
def d(): return 1

def e():
    return f()

def c():
    d()
    e()
    return f()

def b(): return c()

def a():
    b()
    return f()

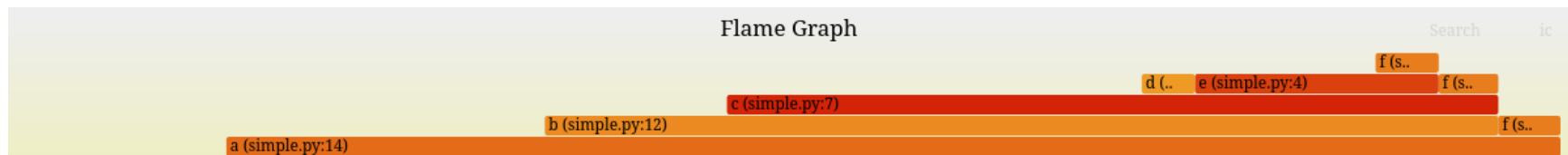
while True:
    a()
```

30 seconds: sketch the shape before the next slide.

Performance under different conditions

Testing in production

Let's profile it



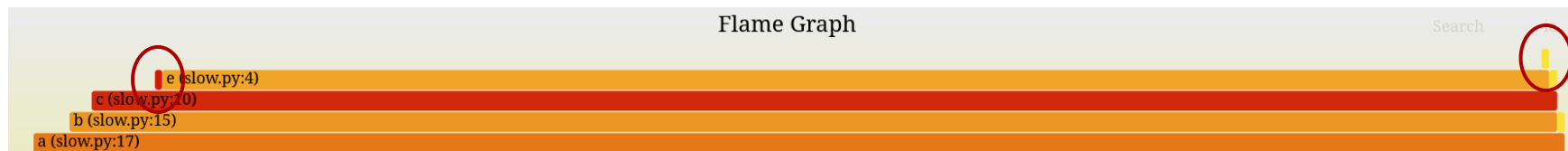
The real flame graph produced by profiling the code from the previous slide.

Q. What does the flame graph for this code look like?

```
def f(): return 1
def d(): return 1
def e():
    uselessSum = 0
    for _ in range(100):
        uselessSum += 1
    return f()
def c():
    d()
    e()
    return f()
def b(): return c()
def a():
    b()
    return f()
while True:
    a()
```

30 seconds: sketch the shape before the next slide.

Let's profile it



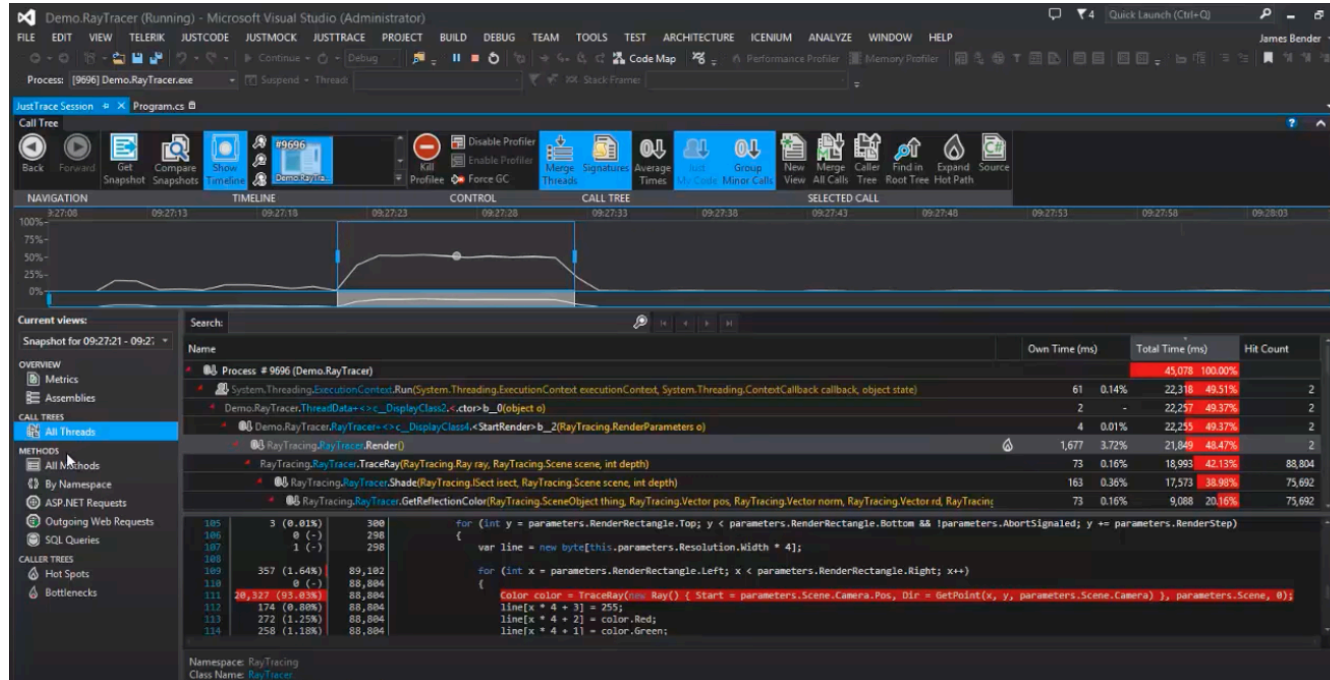
The real flame graph produced by profiling the code from the previous slide. The circled bars are `d()` and `f()`.

`f5()` is wider than `f6()` – `f10()`

- `f5()` is more commonly found at the top of the stack traces when they were collected: it does real work (`useLessSum` loop) before calling onward.
- `f6()` – `f10()` appear less frequently at the top: each just immediately calls the next, so any one of them is rarely the "currently running" frame a sample catches.

The flame graph would show a wide `f5()` box and a narrow stack of `f6()` – `f10()` boxes above it.

Profilers often included in IDEs



Performance under different conditions

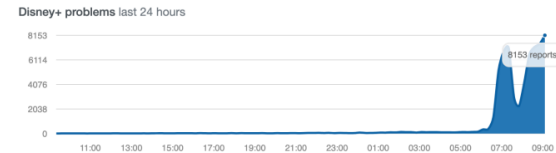
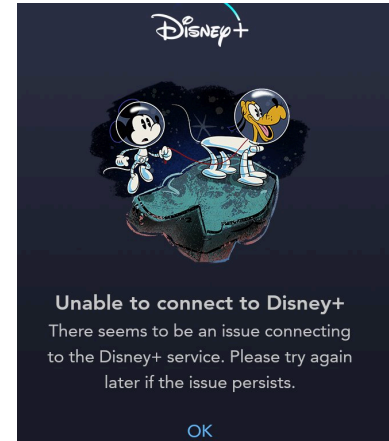
Testing in production

Stress testing

- Scalability/robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Key idea: throw large amounts of input/requests and see how the program behaves.
- Often a way to test the error-handling capabilities of the application.

Real Issues: Disney+ Launch

- Lots of issues reported on launch day.
- Disney had planned for a spike in traffic: tested massive concurrent video streaming capability.
- **But:** the stress was in paths other than streaming:
 - User account creation.
 - Logins and auth.
 - Browsing old titles.



Soak testing

- A system may behave exactly as expected under artificially limited execution conditions, but fail in production after extended use.
E.g., memory leaks may take longer to lead to failure.
- **Soak testing** a system involves applying a significant load over a significant period of time and observing system resilience.
- Time-consuming to run, but useful at big release milestones or when making infrastructure changes.

Activity 1: choose a performance experiment

- **E-commerce:** checkout becomes slow only during a sale.
- **Self-driving car software:** processing latency spikes on complex scenes.
- **Mobile game:** performance steadily degrades after an hour of play.



Front – E-commerce

Middle – Self-driving

Back – Mobile game

In groups of 2–3, pick the scenario for your row:

- How would you design the experiment: profiling, stress testing, or soak testing, and for how long?
- What would you measure?
- What pattern in the results would help you identify the cause?
- Why that technique, and not one of the other two?

5 minutes in groups

Outline

- Performance under different conditions
 - Efficiency: Profiling and tracing
 - Scalability: Stress testing
 - Stability over time: Soak testing
- Testing in production
 - Reliability and resilience: Chaos engineering
 - Usability: A/B testing

Testing in production

Beta testing

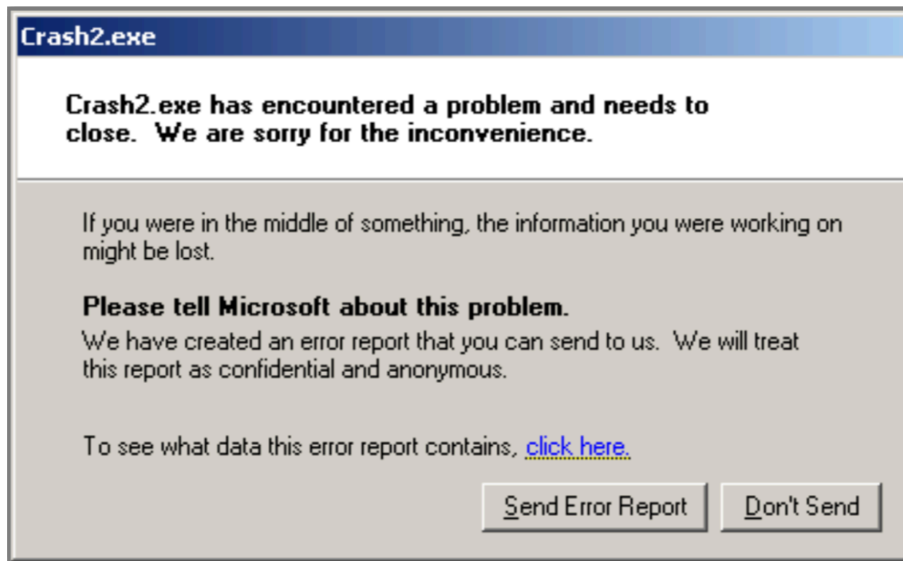


Real users, before broad release, on real hardware and real workflows.

Performance under different conditions

Testing in production

Telemetry



Low-overhead signals collected from deployed use: error rates, latency, crash reports.

Catching performance bugs



- Observation in natural environment
- Real-time data collection
- Understanding ecosystem impact
- Non-invasive techniques
- Behavioral pattern analysis



- Observation in real-time operation
- Monitoring system performance & resources
- Analyzing software's interaction with its surroundings
- Minimal impact on running application
- Detecting anomalies and performance issues

Beta testing and telemetry are low-overhead monitoring: they let us see what's actually happening in the natural environment, not just in staging.

Reliability testing

- What happens when some components of a large, complex system fail?
Can the system recover and keep working?
- How can you test the reliability of something as complex as Netflix, Google Maps, or Instagram?
- One idea: simulate a large-scale deployment and induce random failures in various components.
- Another idea: test in production with **Chaos Engineering**.

What is chaos engineering?

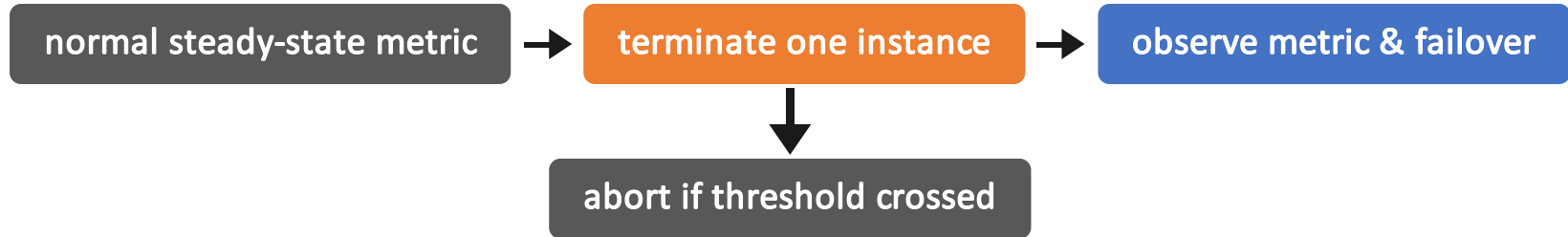
"Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."

principlesofchaos.org

Chaos Engineering: Testing in Production

- Purposefully take down components in a **live deployment**.
- Observe system response. Do failovers work correctly?
- Tests the failure-handling and fallback capabilities of large systems.
- Useful in preparing for natural disasters or cyberattacks.

A chaos experiment needs a prediction.



Example: Google

- Terminate network in Sao Paulo for testing:
Hidden dependency takes down links in Mexico, which would have remained undiscovered without testing.
- Turn off a data center to find that machines won't come back:
Ran out of DHCP leases (for IP address allocation) when a large number of machines came back online unexpectedly.

A bounded failure revealed a hidden dependency that code review and staging tests never would have.

Why would you break things on purpose?

Firefighters train in realistic, high-stress simulations. Sometimes so real they aren't even told it's a drill.



Automating GUI/Web Testing

- This is hard.
- Capture and replay strategy: mouse actions, system events.
- Test scripts: click on button labeled "Start", expect value X in field Y.
- Lots of tools and frameworks (e.g., Selenium for browsers).
- Can avoid load on GUI testing by separating model from GUI.
- Beyond functional correctness?

Usability: A/B Testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.

Activity 2: Design an A/B Test

A pet store is promoting the same sale in an email. It tests two versions that differ only in the photo. The goal is to increase purchases.



Variant A

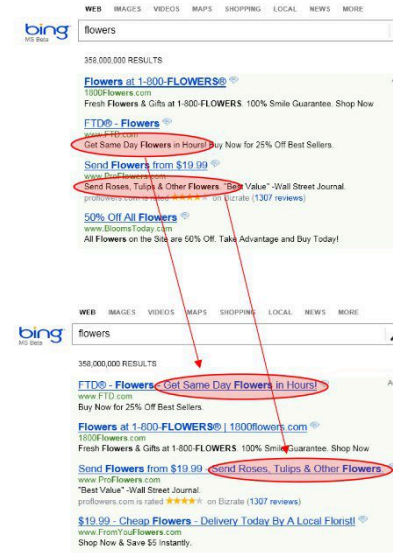
Variant B

1. What is the **primary measure**? What else would you monitor?
2. How would you assign customers to A and B? What must stay the same?
3. B gets more clicks but fewer purchases per emailed customer. Would you use B? What else do you need to know?

30 seconds to look at A/B; 2 minutes paired discussion; 2 minutes to collect answers.

Bing Experiment

- Experiment: ad display at Bing.
- Suggestion prioritized low; not implemented for 6 months.
- Ran A/B test in production.
- Within 2 hours, revenue-too-high alarm triggered, suggesting a serious bug (e.g., double billing).
- Revenue increase of 12%, about \$100M annually in the US.
- Did not hurt user-experience metrics.



Kohavi, Tang & Xu, "Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing," 2020.

A/B Testing

- Requires monitoring tools and telemetry.
- Requires good metrics and statistical tools to identify significant differences.

E.g., clicks, purchases, video plays.

- Must control for confounding factors.
- Automation:

Stop experiments when confident in results.

Stop experiments resulting in bad outcomes (crashes, very low sales).

Automated reporting, dashboards.

Random assignment, a preplanned analysis/stopping rule, and checking guardrail metrics matter more than stopping as soon as an attractive number appears.

Which technique answers which question?

- Slow for some users? **Profile / trace.**
- Fails at peak load? **Stress test.**
- Fails after hours of use? **Soak test.**
- Depends on a component staying up? **Chaos experiment.**
- Unsure which interface helps users? **A/B test.**

Match the experiment to the symptom.

Mid-Semester Retrospective

- What should we **start** doing?
- What should we **stop** doing?
- What should we **keep** doing?

<https://forms.gle/UWjQTEqaoiUX8bV3A>

