CS 4530
Software Engineering

Lecture 1 - Course Overview + Introductions

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Today's Agenda

Introductions

Course Mechanics

Discussion: Code style

Code style activity

Zoom Mechanics

- Recording: TBD
- If you feel comfortable having your camera on, please do so! If not: a photo?
- I can see the zoom chat while lecturing, slack while you're in breakout rooms
- If you have a question or comment, please either:
  - "Raise hand" - I will call on you
  - Write "Q: <my question>" in chat - I will answer your question, and might mention your name and ask you a follow-up to make sure your question is addressed
  - Write "SQ: <my question>" in chat - I will answer your question, and not mention your name or expect you to respond verbally

Introductions

- Me
  - Research: Software Engineering, Program Analysis
  - Open source startup: Clowdr (Virtual conferences - React/NodeJS/Vonage/Hasura/Postgres)
- Reminder of coordinated sections with:
  - Professor John Boyland
  - Professor Mitchell Wand

Teaching Assistants


Sagar Madhu Ayi


Joseph Burns


Michael Davinroy


Yuting Gan


Eiki Kan


Satyajit Gokhale


Guneet Kaur


Ben Schultze

Office hours: 5 days a week

Monday: Sagar Madhu Ayi @ 1:30-3:30PM, Yuting Gan @ 4:30PM - 6:30PM
Tuesday: Benjamin Schultze @ 3:00PM - 5:00PM
Wednesday: Joseph Burns @ Noon-2:00pm, Eiki Kan @ 4:35PM - 6:35PM
Thursday: Michael Davinroy @ 1:00PM - 2:00PM
Friday: Michael Davinroy @ 9:00AM - 10:00AM, Guneet Kaur @ 11:00AM- 1:00PM

Introductions [Poll]

https://pollev.com/jbell

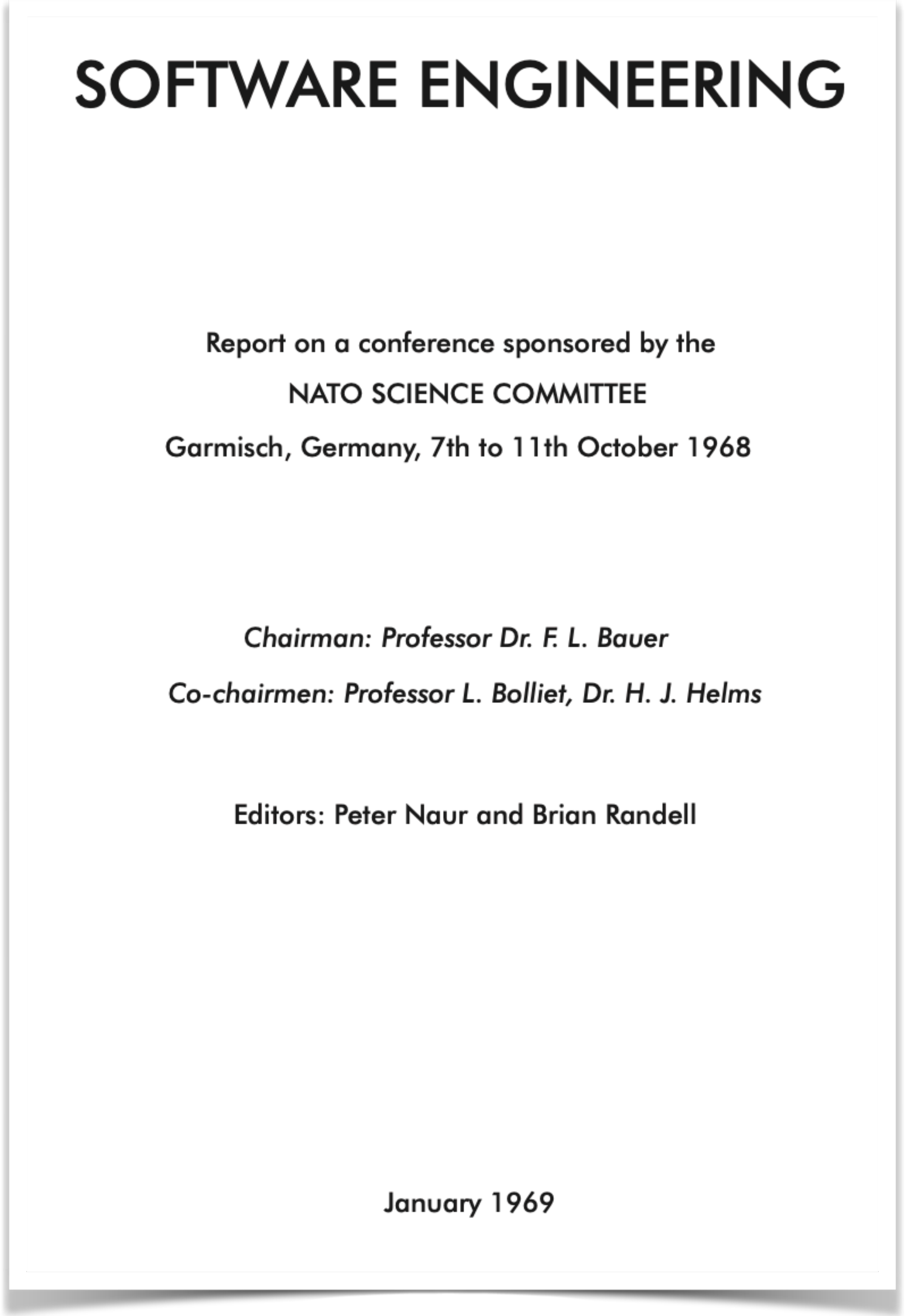
Course Mechanics

- See syllabus for all of the usual stuff
- Our goal is to provide a productive learning environment to both remote and on-the-ground students
- Lecture videos posted at start of week: watch videos before coming to class
- During scheduled class time: discussion, activities. If you come in person, bring laptop and headphones
    - Note: 10% of course grade is based on your participation in these activities
    - Please contact me if you are regularly not able to attend class due to extreme difference in time zone

Software Engineering as a Discipline c. 1969

[Software Engineering as a Class]

- Software was very inefficient
- Software was of low quality
- Software often did not meet requirements
- Projects were unmanageable and code difficult to maintain
- Software was never delivered



SOFTWARE ENGINEERING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE
Garmisch, Germany, 7th to 11th October 1968

Chairman: Professor Dr. F. L. Bauer
Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

A call to action:
We must study
how to build
software

Software Engineering as a Discipline

[Software Engineering as a Class]

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

- Edsger W. Dijkstra, in his 1972 Turing Award acceptance speech

Increase in computational capacity over time

Increase over software complexity?

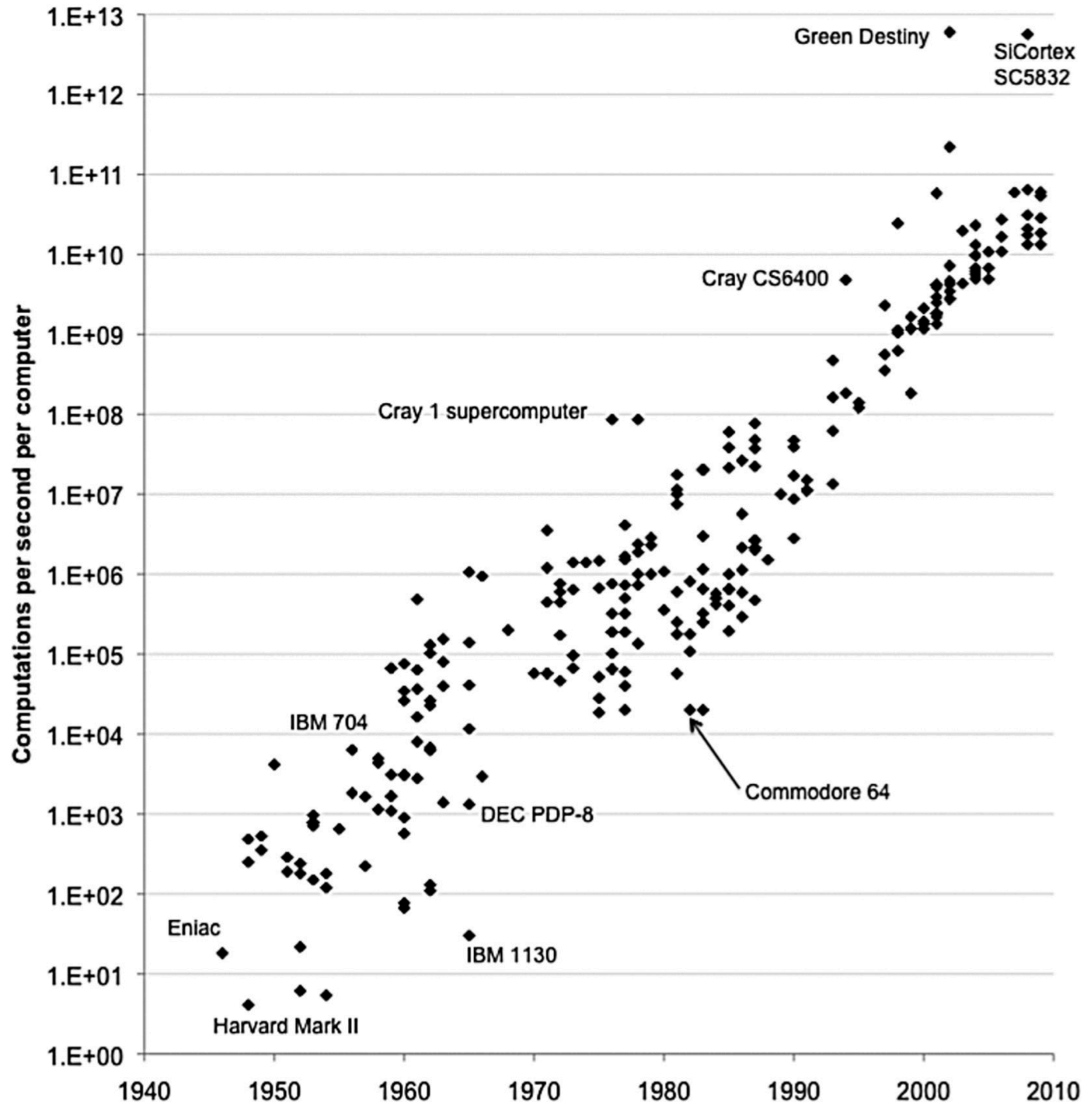
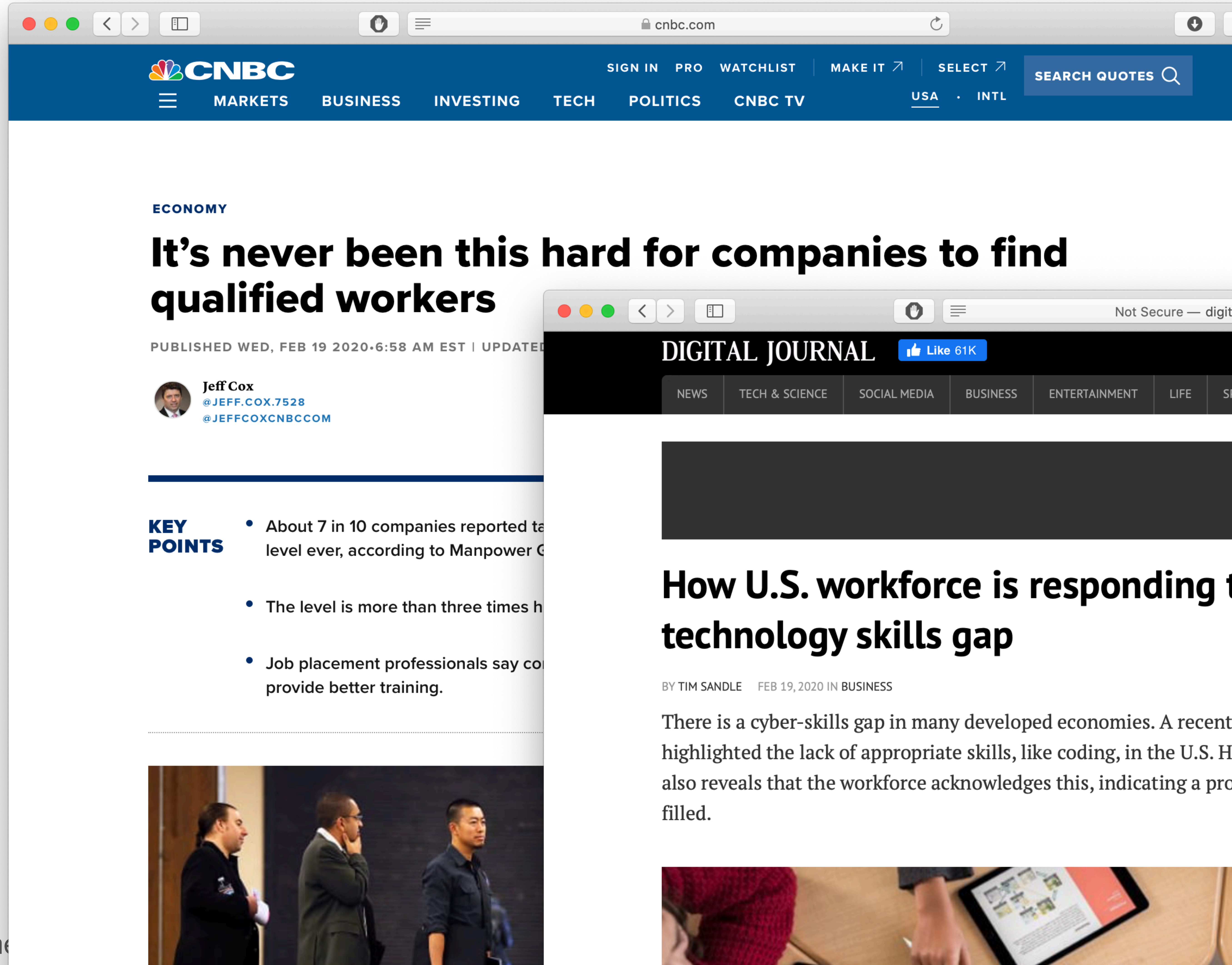


Figure 2. Computational capacity over time (computations/second per computer). These data are based on William D. Nordhaus' 2007 work,[9] with additional data added post-1985 for computers not considered in his study. Doubling time for personal computers only (1975 to 2009) is 1.5 years.



"Implications of Historical Trends in the Electrical Efficiency of Computing" Koome

Software Engineering is about People

Disclaimer: Software Engineering is full of opinions

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"

- Martin Fowler

Why be pedantic about software design?

Software Engineering is about People

```
function calculateFoo(x: number, y: number, increment: boolean): number {
  if (increment)
    x++;
    x *= 2;
  x += y;
  return x;
}

```

`calculateFoo(3, 5, true) = ?` 13

`calculateFoo(3, 5, false) = ?` ~~8~~ 11

Why be pedantic about software design?

What’s wrong with this code?

```
function anotherExample(value: number): void {
  switch (value) {
    case 1:
      doSomething();
    case 2:
      doSomethingElse();
      break;
    default:
      doDefaultThing();
  }
}
```

Software Engineering is about People

Software design is about people

```java
HashSet<String> mySet = new HashSet<String>();
mySet.add("a");
mySet.add("b");
Iterator<String> iter = mySet.iterator();
System.out.println(iter.next()); //What is printed?
```

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

-JavaDoc for HashSet

1,000,000 trials: “a” is printed every time

BUT NOT GUARANTEED

Software Engineering is about People

Software is about People

```
class BookTest {
    @Test
    public void testGetStringRepresentation() {
        Book b = new Book("book", "name");
        assertEquals("{\"title\":\"book\",\"author\":\"name\"}",
                b.getStringRepresentation());
    }
}
```

What could go wrong here?

What if Book is just a HashMap?

```
{
    "title": "book",
    “author": "name"
}
```

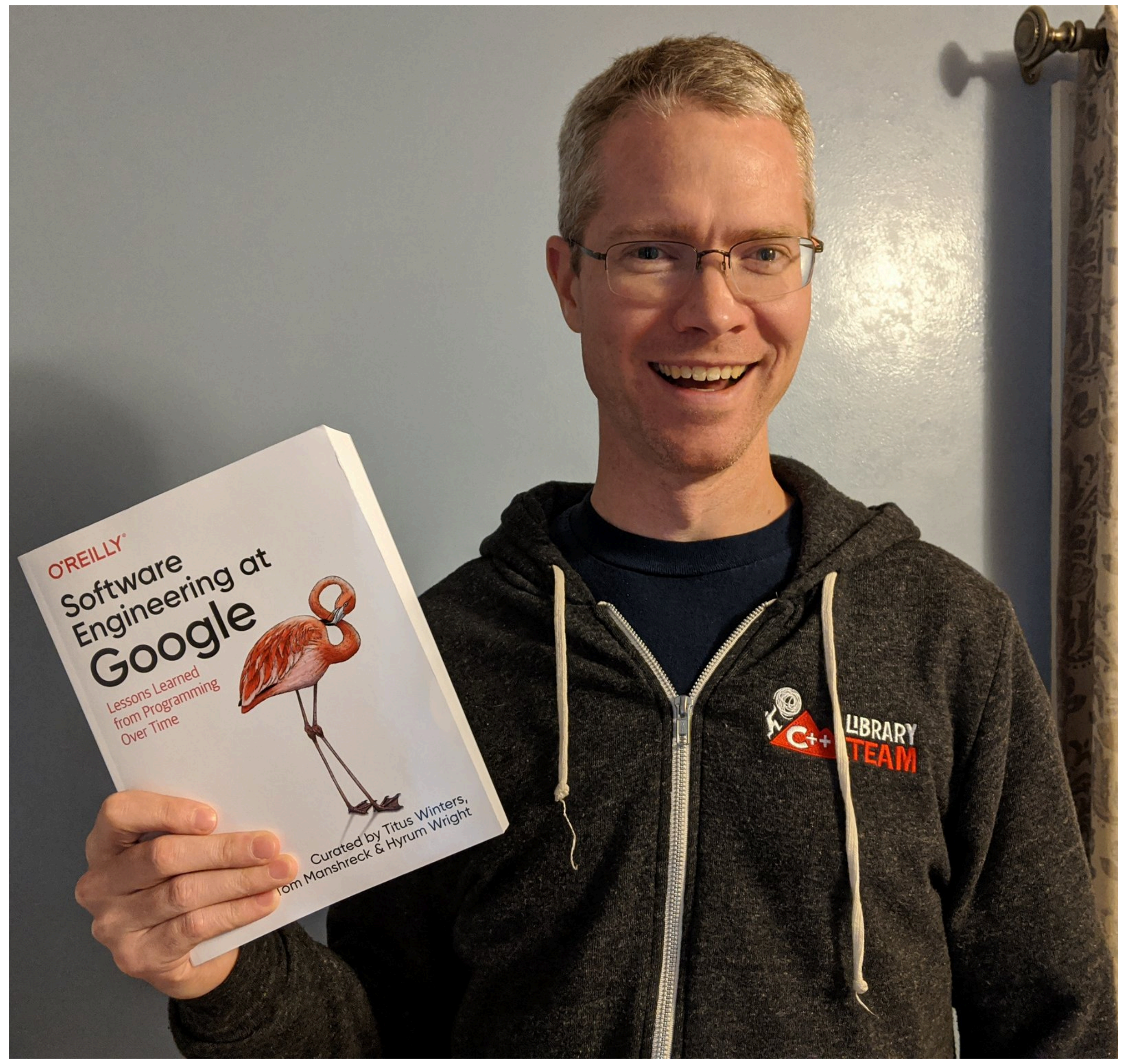
```
{
    “author": “name",
    "title": "book"
}
```

Both are possible :(

Whose fault is this?

What’s Software Engineering’s Answer?

Hyrum’s Law



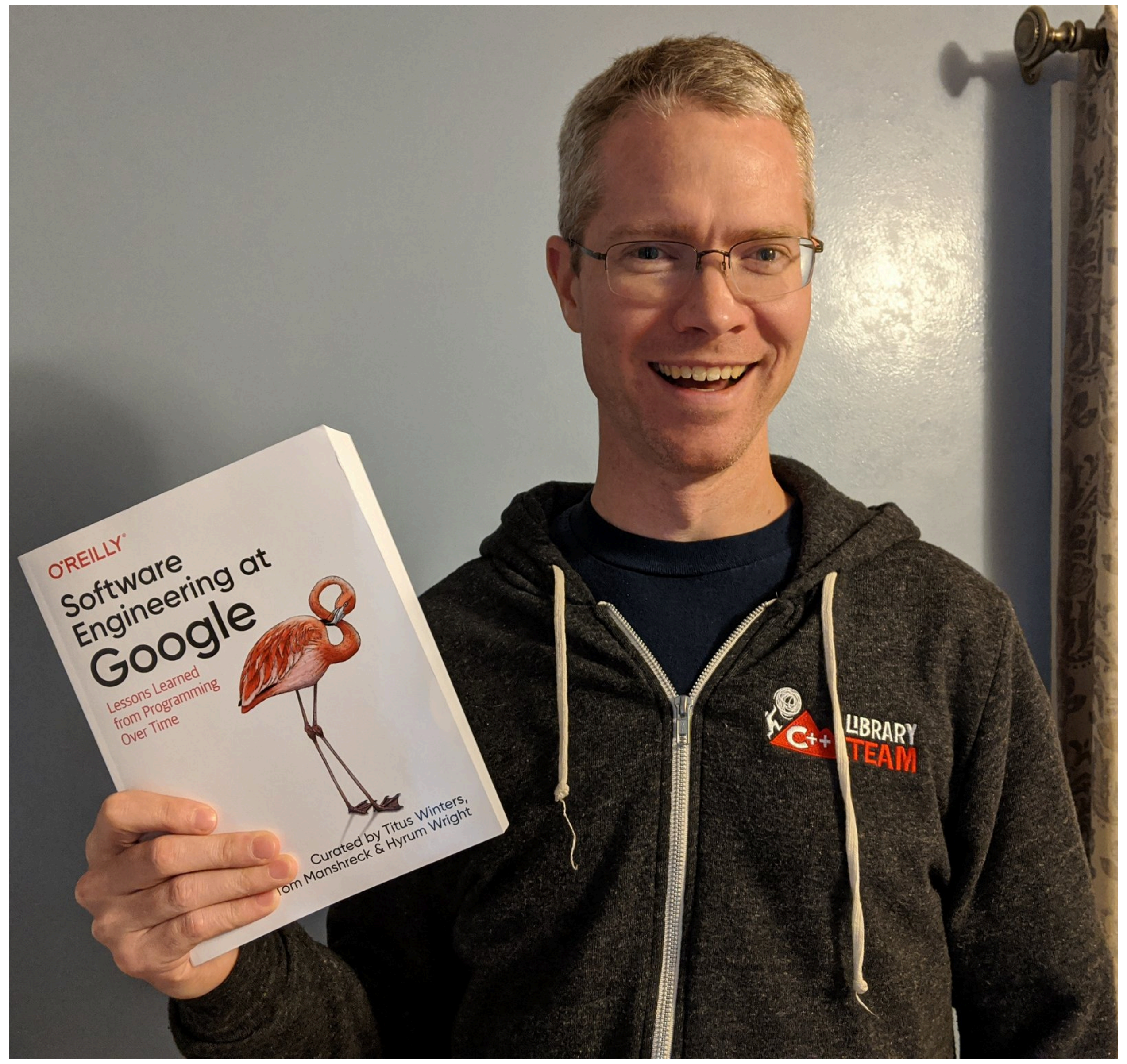
“With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.”

-Hyrum Wright

For more: see “Software Engineering at Google” Ch 1

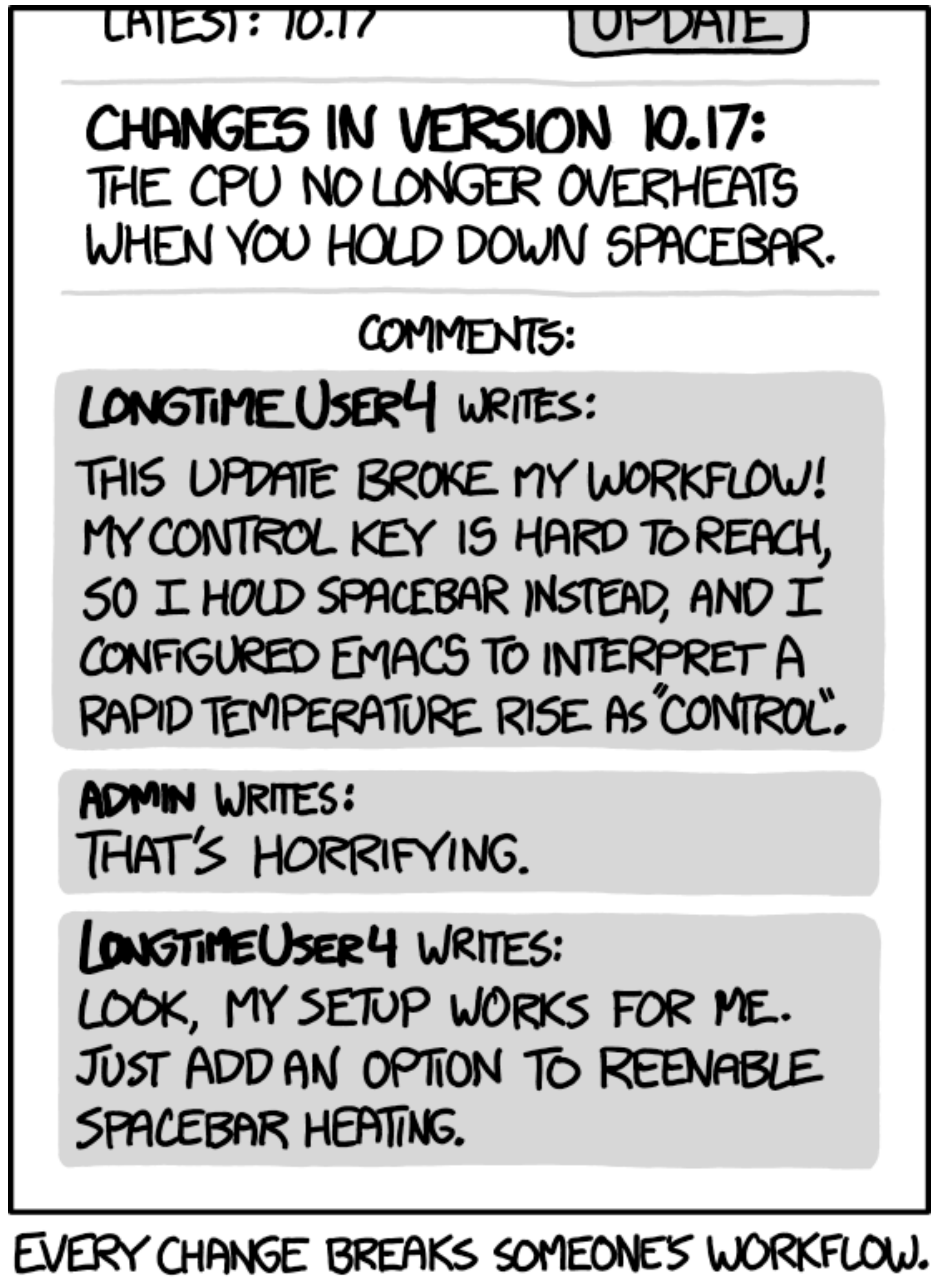
What’s Software Engineering’s Answer?

Hyrum’s Law



“With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.”

-Hyrum Wright



XKCD #1172

What's Software Engineering's Answer?

Automatically detecting this!

2016 IEEE International Conference on Software Testing, Verification and Validation

Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications

August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
Email: {awshi2,gyori,legunse2,marinov}@illinois.edu

***Abstract*—Some commonly used methods have non-deterministic specifications, e.g., iterating through a set can return the elements in any order. However, non-deterministic specifications typically have deterministic implementations, e.g., iterating through two sets constructed in the same way may return their elements in the same order. We use the term *ADINS code* to refer to code that Assumes a Deterministic Implementation of a method with a Non-deterministic Specification. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. Further, ADINS code can lead to *flaky tests*—tests that pass or fail seemingly non-deterministically.**

We present a simple technique, called NonDex, for detecting flaky tests due to ADINS code. We implemented NonDex for Java: we found 31 methods with non-deterministic specifications in the Java Standard Library, manually built non-deterministic models for these methods, and used a modified Java Virtual Machine to explore various non-deterministic choices. We evaluated NonDex on 195 open-source projects from GitHub and 72 student submissions from a programming homework assignment. NonDex detected 60 flaky tests in 21 open-source projects and 110 flaky tests in 34 student submissions.

I. Introduction

Non-deterministic specifications are not uncommon for many methods, including in the standard libraries of many programming languages. For example, the specification for the `Object#hashCode()` method in Java can return any integer. Non-deterministic specifications are not restricted to simple APIs. The order in which elements of a set are returned by an iterator is not-specified—it can be any order. The order in which entries in a SQL table are returned is also sometimes not specified—it depends on the query. Such specifications give implementers more freedom to develop various implementations for different goals, e.g., to optimize performance, while still satisfying the specification.

Even when specifications allow for non-determinism, typical implementations of such specifications are often deterministic, with respect to certain controlled sources. For example, `Object#hashCode()` could return the same integer (if one controls for all other sources, e.g., OpenJDK Java 8 could return a deterministic value on the first call if the under-

code—is often bad. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. For example, Java code that assumes a specific iteration order of a `HashSet`, e.g., that a `HashSet` with elements `1` and `2` will be always represented as a string `{1, 2}` rather than `{2, 1}`, is ADINS and not robust: the Java implementation of `HashSet` can change such that the iteration order of the elements changes and the string differs.

Unexpected behavior of ADINS code can lead to *flaky tests*, which are tests that seem to non-deterministically pass or fail. Flaky tests are bad as they can mask bugs (pass when there are bugs) or raise false alarms (fail when there are no bugs). A test that executes ADINS code can be flaky if it assumes that some values are deterministic even if they can change: when the assumptions hold, the test passes, but when the assumptions do not hold, the test may fail. Not all flaky tests are due to ADINS code, e.g., a test asserting that a file system contains `/tmp` could pass on one machine but fail on another. Flaky tests are emerging as an active research topic, with recent work on characterizing [25], detecting [2], [4], [10], [12], [14], [38], and avoiding [1], [22] flaky tests. However, no previous research investigated ADINS code as a cause for flaky tests.

While flaky tests are an important problem in software *practice* and *research*, we also encountered them in *teaching*. Typically, the teaching staff grades students' solutions to programming assignments using automated tests. These tests can be flaky, and as a result students with correct solutions may have failing tests, and students with incorrect solutions may have passing tests. We discuss more details from one recent course in Section IV-B. Besides educating people about flaky tests, how can we help practitioners in the real world and the students in our courses to detect more flaky tests faster?

We propose a simple technique, called NonDex, to detect flaky tests due to ADINS code. We implement NonDex for Java, but it can be easily generalized to any other language. In a nutshell, we identify 31 methods with non-deterministic specifications as discussed in Section III-A, wrote models for these methods to produce various non-deterministic choices,

https://github.com/TestingResearchIllinois/NonDex

README.md

build passing | build passing | code climate 86 issues | code quality D

NonDex is a tool for detecting and debugging wrong assumptions on under-determined Java APIs. An example of such an assumption is when code assumes the order of iterating through the entries in a java.util.HashMap is in a specific, deterministic order, but the specification for java.util.HashMap is under-determined and states that this iteration order is not guaranteed to be in any particular order. Such assumptions can hurt portability for an application when they are moved to other environments with a different Java runtime. NonDex explores different behaviors of under-determined APIs and reports test failures under different explored behaviors; NonDex only explores behaviors that are allowed by the specification and any test failure indicates an assumption on an under-determined Java API. NonDex helps expose such brittle assumptions to the developers early, so they can fix the assumption before it becomes a problem far in the future and more difficult to fix.

Supported APIs:

The list of supported APIs can be found here

Prerequisites:

```
- Java 8 (Oracle JDK, OpenJDK).
- Surefire present in the POM.
```

Build (Maven):

Why be pedantic about software design?

Software Engineering is about People

```
1. function calculateFoo(x: number, y: number, increment: boolean): number {
2.   if (increment)
3.     x++;
4.     x *= 2;
5.   x += y;
6.   return x;
7. }
8.
```

```
calculateFoo(3, 5, true) = ?
```
13

```
calculateFoo(3, 5, false) = ?
```
~~8~~ 11

Software engineering tools to the rescue!

```
2:3   error   Expected { after 'if' condition             curly
3:5   error   Expected no linebreak before this statement  nonblock-statement-body-position
3:5   error   Unary operator '++' used                    no-plusplus
3:5   error   Assignment to function parameter 'x'        no-param-reassign
4:3   error   Assignment to function parameter 'x'        no-param-reassign
5:3   error   Assignment to function parameter 'x'        no-param-reassign
```

Why be pedantic about software design?

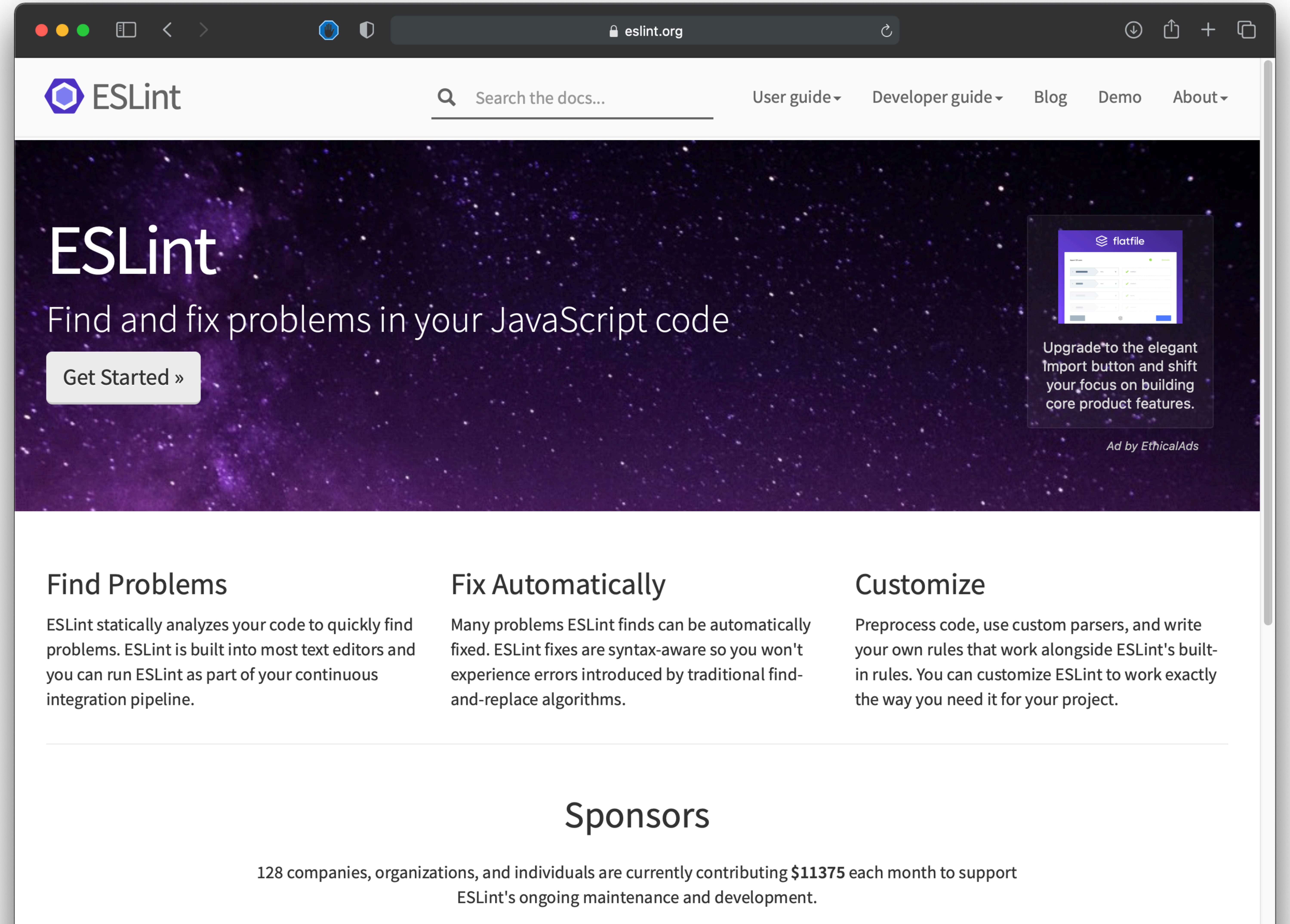
What's wrong with this code?

```
1.function anotherExample(value: number): void {
2.  switch (value) {
3.    case 1:
4.      doSomething();
5.    case 2:
6.      doSomethingElse();
7.      break;
8.    default:
9.      doDefaultThing();
10.  }
11.}
```

```
5:5   error   Expected a 'break' statement before 'case'   no-fallthrough
```

Our first Software Engineering Tool

Linters: your friend, your foe



Review: Design Principles

Five General Principles
1. Use Good Names
2. Design Your Data
3. One method/one job
4. Don't Repeat Yourself
5. Don't Hardcode Things That Are Likely To Change

Five Principles for OO Programming
1. Make Your Interfaces Meaningful
2. Depend only on behaviors, not their implementation
3. Keep Things as Private as You Can
4. Favor Dynamic Dispatch Over Conditionals
5. Favor Interfaces Over Subclassing

Activity: Design Principles and Coding Style

- Right now: Review some of your previous coding projects. This could be a homework, a term project, or something from outside of class, so long as it’s something that you can share. Find one or two examples in your code where either:
  - you used one of the principles and it was helpful
  - you didn't use one of the principles and it would have helped if you'd used it.
- Be prepared to share your code and tell the class
  - what the relevant principle was and
  - how it either helped or would have helped.

Activity: Design Principles and Coding Style

Breakout groups

- In groups of 4, discuss the design and code style issues that you each found
- It's not necessary that all 4 of you present your code to each other, most important is to have a good discussion
- Post your findings in Slack in #section-bell (https://nusespring2021.slack.com)
- After 15 minutes, we'll come back together and share some examples all together

Homework 1 Preview

What we’re building towards…