# ;login:

SUMMER 2020 VOL. 45, NO. 2

usenix®
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

**Finding Anomalies in Linux Performance**
*Richard L. Sites*

**Misaligned Incentives and Bad Software Design**
*Marianne Bellotti*

**Failures in Enterprise SSDs**
*Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder*

**Using ML in Support of SRE**
*Steven Ross and Todd Underwood*

**Interview with Natalie Silvanovich**
*Rik Farrow*

Columns

Culture of Complacency and Crisis
*Laura Nolan*

Using eBPF, Part 2
*Dave Josephsen*

Supporting OSS
*Dan Geer and George P. Sieniawski*

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

## Thanks to our USENIX Supporters!

USENIX appreciates the financial assistance our Supporters provide to subsidize our day-to-day operations and to continue our non-profit mission. Our supporters help ensure:

- Free and open access to technical information
- Student Grants and Diversity Grants to participate in USENIX conferences
- The nexus between academic research and industry practice
- Diversity and representation in the technical workplace

We need you now more than ever! Contact us at sponsorship@usenix.org.

### USENIX PATRONS

Bloomberg FACEBOOK Google

Microsoft NetApp®

### USENIX BENEFACTORS

amazon ORACLE® THINKST CANARY

TWO SIGMA vmware®

### USENIX PARTNERS

ProPrivacy RESTORE PRIVACY TOP10VPN

We offer our heartfelt appreciation to the following sponsors and champions of conference diversity, open access, and our SREcon communities via their sponsorship of multiple conferences:

| | | | |
|---|---|---|---|
| Ethyca | Dropbox | Microsoft Azure | Packet |
| Datadog | Goldman Sachs | LinkedIn | Salesforce |

**More information at www.usenix.org/supporters**

# ;login:

SUMMER 2020 VOL. 45, NO. 2

EDITORIAL

2 **Musings** *Rik Farrow*

FILE SYSTEMS AND STORAGE

6 **A Study of SSD Reliability in Large Scale Enterprise Storage Deployments** *Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder*

13 **SkyhookDM: Data Processing in Ceph with Programmable Storage** *Jeff LeFevre and Carlos Maltzahn*

SECURITY

19 **Interview with Natalie Silvanovich** *Rik Farrow*

PROGRAMMING

22 **Anomalies in Linux Processor Use** *Richard L. Sites*

27 **Revisiting Conway's Law** *Marianne Bellotti*

SRE

30 **Decision-Making Using Service Level Objectives** *Alex Hidalgo*

33 **ML for Operations: Pitfalls, Dead Ends, and Hope** *Steven Ross and Todd Underwood*

COLUMNS

38 **Site Reliability Engineering and the Crisis/Complacency Cycle** *Laura Nolan*

42 **iVoyeur: eBPF Tools** *Dave Josephsen*

45 **Who Will Pay the Piper for Open Source Software Maintenance? Can We Increase Reliability as We Increase Reliance?** *Dan Geer and George P. Sieniawski*

49 **/dev/random: Rewind Your Mind** *Robert G. Ferrell*

BOOKS

51 **Book Reviews** *Mark Lamourine*

USENIX NOTES

54 **Running Virtual PC (vPC) Meetings** *Erez Zadok and Ada Gavrilovska, 2020 USENIX Annual Technical Conference (USENIX ATC '20) Program Co-Chairs*

usenix®
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow

MANAGING EDITOR
Michele Nelson

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Ann Heron
Jasmine Murcia
Olivia Vernetti

TYPESETTER
Linda Davis

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710, USA
Phone: +1 510.528.8649
login@usenix.org

www.usenix.org

*;login:* is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published quarterly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA.

A one-year print subscription to *;login:* magazine is available only to USENIX Association members at the Sustainer level and higher. Periodicals postage is paid at Berkeley, CA, USA, and additional mailing offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA.

EDITORIAL

# Musings

RIK FARROW

Rik is the editor of ;login:.
rik@usenix.org

Cost externalization is usually applied to the use of the environment as a dumping ground for industrial waste. Instead of having to pay for cleaning up sulfur and nitrogen dioxides from burning coal, companies passed those costs along to humans and the world. Turns out, open source software has a parallel problem.

Many of you would not have been alive to experience the ’70s, and too young to be bored by the ’80s. I’m not referring to the culture wars but rather to the acid rain that was eating holes in tree leaves, ruining crops, and hurting wildlife [1]. The yellow smog was particularly bad when combined with summer heat and ozone, limiting visibility in big US cities to one hundred meters on some days.

In the 1990s, the US Congress passed laws forcing coal burners to *scrub* their emissions. That made burning coal more expensive, continuing to do so even in 2020, but eventually made the air a lot cleaner [2] by removing nearly half of the sulfur and nitrogen dioxides from coal burning plants’ exhausts.

Moving costs off of your company’s balance sheet appears to be a fine idea, whether you are communist or capitalist. You can read a short blog entry about cost externalization [3], but all you really need to know is that some companies have artificially lowered their production costs.

I started out by stating that OSS has a similar problem, but I don’t mean air pollution. Many companies use OSS without contributing to its maintenance or creation. They have externalized the costs of programming the software they need to run their businesses to OSS developers—people whom they usually do not pay.

Let’s take a particularly toxic example: the Equifax hack of 2017 [4]. Equifax was using the Apache Struts software as part of the web front-end they used to make money. Equifax, like the other credit agencies, collected data on individuals and families and sold that to potential creditors. In Equifax’s case, attackers took advantage of the Struts framework to invade the Equifax network, and stole data for over 147 million people. The vulnerability they used had been patched in March, while the hack began in July.

I hope that my comparison appears fair to you—that using OSS without contributing to its support is a form of cost externalization. If Equifax had been actively supporting Struts, I believe they would also have been very aware of the vulnerability and would have patched it.

Equifax will be fined and forced to “repay” those whose data had been stolen—to the tune of hundreds of millions of dollars, maybe [5]. Seems like crime does pay, or perhaps externalizing your costs to society works well enough most of the time.

## The Lineup

I was inspired to write about cost externalization after reading Dan Geer and George Sieniawski’s article about paying for the maintenance of OSS. It appears that externalizing costs extends to even tiny code snippets, such as the 11 lines of JavaScript that millions of programmers had been using.

But that's not where we begin in this issue. An award-winning paper about the reliability of enterprise SSDs forms the basis for the first feature article. Maneas et al. used data provided by NetApp to examine four different failure modes seen in SSDs. Interestingly, the failures over time of enterprise SSDs are very different from those of hard drives.

Jeff LeFevre and Carlos Maltzahn explore a way to leverage Ceph, the distributed storage system, to move computation to the location of data. SkyhookDM takes advantage of Ceph's design to distribute data across Ceph's Object Storage Devices so that work, such as SQL queries, can be executed on the system where data resides instead of having to copy all data to a single server first.

I interviewed Natalie Silvanovich, part of Google's Project Zero team. Natalie, the first woman on the team, talked about the techniques she uses while bug-searching, having another woman join the team, and things you should know or learn about if you want to learn more about finding exploitable bugs.

Dick Sites volunteered to write an article demonstrating some uses of his kernel monitoring software, KUtrace. KUtrace provides much finer observations of CPU activity while the CPU executes kernel code. Sites describes four activities where the kernel is wasting a lot of CPU cycles that likely occur commonly enough to be serious issues.

Marianne Bellotti writes about how misaligned incentives result in bad software design. Most people who have heard of Conway's Law know that it describes how designs mirror organizational structures, but Bellotti uncovers a different facet of the law: programmers are incentivized to *stand out,* and that often results in championing their own additions to code that no one else can support.

Alex Hidalgo considers how best to use service level objectives as a tool in decision-making. Hidalgo expresses concerns about how SLOs are becoming buzzwords, when SLOs and SLIs can be used to create more system reliability.

Steve Ross and Todd Underwood take a look at using ML in support of SRE. Both engineers support ML and have often been asked to use ML as part of the support infrastructure. The authors explain machine learning and then point out serious issues with applying ML to SRE tasks.

Laura Nolan focuses her SRE column on decision-making, making that the theme of this issue. Nolan, however, disparages the current culture of complacency that encourages ignoring potential problems until they become crises. Nolan uses the response to the coronavirus pandemic in process as I write this as an example of the crisis/complacency dynamic.

Dave Josephsen, who lives in self-quarantine by choice, continues his exploration of BPF scripts. In particular, Dave explains the `biolatency` script and how it integrates with the block I/O (the *bio*) portion of the Linux kernel.

I've mentioned Geer and Sieniawski's column already as having inspired my musings about externalization. Their focus is actually on how widely OSS is used by enterprises while few take care to actively support the software they use, as they would with commercial software that they pay for.

Mark Lamourine has a container focus this issue, reviewing books about Docker, microservices, and containers. Mark joined the decision-making crowd with reviews of related books.

Robert G. Ferrell was inspired by the failed foray, related by Ross and Underwood, to have ML take over SRE. Robert spins this a bit differently, as he instead discusses how humans and AI will become competitors.

Peter Norton and Chris McEniry didn't write for this issue.

In the world we live and work in, market forces are supposed to rein in cheaters—those who sell bad products or take advantage of our culture to cut costs. In reality, market forces seem to encourage cheating, even when companies that do this get caught in the process. Oil companies are still supported by tax breaks, while companies like Theranos thrive for a while until the illusion they created via marketing dissipates.

And perhaps we should give Equifax a break. After the attack, people feared that identities would be stolen, bank accounts drained, and false tax forms filed. Instead, it turns out the Chinese hackers were to blame [6]. So instead of being worried about identity theft, we should be worried about Chinese intelligence operatives using our personal data to blackmail us.

Oh, well. I bet the people at Equifax weren't thinking about that when they designed their web front-ends using someone else's code.

***References***

[1] "Effects of Acid Rain": https://www.epa.gov/acidrain/effects-acid-rain.

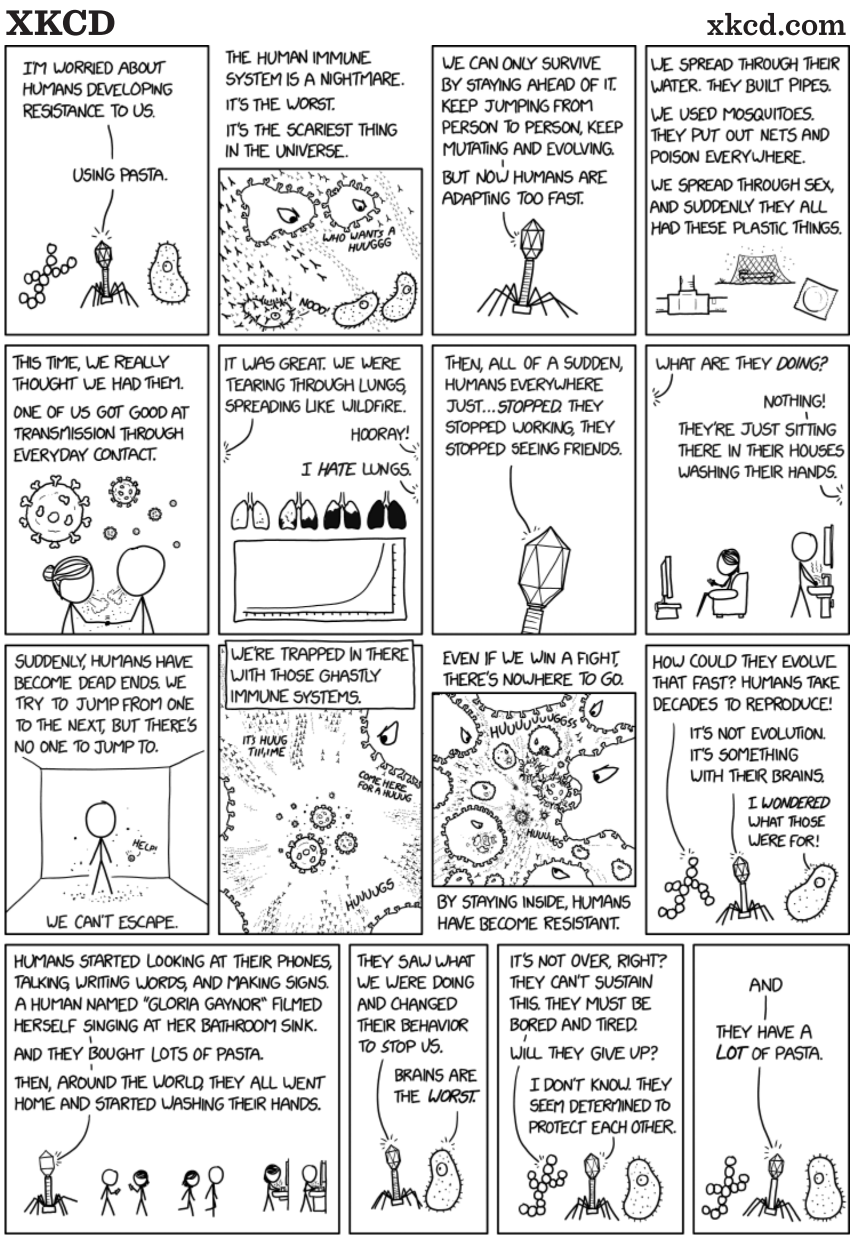
[2] N. S. Rastogi, "Whatever Happened to Acid Rain?" *Slate*, August 2009: https://slate.com/technology/2009/08/whatever-happened-to-acid-rain.html.

[3] J. Whitehead and T. Haab, "ECON 101: Negative Externality": https://www.env-econ.net/negative-externality.html.

[4] L. H. Newman, "Equifax Officially Has No Excuse," *WIRED*, September 14, 2017: https://www.wired.com/story/equifax-breach-no-excuse/.

[5] Z. Whittaker, "FTC Slaps Equifax with a Fine of up to $700 Million for 2017 Data Breach," TechCrunch, July 22, 2019: https://techcrunch.com/2019/07/22/equifax-fine-ftc/.

[6] B. Krebs, "U.S. Charges 4 Chinese Military Officers in 2017 Equifax Hack," Krebs on Security, February 10, 2020: https://krebsonsecurity.com/tag/equifax-breach/.


XKCD
xkcd.com
I'M WORRIED ABOUT HUMANS DEVELOPING RESISTANCE TO US.
USING PASTA.
THE HUMAN IMMUNE SYSTEM IS A NIGHTMARE.
IT'S THE WORST.
IT'S THE SCARIEST THING IN THE UNIVERSE.
WHO WANTS A HUUGGG
NOOO!
WE CAN ONLY SURVIVE BY STAYING AHEAD OF IT. KEEP JUMPING FROM PERSON TO PERSON, KEEP MUTATING AND EVOLVING.
BUT NOW HUMANS ARE ADAPTING TOO FAST.
WE SPREAD THROUGH THEIR WATER. THEY BUILT PIPES.
WE USED MOSQUITOES. THEY PUT OUT NETS AND POISON EVERYWHERE.
WE SPREAD THROUGH SEX, AND SUDDENLY THEY ALL HAD THESE PLASTIC THINGS.
THIS TIME, WE REALLY THOUGHT WE HAD THEM.
ONE OF US GOT GOOD AT TRANSMISSION THROUGH EVERYDAY CONTACT.
IT WAS GREAT. WE WERE TEARING THROUGH LUNGS, SPREADING LIKE WILDFIRE.
HOORAY!
I HATE LUNGS.
THEN, ALL OF A SUDDEN, HUMANS EVERYWHERE JUST... STOPPED. THEY STOPPED WORKING, THEY STOPPED SEEING FRIENDS.
WHAT ARE THEY DOING?
NOTHING!
THEY'RE JUST SITTING THERE IN THEIR HOUSES WASHING THEIR HANDS.
SUDDENLY, HUMANS HAVE BECOME DEAD ENDS. WE TRY TO JUMP FROM ONE TO THE NEXT, BUT THERE'S NO ONE TO JUMP TO.
HELP
WE CAN'T ESCAPE.
WE'RE TRAPPED IN THERE WITH THOSE GHASTLY IMMUNE SYSTEMS.
ITS HUUG TIIIME
COME HERE FOR A HUUUG
HUUUGS
EVEN IF WE WIN A FIGHT, THERE'S NOWHERE TO GO.
HUUUUUUGGSS
HUUUGS
BY STAYING INSIDE, HUMANS HAVE BECOME RESISTANT.
HOW COULD THEY EVOLVE THAT FAST? HUMANS TAKE DECADES TO REPRODUCE!
IT'S NOT EVOLUTION. IT'S SOMETHING WITH THEIR BRAINS.
I WONDERED WHAT THOSE WERE FOR!
HUMANS STARTED LOOKING AT THEIR PHONES, TALKING, WRITING WORDS, AND MAKING SIGNS. A HUMAN NAMED "GLORIA GAYNOR" FILMED HERSELF SINGING AT HER BATHROOM SINK.
AND THEY BOUGHT LOTS OF PASTA.
THEN, AROUND THE WORLD, THEY ALL WENT HOME AND STARTED WASHING THEIR HANDS.
THEY SAW WHAT WE WERE DOING AND CHANGED THEIR BEHAVIOR TO STOP US.
BRAINS ARE THE WORST.
IT'S NOT OVER, RIGHT? THEY CAN'T SUSTAIN THIS. THEY MUST BE BORED AND TIRED.
WILL THEY GIVE UP?
I DON'T KNOW. THEY SEEM DETERMINED TO PROTECT EACH OTHER.
AND
THEY HAVE A LOT OF PASTA.

FreeBSD JOURNAL

# The FreeBSD Journal is now Free!

## Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

**DON'T MISS A SINGLE ISSUE!**

Sept/Oct 2018
FreeBSD JOURNAL
networking
TCP Stack Validation at Netflix
Internetworking in FreeBSD
A Quick Tour of SDN Using Mininet
FreeBSD and Processor TRENDS

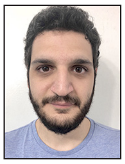
## 2020 Editorial Calendar:

- FreeBSD in Research (Jan/Feb 2020)
- Filesystems (March/April 2020)
- Network Performance (May/June 2020)
- Benchmarking/Tuning (July/Aug 2020)
- Contributing/Onboarding (Sept/Oct 2020)
- Workflows/CI (Nov/Dec 2020)

**Find out more at:** freebsdfoundation.org/journal

# A Study of SSD Reliability in Large Scale Enterprise Storage Deployments

STATHIS MANEAS, KAVEH MAHDAVIANI, TIM EMAMI, AND BIANCA SCHROEDER



Stathis Maneas is a PhD candidate in the Department of Computer Science at the University of Toronto. Prior to that, he obtained his MSc and BSc degrees at the National and Kapodistrian University of Athens. His main research interests include the design and implementation of computer systems, especially storage and file systems, and distributed systems. His current research focuses on the reliability aspect of systems. smaneas@cs.toronto.edu

Kaveh Mahdaviani is a Postdoctoral Fellow in the Department of Computer Science at the University of Toronto. Before that, he completed his PhD, MSc, and BSc programs at the University of Toronto, University of Alberta, and Isfahan University of Technology, respectively. His research focuses on distributed systems, information theory, and coding theory. mahdaviani@cs.toronto.edu

Tim Emami is a Senior Technical Director and the Storage Media subject matter expert at NetApp. During his over 13-year tenure, he helped pioneer ONTAP's transition from disk to flash. Prior to NetApp, he worked in R&D roles at Maxtor, Quantum, WDC, and StorCard. He studied control systems at California Polytechnic State Institute, San Luis Obispo. Tim.Emami@netapp.com

We present the first large-scale field study of NAND-based SSDs deployed in enterprise storage systems. Our study is based on field data, collected over 2.5 years, for a sample of almost 1.4 million drives from the total SSD population of a major enterprise storage vendor (NetApp). The data allows us to study a large number of factors that were not included in prior work, such as the effect of firmware versions, the reliability of TLC NAND, and correlations between drives within a RAID group. Our analysis provides insight into flash reliability, along with a number of practical implications.

System reliability is arguably one of the most important aspects of a storage system, and, as such, a large body of work exists on the topic of storage device reliability. Much of the older work is focused on hard disk drives (HDDs) [1, 5–7], but as more data is being stored on solid state drives (SSDs), the focus has recently shifted to the reliability of SSDs. In addition to a large amount of work on SSDs in lab conditions under controlled experiments, the first field studies reporting on SSD reliability in deployed production systems have recently appeared [3, 4, 8, 10]. These studies are based on data collected at datacenters at Facebook, Microsoft, Google, and Alibaba, where drives are deployed as part of large distributed storage systems. However, we observe that there still are a number of critical gaps in the existing literature that this work is striving to bridge:

- There were no studies that focus on *enterprise storage systems*. The drives, workloads, and reliability mechanisms in these systems can differ significantly from those in cloud datacenters. For example, the drives used in enterprise storage systems include high-end drives, and reliability is ensured through (single, double, or triple parity) RAID, instead of replication or distributed storage codes.
- We also observe that existing studies do not cover some of the most important characteristics of failures that are required for building realistic failure models, in order to compute metrics such as the mean time to data loss. This includes, for example, a breakdown of the reasons for drive replacements, including the scope of the underlying problem and the corresponding repair action (RAID reconstruction versus draining the drive), and most importantly, an understanding of the correlations between drive replacements in the same RAID group.

In this article, we present some selected findings of our work. For detailed results, please see our USENIX FAST '20 paper [2].

## Reasons for Replacements

SSD replacement can be triggered for various reasons, and different subsystems in the storage hierarchy can detect issues that trigger the replacement of drives. For example, issues might be reported by the drive itself, the storage layer, or the file system. Table 1 describes the different *reason types* that can trigger a drive replacement, along with their frequency, the recovery action taken by the system, and the scope of the problem. We group the different reason types behind SSD replacements into four *categories*, labeled A to D, based on their severity.

AND STORAGE

Bianca Schroeder is an Associate Professor and Canada Research Chair in the Computer Science Department at the University of Toronto. She completed her PhD and a post-doc at Carnegie Mellon University under the guidance of Mor Harchol-Balter and Garth Gibson, respectively. She is an Alfred P. Sloan Research Fellow and the recipient of the Outstanding Young Canadian Computer Science Prize of the Canadian Association for Computer Science, an Ontario Early Researcher Award, five best paper awards and a Test of Time award. bianca@cs.toronto.edu

| Category | Type | Pct. | Annual Repl. Rate (%) | Recovery Action | Scope |
|---|---|---|---|---|---|
| A | SCSI Error | 32.78 | 0.055 | RAID Reconstruction | Full |
|  | Unresponsive Drive | 0.60 | 0.001 |  |  |
| B | Lost Writes | 13.54 | 0.023 | RAID Reconstruction | Partial |
| C | Aborted Commands | 13.56 | 0.023 | RAID Reconstruction | Partial |
|  | Disk Ownership I/O Errors | 3.27 | 0.005 |  |  |
|  | Command Timeouts | 1.81 | 0.003 |  |  |
| D | Predictive Failures | 12.78 | 0.021 | Disk Copy | Zero |
|  | Threshold Exceeded | 12.73 | 0.020 |  |  |
|  | Recommended Failures | 8.93 | 0.015 |  |  |

**Table 1:** Description of reason types that can trigger a drive replacement. Disk copy operations are performed only where possible (e.g., a spare disk must be available).

The most benign category is category D, which relates to replacements that were triggered by logic either inside the drive or at higher levels in the system, which predicts future drive failure, based on, for example, previous errors, timeouts, and a drive's SMART statistics [9]. The most severe category is category A, which comprises those situations where drives become completely unresponsive, or where the SCSI layer detects a hardware error (reported by the drive) severe enough to trigger immediate replacement and RAID reconstruction of its data.

Category B refers to drive replacements that are taking place when the system suspects the drive to have *lost a write*, e.g., because it did not perform the write at all, wrote it to a wrong location, or otherwise corrupted the write. The root cause could be a firmware bug in the drive, although other layers in the storage stack could be responsible as well. As there are many potential causes, a heuristic is used to decide whether to trigger a replacement or not.

Finally, in category C most of the reasons for replacements are related to commands that were aborted or timed out. For instance, a command can be aborted when the host has sent some write commands to the device, but the actual data never reached the device due to connection issues. Ownership errors are related to the subsystem that keeps track of which node owns a drive; if an error occurs during the communication with this subsystem, the drive is marked as *failed*.

When examining the frequency of each individual type, we observe that SCSI errors are the most common type, responsible for ~33% of all replacements and, unfortunately, also one of the most severe reason types. On the other hand, drives rarely become completely unresponsive (0.60% of all replacements). Fortunately, one-third of all drive replacements are merely preventative (category D), using predictions of future drive failures, and are hence unlikely to have severe impact on system reliability. Finally, the two remaining categories (B and C) are roughly equally common, and both have the potential of partial data loss if RAID reconstruction of the affected data should turn out unsuccessful.

**Finding 1:** *One-third of replacements are associated with one of the most severe reason types (i.e., SCSI errors); on the other hand, one-third of drive replacements are merely preventative, based on predictions.*

## Factors Impacting Replacement Rates

We evaluate how different factors impact the replacement rates of the SSDs in our data set. We make use of the annual replacement rate (ARR) metric, which is commonly used to report failure frequency [4, 5, 7] and is defined as follows:

$$ARR = \frac{\text{Total failed devices}}{\text{Total device years}} \; in\,\%$$

### *Usage and Age*

It is well known that usage, and the wear-out of flash cells that comes with it, affects the reliability of flash-based SSDs; drives are guaranteed to remain functional for only a certain number of program/erase (PE) cycles. In our data set, SLC drives have a PE cycles limit of 100K, whereas the limit of most cMLC, eMLC, and 3D-TLC drives is equal to 10K cycles, with the exception of a few eMLC drive families with a 30K PE cycles limit.

Each drive reports the number of PE cycles it has experienced as a percentage of its PE cycle limit (denoted as *rated life used*), allowing us to study how usage affects replacement rates. Unfortunately, the rated life used is only reported as a truncated integer, and a significant fraction of drives report a zero for this metric, indicating less than 1% of their rated life has been used. Therefore, our first step is a comparison of the ARR of drives that report less than 1% versus more than 1% of their rated life used. The results for eMLC and 3D-TLC drives are shown in Figure 1, which includes both overall replacement rates ("All") and rates broken down by their replacement category (A to D). Throughout the article, error bars refer to 95th percentile confidence intervals; we also exclude two outlier models, i.e., I-C and II-C, with unusually high replacement rates to not obscure trends except for graphs involving individual drive families.

Figure 1 provides evidence for effects of infant mortality. For example, eMLC drives, the drives with less than 1% rated life used, are more likely (1.25x) to be replaced than those with more than 1% of rated life used. When further breaking results down by reason category, we find that drives with less usage consistently experience higher replacement rates for all categories.

Making conclusive claims for the 3D-TLC drives is harder due to limited data on drives above 1% of rated life used, resulting in wide confidence intervals. However, where we have enough data, observations are similar to those for eMLC drives, e.g., we see a significant drop in lost writes for drives above 1% of rated life used.

We also look at replacement rates as a function of a drive's age measured by its total months in the field. Figure 2 shows the conditional probability of a drive being replaced in a given month of its life, i.e., the probability that the drive will fail in month $x$ given that it has survived up to the end of month $x$-1.

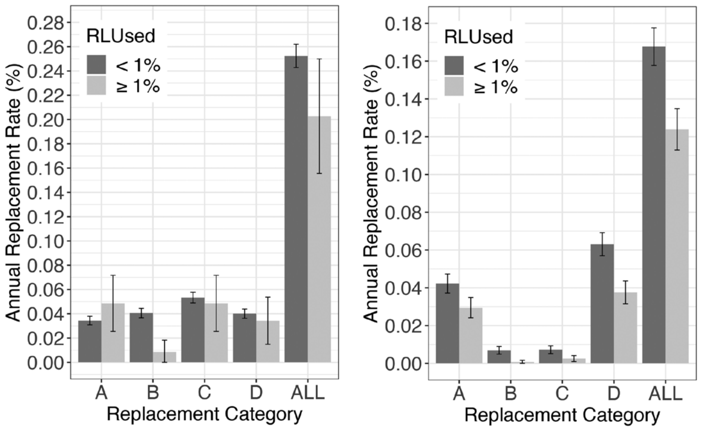


**Figure 1:** Annual replacement rate per flash type based on the drives' "rated-life-used" percentage

We observe an unexpectedly long period of infant mortality with a shape that differs from the common "bathtub" model, often used in reliability theory. The bathtub model assumes a short initial period of high failure rates, which then quickly drops. Instead, we observe for both 3D-TLC and eMLC drives, a long period (12–15 months) of increasing failure rates, followed by a lengthy period (another 6–12 months) of slowly decreasing failure rates, before rates finally stabilize. This brings up the question of what could be done to reduce these effects. One might consider, for example, an extended, more intense burn-in period before deployment, where drives are subjected to longer periods of high read and write loads. Given the low consumption of PE cycles that drives see in the field (99% of drives do not even use up 1% of their PE cycle limit), there seems to be room to sacrifice some PE cycles in the burn-in process.

Finally, it might be surprising that we do not observe an increase in ARR for drives towards the end of their life, but the majority of drives, even those deployed for several years, do not experience a large number of PE cycles.

**Finding 2:** *We observe a very drawn-out period of infant mortality, which can last more than a year, and see failure rates 2–3x larger than later in life.*

### *Flash and Drive Type*

The drive models in our study differ in the type of flash they are based on, i.e., in how many bits are encoded in a single flash cell. For instance, Single-Level Cell (SLC) drives encode only one bit per cell, while Multi-Level Cell (MLC) drives encode two bits in one cell for higher data density and thus a lower total cost, but potentially higher propensity to errors. The most recent generation of flash in our data set is based on Triple-Level Cell (3D-TLC) flash with three bits per cell.

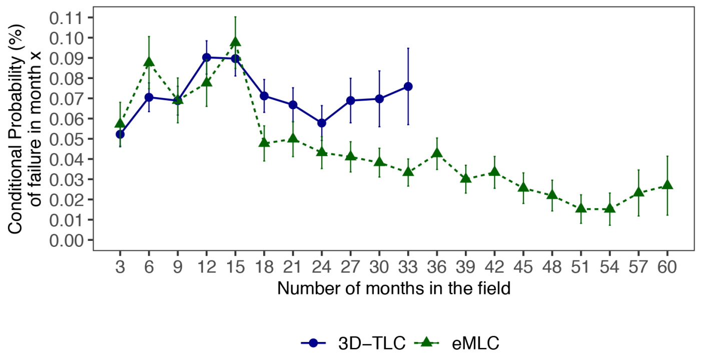


**Figure 2:** Conditional probability of failure based on a drive's age (number of months in the field) for all drive families

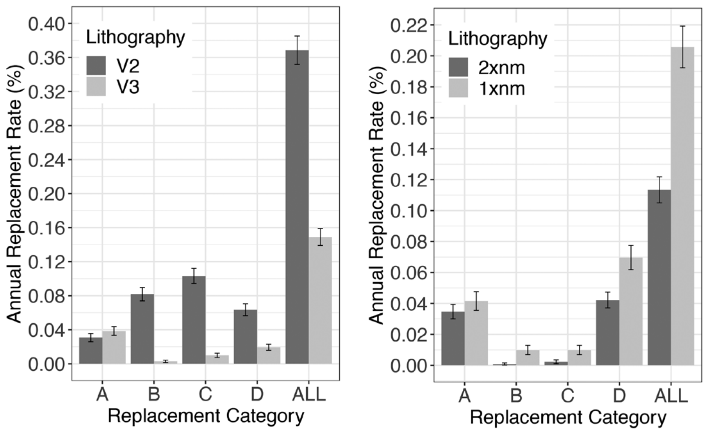


**Figure 3:** Annual replacement rate per flash type and lithography broken down by replacement category. The 1x nm and 2x nm notations denote any lithography in the range of 10–19 nm and 20–29 nm, respectively.

We turn to Figures 1 and 3 to compare 3D-TLC and eMLC drives, which take usage and lithography into account. Figure 1 indicates that ARRs for 3D-TLC drives are around 1.5x higher than for eMLC drives, when comparing similar levels of usage. Figure 3 paints a more complex picture. While V2 3D-TLC drives have a significantly higher replacement rate than any of the other groups, the V3 3D-TLC drives are actually comparable to 2x nm eMLC drives, and in fact have lower ARR than the 1x nm eMLC drives. So lithography might play a larger role than flash type alone; we take a closer look at lithography below.

When we compare the results for the MLC drives in our data set against previous work, we observe that Narayanan et al. [4] report replacement rates between 0.5–1% for their *consumer* class MLC drives, with the exception of a single enterprise class model, whose replacement rate is equal to 0.1%; however, the authors in [4] consider only fail-stop failures. In our study, we consider different types of failures, and, thus, the reported replacement rates would have been even smaller had we considered only fail-stop failures.

**Finding 3:** *Overall, the highest replacement rates in our study are associated with 3D-TLC drives. However, no single flash type has noticeably higher replacement rates than the other flash types in this work, indicating that other factors (e.g., lithography) can have a bigger impact on reliability.*

## *Lithography*

Lithography has been shown to be highly correlated with a drive's raw bit error rate (RBER); models with smaller lithography report higher RBERs according to a study based on datacenter drives [8], but not necessarily higher replacement rates. We explore what these trends look like for the drives in enterprise storage systems. To separate the effect of lithography from flash type, we perform the analysis separately for each flash type.

The bar graph in Figure 3 (right) shows the ARR for eMLC drives separated into 2x nm and 1x nm lithographies broken down by failure category, also including one bar for replacements of all categories. The 1x nm and 2x nm notations denote any lithography in the range of 10–19 nm and 20–29 nm, respectively. We observe that the higher density 1x nm drives experience almost twice the replacement rate of 2x nm drives. Also, replacement rates for each of the individual reason types are higher for 1x nm drives than for 2x nm, with the only exception of reason category A, which corresponds to unresponsive drives.

In contrast to eMLC drives, the 3D-TLC drives see higher replacement rates for the lower density V2 drives, which internally have fewer layers than V3. When breaking replacement rates down by failure reason, we observe that consistently with the results for TLC drives, the only reason code that is not affected by lithography is category A, which corresponds to unresponsive drives.

**Finding 4:** *In contrast to previous work, higher density drives do not always see higher replacement rates. In fact, we observe that, although higher density eMLC drives have higher replacement rates, this trend is reversed for 3D-TLC.*

## *Firmware Version*

Given that bugs in a drive's firmware can lead to drive errors or, in the worst case, to an unresponsive drive, we are interested to see whether different firmware versions are associated with a different ARR. Each drive model in our study experiences different firmware versions over time. We name the first firmware version of a model FV1, the next one FV2, and so on. An individual drive's firmware might be updated to a new version, but we observe that the majority of drives (70%) appear under the same firmware version in all data snapshots.

Figure 4 shows the ARR associated with different firmware versions for each drive family. Considering that firmware varies across drive families and manufacturers, it only makes sense to compare the ARR of different firmware versions within the same drive family. To avoid other confounding factors, in particular

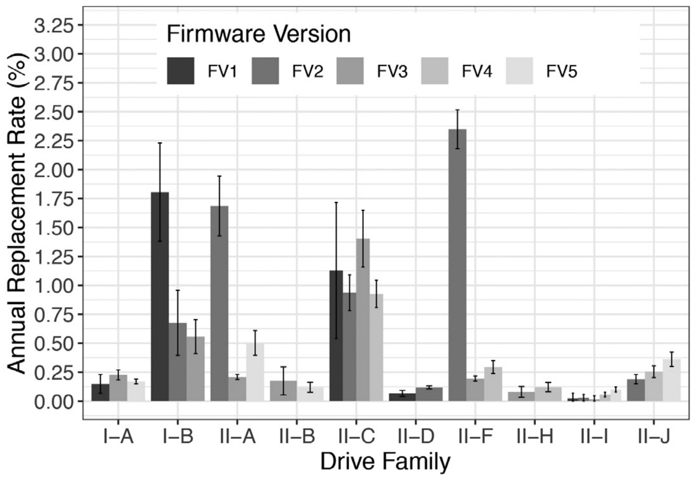


**Figure 4:** Effect of the firmware version on replacement rates broken down by drive family

age and usage, the graph in Figure 4 only includes drives with rated life used of less than 1% (the majority of drives).

We find that drives' firmware version can have a tremendous impact on reliability. In particular, the earliest versions can have an order of magnitude higher ARR than later versions. This effect is most notable for families I-B (more than 2x decrease in ARR from FV1 to FV2), II-A (8x decrease from FV2 to FV3), and II-F (more than 10x decrease from FV2 to FV3). Finally, we note that this effect persists even if we only include drives whose firmware has never changed in our data snapshots.

**Finding 5:** *Earlier firmware versions can be correlated with significantly higher replacement rates, emphasizing the importance of firmware updates.*

## Correlations between Drive Failures

A key question when deriving reliability estimates—e.g., for different RAID configurations—is how failures of drives within the same RAID group are correlated.

For a detailed understanding of correlations, we consider all RAID groups that have experienced more than one drive replacement over the course of our observation period, and plot in Figure 5 the time between consecutive drive replacements within the same RAID group. We observe that very commonly, the second drive replacement follows the preceding one within a short time interval. For example, 46% of consecutive replacements take place at most one day after the previous replacement, while 52% of all consecutive replacements take place within a week of the previous replacement.

Another important question in RAID reliability modeling is how the chance of multiple failures grows as the number of drives in the RAID group increases. Figure 6 (left) presents, for the most common RAID group sizes, the percentage of RAID groups of

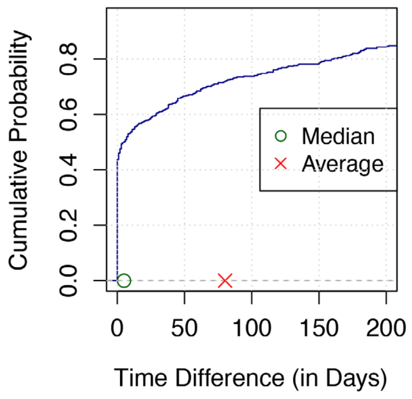


**Figure 5:** Time difference between successive replacements within RAID groups

that size that experienced at least one drive replacement. As one would expect, larger RAID groups have a higher chance of experiencing a drive replacement; yet, the effect of a RAID group's size on the replacement rates saturates for RAID groups comprising more than 18 drives.

Concerning multiple failures within the same RAID group, we make an interesting observation in Figure 6 (middle). When we look at the percentage of RAID groups that have experienced at least two drive replacements (potential double failure), this does not seem to be clearly correlated with RAID group size. In other words, the largest RAID group sizes do not necessarily seem to have a higher rate of double (or multiple) failures compared to smaller RAID groups.

This observation is confirmed when we look at the conditional probability that a RAID group will experience more replacements, *given that it has already experienced another replacement*, in Figure 6 (right). More precisely, for each RAID group size, we consider those RAID groups that had at least one drive replacement and compute what percentage of them had at least one more replacement within a week. Interestingly, we observe there is no clear trend that larger RAID group sizes have a larger chance of one drive replacement being followed by more replacements. Note that, as already mentioned, the chance of experiencing a drive failure grows with the size of the RAID group (Figure 6 left); however, the chance of *correlated failures* does not show a direct relationship with the group's size.

**Finding 6:** *While large RAID groups have a larger number of drive replacements, we find no evidence that the rate of multiple failures per group (which is what can create potential for data loss) is correlated with RAID group size. The reason seems to be that the likelihood of a follow-up failure after a first failure is not correlated with RAID group size.*

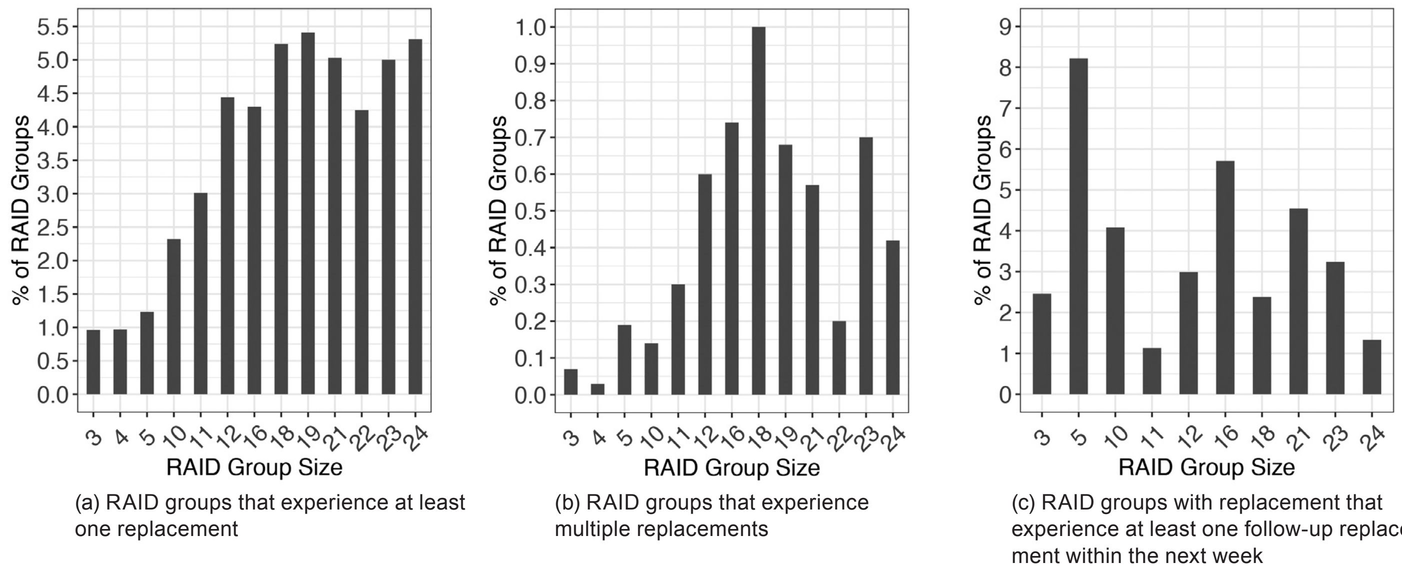


**Figure 6:** Statistics on replacements within RAID groups

## Conclusion

Previous work has focused on the reliability characteristics of SSDs deployed in distributed datacenter storage systems. Our work presents the first large-scale field study of NAND-based SSDs in *enterprise storage systems* [2]. Below, we summarize some of the most important findings and implications of our work:

- Our observations emphasize the importance of firmware updates, as earlier firmware versions can be correlated with significantly higher failure rates. Yet 70% of drives remain at the same firmware version throughout the length of our study. Consequently, we encourage enterprise storage vendors to make firmware upgrades as easy and painless as possible so that customers apply the upgrades without worries about stability issues.
- We observe significant correlations between failures within RAID groups. This emphasizes the importance of incorporating correlated failures into any analytical models in order to arrive at realistic estimates of the probability of data loss. It also makes a case for more than just single-parity RAID.
- The failure rates in our study do not resemble the "bathtub" shape assumed by classical reliability models. Instead, we observe no signs of increased failure rates at end of life and also a very drawn-out period of infant mortality, which can last for more than a year and see failure rates 2–3x larger than later in life.
- There has been a fear that the limited PE cycles of NAND SSDs can create a threat to data reliability in the later part of a RAID system's life due to correlated wear-out failures, as the drives in a RAID group age at the same rate. Instead, we observe that correlated failures due to infant mortality are likely to be a bigger threat.

***Acknowledgments***

We would like to acknowledge several people at NetApp for their contributions to this work: Rodney Dekoning, Saumyabrata Bandyopadhyay, and Anita Jindal for their early support and encouragement, and Aziz Htite, who helped cross-validate our data and assumptions along the way. We would also like to thank the internal reviewers within the ATG, ONTAP WAFL, and RAID groups for their careful and detailed feedback. A very special thank you to Biren Fondekar's Active IQ team in Bangalore: Asha Gangolli, Kavitha Degavinti, and finally Vinay N., who spent countless late nights on the phone with us as we cleaned and curated this work's foundational data sets. We also thank our USENIX FAST reviewers and our shepherd, Devesh Tiwari, for their detailed feedback and valuable suggestions.

***References***

[1] D. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An Analysis of Latent Sector Errors in Disk Drives," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pp. 289–300.

[2] S. Maneas, K. Mahdaviani, T. Emami, and B. Schroeder, "A Study of SSD Reliability in Large Scale Enterprise Storage Deployments," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pp. 137–149: https://www.usenix.org/system/files/fast20-maneas.pdf.

[3] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A Large-Scale Study of Flash Memory Failures in the Field," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, pp. 177–190.

[4] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid, "SSD Failures in Datacenters: What? When? And Why?" in *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, pp. 7:1–7:11.

[5] E. Pinheiro, W.-D. Weber, and L. André Barroso, "Failure Trends in a Large Disk Drive Population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pp. 17–23: https://www.usenix.org/legacy/event/fast07/tech/full_papers/pinheiro/pinheiro.pdf.

[6] B. Schroeder, S. Damouras, and P. Gill, "Understanding Latent Sector Errors and How to Protect Against Them," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3 (September 2010), pp. 9:1–9:23.

[7] B. Schroeder and G. A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pp. 1–16: https://www.usenix.org/legacy/event/fast07/tech/schroeder/schroeder.pdf.

[8] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash Reliability in Production: The Expected and the Unexpected," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 67–80: https://www.usenix.org/system/files/conference/fast16/fast16-papers-schroeder.pdf.

[9] Wikipedia, "S.M.A.R.T.": https://en.wikipedia.org/wiki/S.M.A.R.T. Accessed: 3/2/20.

[10] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, "Lessons and Actions: What We Learned from 10k SSD-Related Storage System Failures," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pp. 961–976: https://www.usenix.org/system/files/atc19-xu_0.pdf.

# SkyhookDM: Data Processing in Ceph with Programmable Storage

JEFF LEFEVRE AND CARLOS MALTZAHN

Jeff LeFevre is an Assistant Adjunct Professor for Computer Science and Engineering at UC Santa Cruz. He currently leads the SkyhookDM project, and his research interests are in cloud databases, database physical design, and storage systems. Dr. LeFevre joined the CSE faculty in 2018 and has previously worked on the Vertica database for HP. jlefevre@ucsc.edu

Carlos Maltzahn is an Adjunct Professor for Computer Science and Engineering at UC Santa Cruz. He is the founder and director of Center for Research in Open Source Software (cross.ucsc.edu) and a co-founder of the Systems Research Lab, known for its cutting-edge work on programmable storage systems, big data storage and processing, scalable data management, distributed system performance management, and practical replicable evaluation of computer systems. In 2005 he co-founded and became a key mentor on Sage Weil's Ceph project. Dr. Maltzahn joined the CSE faculty in 2008, has graduated nine PhD students since, and has previously worked on storage for NetApp. carlosm@ucsc.edu

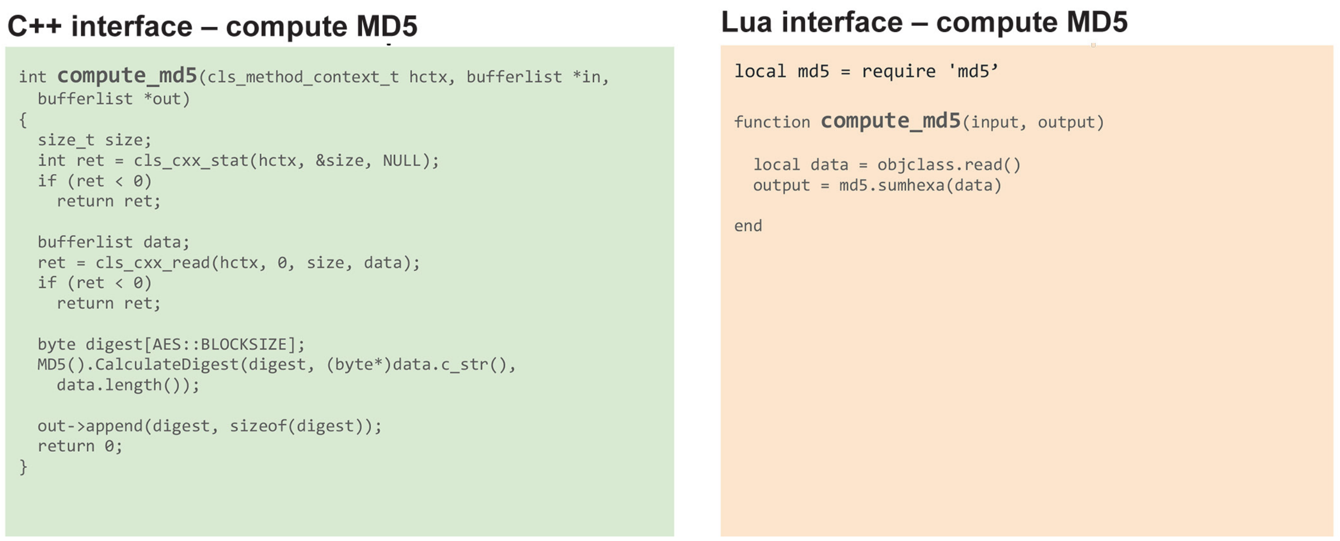
With ever larger data sets and cloud-based storage systems, it becomes increasingly more attractive to move computation to data, a common principle in big data systems. Historically, data management systems have pushed computation nearest to the data in order to reduce data moving through query execution pipelines. Computational storage approaches address the problem of both data reduction nearest the source as well as offloading some processing to the storage layer.

As storage systems become more disaggregated from client applications, such as distributed object storage (e.g., S3, Swift, and Ceph), there is new interest in computational storage disaggregated over networks [7]. There has also been a long line of efforts toward computational storage, including custom hardware and software for intelligent disks and active storage [5, 6, 13, 15], commercial appliances, and middleware approaches in the cloud [1, 2].

Recent research on *programmable storage systems* [4, 9–12, 14, 16] takes the approach of exposing, augmenting, or combining existing functionality already present in the storage system to increase storage capabilities, performance, or provide new storage APIs and services to clients. There are several benefits to this approach, including (1) building upon a trusted, production quality storage system rather than starting from scratch or building a one-off solution; (2) requiring no additional system or hardware (e.g., Zookeeper or specialized disks) to be installed to provide these new functions; and (3) avoiding the need for each client to reimplement available functionality on a per-use-case basis by simply accessing newly available storage services as they become available.

The Skyhook Data Management project (skyhookdm.com) [8, 9] utilizes programmable storage methods to extend Ceph with data processing and management capabilities. Our methods are applied directly to objects or across groups of objects by the storage system itself. These capabilities are implemented as extensions to Ceph's through its existing `cls` mechanism. This mechanism allows users to install custom functions that can be applied to objects in addition to `read()` or `write()`. Our approach that includes custom extensions along with data partitioning and structured data storage enables storage servers to semantically interpret object data in order to execute functions such as SQL SELECT, PROJECT, and AGGREGATE. We also developed extensions for data management functions that perform physical design tasks such as indexing, data repartitioning, and formatting. Partitioning and formatting can be especially useful in the context of row versus column-oriented formats for workload processing.

The immediate benefits of this approach are I/O and CPU scalability (for certain functions) that grows or shrinks along with the number of storage servers. Since objects are semantically self-contained (i.e., a database partition) and are the entities that custom functions operate on, and since the storage system automatically rebalances objects across available servers—our approach, using I/O and compute elasticity, can directly benefit any storage client application that is able to take advantage of these methods.



C++ interface – compute MD5

```
int compute_md5(cls_method_context_t hctx, bufferlist *in,
  bufferlist *out)
{
  size_t size;
  int ret = cls_cxx_stat(hctx, &size, NULL);
  if (ret < 0)
    return ret;

  bufferlist data;
  ret = cls_cxx_read(hctx, 0, size, data);
  if (ret < 0)
    return ret;

  byte digest[AES::BLOCKSIZE];
  MD5().CalculateDigest(digest, (byte*)data.c_str(),
    data.length());

  out->append(digest, sizeof(digest));
  return 0;
}
```

Lua interface – compute MD5

```
local md5 = require 'md5'

function compute_md5(input, output)

  local data = objclass.read()
  output = md5.sumhexa(data)

end
```

**Figure 1:** Example Ceph custom object class method to compute MD5 sum

The key insight of SkyhookDM is to simplify data management and minimize data movement by enabling the storage system to semantically interpret, manage, and process data. This can dramatically reduce the complexity of coordination and resources required higher up the software stack at the application layer. For example, applying a filter, building an index, or reformatting data can happen in parallel remotely on individual objects. This is because the necessary context for many common data processing and management tasks resides with the data, which makes data movement to establish computational contexts entirely unnecessary. In our work, this context includes the data semantics and the processing functions that are included in our formatted data within objects and our shared library extensions available within the storage servers, respectively.

For example, a single node database such as PostgreSQL can push query operations into the storage layer through its external table interface (foreign data wrapper), which can invoke these functions on objects and, hence, distribute computation across many storage nodes. For file interfaces, a similar mechanism is available in the scientific file format HDF5 with its Virtual Object Layer (VOL) that enables HDF5 to be mapped to non-POSIX storage back ends. Hence, similar to partitioning a database table, a large file can be "extended" by HDF5 functions into smaller objects across many storage nodes. SkyhookDM also provides a Python client using the Pandas DataFrame abstraction. In these ways, our methods can be used to scale out a database or another data-intensive application designed to run only on a single node.

Our approach to extend storage with data management tasks has several significant benefits:

- Increased query performance when pushing down computation directly to objects across many storage servers.
- Reduced network I/O and host interconnect bandwidth for computations that result in data reduction (e.g., SELECT, PROJECT, AGGREGATE).
- Reduced CPU at clients due to offloading and reduced CPU both at clients and servers due to creating, sending, and receiving fewer packets for data reduction queries.
- Reduced application complexity and resources for data management tasks such as indexing, re-partitioning data, or converting between formats (e.g., row to column).
- Support for operating on multiple data formats in storage, and the capability to extend support for other formats.
- Fewer application-level storage system assumptions of (possibly out-of-date) "storage-friendly" access patterns and more intelligent storage systems adapting to new devices.

Next we provide a short background, our architecture and methodology, and a few experiments to evaluate performance, scalability, and overhead of our approach to in-storage processing.

## Background

Ceph is a widely used open source distributed object storage system created by Sage Weil at UC Santa Cruz and backed by Red Hat, IBM, and many other large corporations. It has no single point of failure, is self-healing, and scales to very large installations of 100's petabytes of data. It provides file, block, and object interfaces. New object methods can be created using cls extensions that are registered as READ and/or WRITE methods and then compiled into shared libraries within the Ceph source tree or via an SDK. These new shared libraries are installed on storage servers (also known as OSDs) in a directory well known to Ceph, `/usr/lib64/rados-classes/`.

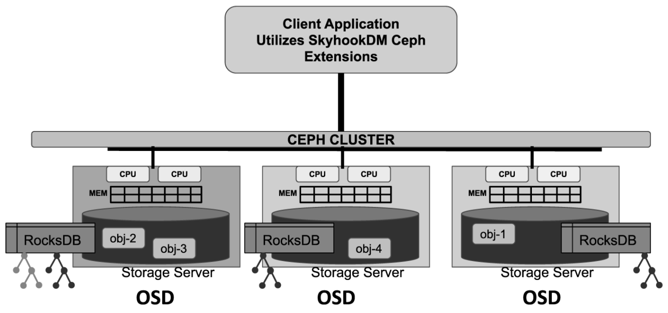


**Figure 2:** SkyhookDM architecture showing a three-node Ceph cluster with four objects

Ceph object classes currently support C++ and Lua interfaces, and an example implementation of an object method would be to compute a checksum, or perhaps generate a thumbnail image as part of a custom read or write method as shown in Figure 1. Critically, partial reads and writes of objects are also possible in Ceph, which is useful to reduce disk I/O for certain queries such as PROJECT a subset of all columns.

Apache Arrow and Google FlatBuffers are fast, in-memory serialization libraries. Arrow was developed for columnar processing and sharing data over the wire, and supports compression and interacts well with other formats, especially Parquet. FlatBuffers was developed for gaming, with a row-based abstraction; is very flexible, including a schema-free interface called FlexBuffers; and supports user-defined structs. They both offer a highest level abstraction of a table. We include these libraries within our shared library code to locally interpret and process object data.

## SkyhookDM Architecture

Figure 2 shows our architecture, with a client application connected to a standard Ceph cluster with SkyhookDM cls extensions installed. The client application connects to Ceph using the standard librados library which makes the extensions available to the client. In this way databases such as PostgreSQL can invoke these extensions via their foreign data wrapper.

Figure 2 depicts a Ceph cluster of three Object Storage Devices (OSDs), each with its own CPU and memory resources that are utilized by our extensions for data processing. Each OSD stores a collection of objects and also has a local RocksDB instance that we exploit as an indexing mechanism.

SkyhookDM extensions are present as a shared library on each OSD, and these extensions can be applied to any local object for customized local processing. During processing, results can be returned to the client in a different format than the internal storage format, e.g., Arrow table, or PostgreSQL binary format from an object with FlatBuffer data format. Since our shared libraries are present on each OSD, they are immediately available to objects stored on newly added OSDs—for instance, when adding nodes to a Ceph cluster.

```
table FB_Meta {
  blob_format      : int32;    // enum SkyFormatType of data stored in blob
  blob_data        : [ubyte];  // formatted data (any supported format)
  blob_size        : uint64;   // number of bytes in data blob
  blob_deleted     : bool;     // is deleted?
  blob_orig_off    : uint64=0; // optional: offset of blob data in orig file
  blob_orig_len    : uint64=0; // optional: num bytes in orig file
  blob_compression : int=0;    // optional: populated by enum {none, lzw, ...}
}
```

**Figure 3:** Per object metadata wrapper regarding the serialized data partition (blob data) stored within

## Methodology

Our work exploits Ceph's cls extension interface by first writing structured data to objects and then adding access methods implementing common SQL operations. We store structured data using popular and very efficient data serialization libraries such as Apache Arrow and Google FlatBuffers and use their APIs to implement new cls access methods. For greater flexibility to support multiple data formats, the structured data includes metadata about itself that expresses higher level information such as the data's current layout, whether or not it is compressed, and the data's length. Figure 3 shows this information, which is itself defined as a Flatbuffer wrapper. This helps to abstract away data layout information from the higher level client applications, creating flexibility to store and process data in various formats as well as reduce the need for client applications to keep track of data formats or compression on a per-object basis.

### *Physical and Logical Data Alignment*

Physical and logical data alignment can be crucial for good performance, including with big data processing frameworks such as MapReduce [3]. In our work, physical and logical alignment is required such that when partitions are stored in objects of structures data, a given object contains a complete logical subset of the original data in order to interpret the data's semantics and perform any meaningful processing. For example, a database table partition could be stored in an object, resulting in a collection of complete rows that can be operated upon. In contrast, byte-aligned physical partitions (e.g., 64 MB) can result in incomplete rows, with part of a row stored across two different objects. Any processing for such rows would need to be performed at a higher layer and first perform a collect or gather operation across perhaps multiple storage servers. This would render object-local processing useless and result in unnecessary network I/O.

### *Data Partitioning and Format*

In our work, we consider row and column-based partitions. Partitions are formatted using fast, in-memory data serialization formats: Google FlatBuffers for row partitions or Apache Arrow for column partitions. Both of these formats allow us to encode the data schema within, which allows the structured data to be interpreted by our cls methods.

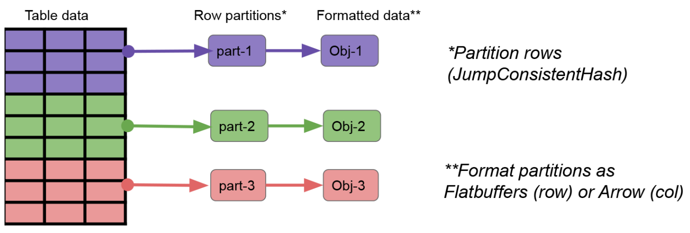


Table data
Row partitions*
Formatted data**
part-1
Obj-1
part-2
Obj-2
part-3
Obj-3
*Partition rows (JumpConsistentHash)
**Format partitions as Flatbuffers (row) or Arrow (col)

**Figure 4:** Data partitioning, formatting, and objects in SkyhookDM

For either row or column storage, data is partitioned, formatted, and named using conventions such as *table_name.partition_num*, resulting in a collection of objects per table where each object represents a logical data partition that is self-contained, with metadata indicating its semantics such as the table schema. Figure 4 depicts how data is partitioned and formatted for a database table. Data semantics are included within the format, which is then wrapped with our metadata wrapper, serialized as binary object data, and stored in Ceph. Data placement is handled by Ceph's pseudo-random data distribution function (CRUSH).

Rather than looking up all object names of a table, our object-naming convention includes content information, such as the table or column name. This results in constant-size metadata per table that includes only a name generator function and a few constants such as total number of objects. Further content-based information is possible to encode in the naming as well, such as row ranges for range-based row partitions (e.g., month). Such content-based object names and generator functions can then also be used for partition pruning during query plan generation.

This partitioning and formatting method achieves logical and physical alignment, embeds data semantics locally within each partition, and, along with the serialization format APIs and custom object classes, allows us to perform processing independently on each partition. Creating many objects (i.e., partitions) and scaling out the number of storage servers can enable a high degree of parallelism for data processing.

### *Data Layout and Physical Design*

Within an object, there are several options to consider for laying out the data, either as a set of byte stream extents in a chunk store, as a set of entries in a key/value store, or combinations of both. The key-value store is a local instance of RocksDB, used by Ceph for managing the local collection of objects on the OSD. Object methods can also access RocksDB via Ceph's `object_map` or `omap` interface. SkyhookDM uses `omap` to store both logical information (data content) and physical information (data offsets). For instance, the logical information includes the row number of a particular value within an objects formatted data (e.g., row *i*) to provide quick lookups for point or range queries. The physical information includes the offsets and lengths of the sequence of data structures within an object. Both indexing and data layout within an object is a consideration for physical design [4] management, such as potentially storing each column of a table as a separate data structure in order to minimize the amount of disk I/O required to retrieve a given column. This helps to improve the performance of PROJECT queries, for example.

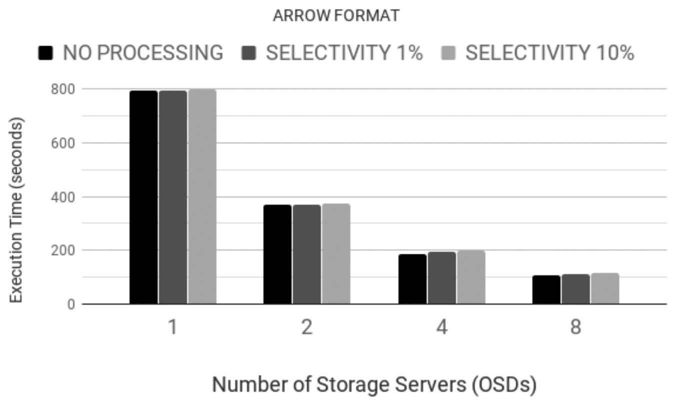


**Figure 5:** Query execution time as the number of storage servers is scaled out

## Evaluation

We executed all experiments on Cloudlab, an NSF bare-metal-as-a-service infrastructure. We used machine profile c220-g5 for all nodes, 2x Intel Xeon Silver 2.2 GHz, 192 GB RAM, 1 TB HDD, and 10 GbE. Our data set was the LINEITEM table from the standard TPC-H database benchmark, with one billion rows. We partition this table, format, and store into 10,000 objects of an equal number of rows. The objects are distributed by Ceph across all storage nodes. Data is formatted as FlatBuffer or Arrow as indicated.

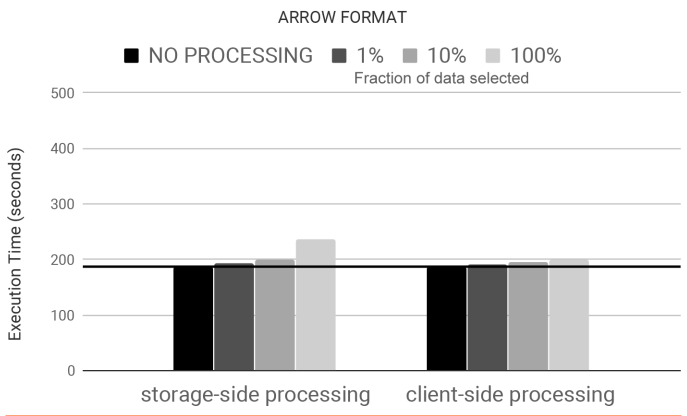


**Figure 6:** Query execution time for storage-side processing (offloading) versus processing on client machine with a four-node Ceph cluster

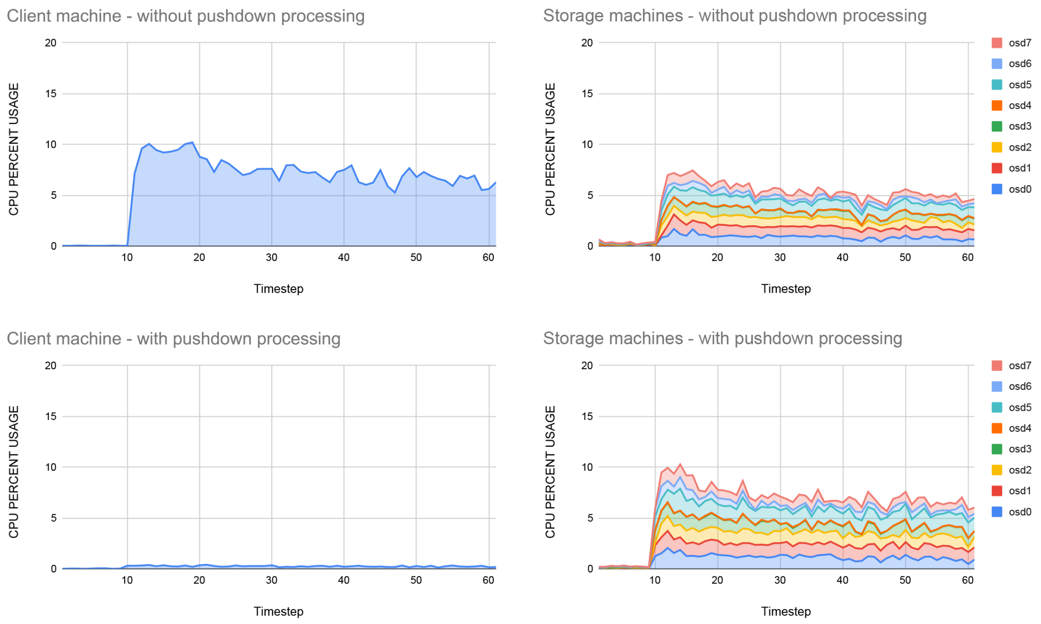


**Figure 7:** CPU usage during first 60 seconds for client machine (left) and eight storage servers (right, stacked) for 1% selectivity file query *without* pushdown processing (top) and *with* pushdown (bottom)

### *Performance Improvement with Pushdown Query Processing*

Figure 5 shows query latency is reduced as the number of storage servers increases. The no-processing bar is a standard read of all the data, representing I/O scale out. The other bars show offloading of a select query for 1% and 10% of data rows. This represents I/O and CPU scale out. The offloading result tracks very closely to the I/O scale out, with little extra overhead for the storage servers to read and process versus only read. This highlights how the computation is distributed across all storage servers. While execution time is not reduced in this case, likely due to many cores and very fast network, the overhead to apply computation in Ceph is low, and CPU usage on the client is dramatically reduced, as we show next.

### *Overhead of Data Processing Libraries in Ceph*

Figure 6 shows query execution time when processing data with all storage machines or on the single client machine. We first execute a standard read (no processing) as a baseline. Then we execute a query that selects 1%, 10%, or 100% of rows. In both cases there is little overhead to apply the data processing in storage except the case when selecting all data. This is due to the extra cost to both filter and then repackage and return each object when all rows pass the filter. This indicates the need for a statistics-based query optimizer to make wise decisions about what computations to offload.

### *CPU Usage with and without Offloading*

Figure 7 shows that without offloading (no pushdown processing), the client spends over 5% of CPU usage to receive packets and apply SELECT (top left). With offloading the client CPU, usage is near zero (bottom left). This is because the client receives only 1% of the original data packets and does not have to apply SELECT. The processing work is shifted to the storage servers (bottom right), showing a small corresponding increase in the stacked total CPU usage that is distributed across all storage servers (bottom right). However, now the work done by storage servers is actually useful for data processing, whereas the work done by storage servers in the without pushdown case (top right) is simply creating and sending packets containing 99% unnecessary data.

## Conclusion

SkyhookDM extends Ceph with object classes and fast serialization libraries to offload computation and data management tasks to storage. We have shown our approach has minimal overhead and scales with the number of storage servers. Our design is highly flexible, utilizing row or column-oriented data formats as well as the ability to dynamically convert between them directly in storage, eliminating the need to bring data into the client for processing or data management tasks. Extending our programmable storage approach to support custom data formats and more complex processing is another goal, and we are currently working on extensions for high energy physics data that uses the ROOT file format.

## Acknowledgments

Support provided by the Center for Research in Open Source Software at UC Santa Cruz (cross.ucsc.edu), NSF Grant OAC-1836650, CNS-1764102, CNS-1705021. We would also like to thank everyone who has contributed to the open source SkyhookDM project.

## *References*

[1] Amazon Redshift Spectrum documentation: https://docs.aws.amazon.com/redshift/latest/dg/c-using-spectrum.html.

[2] Swift storelet engine overview documentation: https://docs.openstack.org/storlets/latest/storlet_engine_overview.html.

[3] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-Based Query Processing in Hadoop," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC11)*, November 12–18, 2011, Seattle, WA.

[4] K. Dahlgren, J. LeFevre, A. Shirwadkar, K. Iizawa, A. Montana, P. Alvaro, and C. Maltzahn, "Towards Physical Design Management in Storage Systems," in *Proceedings of the 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, November 18, 2019, Denver, CO.

[5] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A Case for Intelligent Disks (IDISKs)," *ACM SIGMOD Record*, vol. 27, no. 3 (September 1998), pp. 42–52.

[6] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon, "In-Storage Processing of Database Scans and Joins," *Information Sciences*, vol. 327 (January 10, 2016), pp. 183–200.

[7] P. Kufeldt, C. Maltzahn, T. Feldman, C. Green, G. Mackey, and S. Tanaka, "Eusocial Storage Devices: Offloading Data Management to Storage Devices that Can Act Collectively," *;login:*, vol. 43, no. 2 (Summer 2018), pp. 16–22.

[8] J. LeFevre and C. Maltzahn, "Scaling Databases and File APIs with Programmable Ceph Object Storage," 2020 Linux Storage and Filesystems Conference (Vault '20), February 24–25, 2020, Santa Clara, CA.

[9] J. LeFevre and N. Watkins, "Skyhook: Programmable Storage for Databases," 2019 Linux Storage and Filesystems Conference (Vault '19), February 25–26, 2019, Boston, MA.

[10] M. A. Sevilla, I. Jimenez, N. Watkins, J. LeFevre, P. Alvaro, S. Finkelstein, P. Donnelly, and C. Maltzahn, "Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace," in *Proceedings of 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*, May 21–25, 2018, Vancouver, BC, Canada.

[11] M. A. Sevilla, R. Nasirigerdeh, C. Maltzahn, J. LeFevre, N. Watkins, P. Alvaro, M. Lawson, J. Lofstead, and J. Pivarski, "Tintenfisch: File System Namespace Schemas and Generators," 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '18), July 9–10, 2018, Boston, MA.

[12] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A Programmable Storage System," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*, April 23–26, 2017, Belgrade, Serbia.

[13] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Database-Aware Semantically-Smart Storage," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, December 13–16, 2005, San Francisco, CA.

[14] N. Watkins, M. A. Sevilla, I. Jimenez, K. Dahlgren, P. Alvaro, S. Finkelstein, and C. Maltzahn, "Declstore: Layering Is for the Faint of Heart," 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17), July 10–11, 2017, Santa Clara, CA.

[15] L. Woods, Z. István, and G. Alonso, "Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading," in *Proceedings of the VLDB Endowment*, vol. 7, no. 11 (July 2014), pp. 963–974.

[16] M. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. Brandt, S. Weil, G.Farnum, and S. Fineberg, "Mantle: A Programmable Metadata Load Balancer for the Ceph File System," in *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC15)*, November 15–20, 2015, Austin, TX.

# Interview with Natalie Silvanovich

RIK FARROW

Natalie Silvanovich is a security researcher on Google Project Zero. Her current focus is browser security, including script engines, WebAssembly, and WebRTC. Previously, she worked in mobile security on the Android Security Team at Google and as a team lead of the Security Research Group at BlackBerry, where her work included finding security issues in mobile software and improving the security of mobile platforms. Outside of work, Natalie enjoys applying her hacking and reverse-engineering skills to unusual targets and has spoken at several conferences on the subject of Tamagotchi hacking. natalie@natashenka.ca

Rik is the editor of *;login:*. rik@usenix.org

I met Natalie Silvanovich at the luncheon during USENIX Security '19 in Santa Clara. We had a fun discussion, and I resolved to spend some time following up later.

*Rik Farrow:* I am familiar with a really "old" way of finding bugs: fuzzing. I know this was very common in the late '90s, and I assume you were using fuzzing sometimes when you worked at BlackBerry. What's different about how you search for bugs today?

*Natalie Silvanovich:* It's been nearly 15 years since I started doing vulnerability research, and in some ways the fundamental techniques for finding security bugs haven't changed much. Fuzzing and code review (or binary analysis for software that doesn't have source code available) are still the techniques I use to find the majority of bugs I report. What has changed is the maturity of each methodology.

There have been a lot of tools and techniques developed over the past few years that have greatly improved the efficiency and effectiveness of fuzzing. I think one of the most important innovations is fuzzers like AFL (http://lcamtuf.coredump.cx/afl/) that use code coverage measurements to guide fuzzing, so that the fuzzer can focus on testing new and unexplored areas of software. Also important are tools that allow for fuzzing to be performed at scale, for developers to easily integrate fuzzing into the development process, and for errors to be more consistently detected when fuzzers hit them.

The flip side of this is that, in general, it is more difficult to find bugs with fuzzing these days. I think this is due to more security awareness among developers, as well as more software teams fuzzing their code as a part of the development process. Fifteen years ago, it was common to find security bugs using simple mutation fuzzing on a single host in a few hours. Now it usually takes more advanced techniques on multiple cores.

Code review techniques have been fairly consistent throughout the years, although now, of course, we know a lot more about bug classes and how attackers can exploit them. It is also generally more challenging to find security bugs with code review these days, probably because software is both better tested and more complex.

*RF:* As part of Project Zero, do you ever work as groups/teams on a project?

*NS:* Yes, we do. In fact, I worked on a large research project on the iPhone [1–5] with Samuel Groß last year. We also do team hackathons a few times a year where we work on the same target together. While we do a lot of independent research, there's a lot to be gained by sharing ideas!

*RF:* Do you and others in Project Zero get direction on what software to search, or can you pick and choose?

*NS:* Project Zero's mission is to "make zero-day hard," and we pick our targets based on this mission. Usually, this means software that has a large user base, a history of being targeted by certain attackers, and/or a vulnerable user base. Team members are free to pick their own targets within the mission, although we also often discuss targets and make goals as a team.

*RF:* You wrote about what someone should do to get hired at Project Zero (https://googleprojectzero.blogspot.com/p/working-at-project-zero.html), and I wondered if you have thought of anything you'd like to add since you posted that?

*NS:* Not really, but I would like to mention that vulnerability research is just one of the many careers available in information security, and that post is very specific to our team.

*RF:* Are there other women working on the Project Zero team? In my experience, the number of women working in security is even lower than in other areas in IT—much lower.

*NS:* I was the only woman on Project Zero for about four years, but we've recently been joined by the amazing Maddie Stone. There are fewer women in information than a lot of other IT fields, but it's improved somewhat over the last few years.

*RF:* With your goal of making zero-day hard, I wonder what things you consider can make security better. I find myself surprised that things have gotten better, as most programmers are average in skill, and the languages they most often write in, C and C++, are the same as they were when they were first created when it comes to security. For example, a programmer can still use `gets()` on Linux, and buffer overflows are still possible, although compiler support for protecting the stack has pushed their exploitation to the heap.

*NS:* This is a huge question, because there are so many ways to improve software security. And I also want to qualify "things getting better"—while I suspect there are fewer bugs per line of code today than there were in the past, there is also more software being used by more users for more applications than ever before. So overall, software security is a more important problem than it has ever been.

Taking the example of a call to `gets()` that causes an overflow, there's a lot of things that can happen during the development process that can stop it from getting into release code. For example:

- The developer understands that `gets()` can lead to vulnerabilities, and doesn't use it.
- The developer's compiler or development environment warns them about `gets()`, and they remove it.
- The repository they submit the code to has pre-submit or compiler checks that reject `gets()`, and the developer can't submit their code until they fix it.
- Submitting code requires the commit to be reviewed by another developer, and that developer finds and fixes the bug.
- The code in the repository is automatically fuzzed, and the bug gets found before release.
- The code is security reviewed before it is released, and the bug gets found before release.
- The crash occurs during beta testing, and the developer fixes it based on the crash log.
- The release binary contains mitigations that make it more time-consuming to exploit memory corruption bugs.

Good "development discipline" can greatly reduce the number of security (and other) bugs in software, and there are a lot of tools and technology available to help with this. Of course, this requires that the organization produce the software to prioritize and invest in security, which is unfortunately not always the case.

*RF:* While I am still a fan of LangSec (langsec.org), I now realize that it is just a part of the overall picture of secure programming practices. What do you think of LangSec, and where do you see that LangSec falls short of what programmers need to be doing?

*NS:* LangSec aims to improve software security by creating formally verifiable languages and parsers that are immune to many common security problems. They view the root cause of security issues to be that most protocols and other input formats are poorly defined and often have many undefined states, and the programming languages that process them also support a huge amount of undefined behavior. They think all software should abstract out all input processing code, and design and implement it in a way that is verifiable and has no undefined states or behavior.

One observation behind LangSec's philosophy is that the language software is written in has a huge influence on the number of vulnerabilities it contains. There is a lot of evidence for this. The most important distinction in my mind is managed (does not allow dynamic memory allocation) versus unmanaged (allows dynamic memory allocation) languages. Since the majority of vulnerabilities exploited by attackers are memory corruption vulnerabilities that occur due to the misuse of dynamically allocated memory, even just moving to dynamic languages has a lot of potential to reduce the number of vulnerabilities in software.

LangSec's goal is lot broader than increasing the use of managed languages, though. Dynamically allocated memory is just one of the causes of the undefined and unverifiable software behavior they want to prevent. Unfortunately, while there would be a lot of benefits to fully verifiable input processing, the reality is that technology is not quite there yet. Even just with managed languages, there are a lot of reasons that developers don't use them, including performance, capabilities, and compatibility with legacy code, and formally verifiable languages have even more limitations. So while LangSec's ideas are very promising for the future, I feel that a lot more work needs to be done before their work is practical for most applications.

Another concern is that LangSec's approach doesn't prevent logic bugs. For example, imagine a shopping website that notifies the warehouse to ship an item before it collects payment. This design has a security problem where if a user gets to the point where the shopping service notifies the warehouse to ship, and then the user stops interacting with the site, the user will get the item for free. Formal verification won't prevent this type of problem, it will only check that the implementation conforms exactly to the design. It is also likely that any formally verifiable language or parser has at least some bugs in it (because all software has bugs), which could lead to security bugs in software that uses that language. It's also possible attackers think of new types of vulnerabilities that no one has thought of yet. So while LangSec's approach would likely greatly reduce the number of vulnerabilities in software, it won't eliminate all of them.

That said, there are two important takeaways from LangSec's approach that developers can use right now. One is that the language they choose to write software in impacts its security a lot. The other is that design is really important. The better defined a feature is, and the more thought that is given to making it easy to implement securely, the more likely it is to be secure.

*RF:* Other than good "development discipline," what else can programmers prevent to make their software more secure?

*NS:* One important strategy for improving software security is Attack Surface Reduction. Put simply, every piece of software has a portion of code that can be manipulated by attackers, and making this as small as possible can have huge returns with regards to preventing vulnerabilities. It's not unusual for Project Zero to find bugs in software features that have low or no usage, meaning they present security risk to users with little benefit. It's important for developers to be aware that all code creates a security risk and other bugs, and to make sure that tradeoff makes sense.

***References***

[1] The Fully Remote Attack Surface of the iPhone: https://googleprojectzero.blogspot.com/2019/08/the-fully-remote-attack-surface-of.html.

[2] The Many Possibilities of CVE-2019-8646: https://googleprojectzero.blogspot.com/2019/08/the-many-possibilities-of-cve-2019-8646.html.

[3] Remote iPhone Exploitation, Part 1: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-1.html.

[4] Remote iPhone Exploitation, Part 2: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-2.html.

[5] Remote iPhone Exploitation, Part 3: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html.

# Anomalies in Linux Processor Use

RICHARD L. SITES

Richard L. Sites is a semi-retired computer architect and software engineer. He received his PhD from Stanford University several decades ago. He was co-architect of the DEC Alpha computers and then worked on performance analysis of software at Adobe and Google. His main interest now is to build better tools for careful non-distorting observation of complex live real-time software, from datacenters to embedded processors in vehicles and elsewhere. dick.sites@gmail.com

Careful observation of Linux dynamic behavior reveals surprising anomalies in its schedulers, its use of modern chip power-saving states, and its memory allocation overhead. Such observation can lead to better understanding of how the actual behavior differs from the pictures in our heads. This understanding can in turn lead to better algorithms to control dynamic behavior.

We study here four such behaviors on x86-64 systems:

1. Scheduling dynamics across the Completely Fair Scheduler, the real-time FIFO scheduler, and the real-time Round-Robin scheduler
2. Dynamic use of `mwait`-sleep-wakeup to save power
3. Dynamic CPU clock frequency changes to save power
4. Invisible cost of heap allocation just *after* allocation

In each case, the interaction of Linux and the underlying hardware can be improved in simple ways.

The software observation tool is KUtrace [1–3], which timestamps and records every transition between kernel-mode and user-mode execution in a live computer system, using less than 1% CPU and memory overhead and thus observing with minimal distortion. Each transition is recorded in just four bytes in a kernel trace buffer—20 bits of timestamp and 12 bits of event number (syscall/sysreturn, interrupt/return, fault/return numbers plus context-switch new process ID, and a handful of other items). Everything else is done by postprocessing a raw binary trace. Depending on the processor, each trace entry takes an average of 12–20 nsec to record, about 30 times faster than `ftrace` [4]. The robustly achieved design point is to handle 200,000 transitions per second per CPU core with less than 1% overhead. I built the first such system at Google over a decade ago, and it and its offspring have been used in live production datacenters since.

## Linux Schedulers: Not Completely Fair

The Linux CPU schedulers juggle program execution by assigning tasks to CPU cores at various times. The Completely Fair Scheduler (CFS) runs each task at equal speed, each getting CPUs/tasks speed over time. The FIFO real-time scheduler runs each task in FIFO order to completion or until it blocks. The Round-Robin real-time scheduler runs like FIFO but imposes a maximum time quantum, moving tasks to the end of a run queue in round-robin fashion at quantum boundaries.

On a four-core Intel i3-7100 processor (actually two physical cores hyperthreaded) running the Linux 4.19.19 LTS (long-term support) kernel version, I ran 1 to 12 identical CPU-bound threads and observed the true scheduling behavior [5]. Each thread repeatedly checksums a 240 KB array that fits into a per-core L2 cache. From the Linux documentation, I expected the resulting timelines for more than four tasks to show each task running periodically and all completing at nearly the same time. Not so.

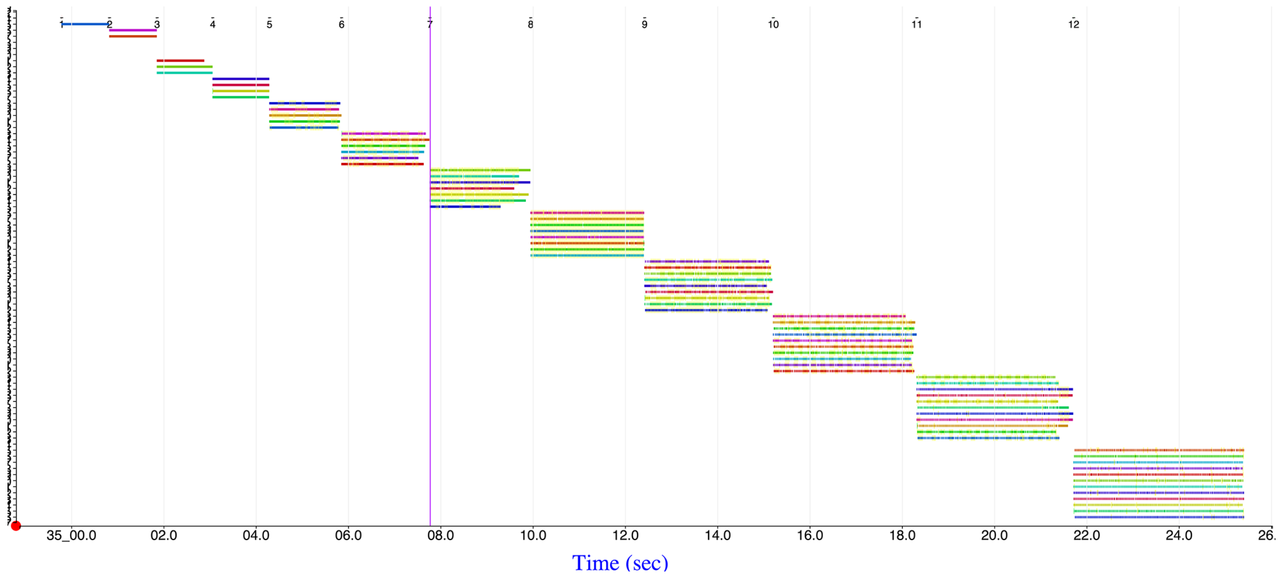


**Figure 1:** Running groups of 1 to 12 compute threads under CFS. The main program spawns one thread at the top left, and when that completes one second later it spawns two threads, then three, etc. With only four logical CPU cores, the scheduler starts its real work with five or more threads. The vertical line marks the group of seven that is expanded in Figure 2.

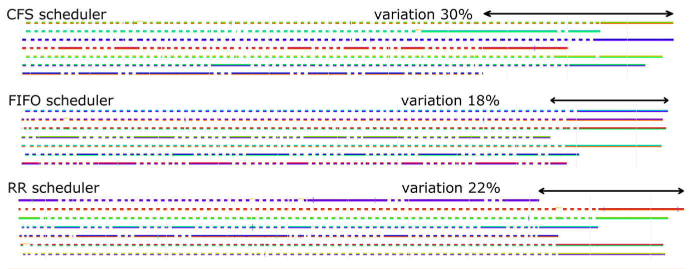


**Figure 2:** Running groups of seven compute-bound threads under the three Linux schedulers, shown over about two seconds total. In each case, the thread-preemption times vary substantially, and some threads complete unexpectedly much sooner than others—arrows depict the largest differences.

Figure 1 shows groups of 1 to 12 threads running under CFS. As the last thread of each group finishes, the next group starts, consuming about 26 seconds in all. The pictures for the other schedulers look similar at this scale. (Note that all the figures in this article appear in color in the online version.)

Looking at just the seven-thread group, Figure 2 shows it for each of the three schedulers. The smallest dashes are 12 ms execution periods (the quantum), chosen by the scheduler based on four cores and timer interrupts every 4 ms. This simple example does not stress the differences that the real-time schedulers would provide in a mixture of batch and real-time programs, but it does highlight their underlying dynamics.

The documentation for these schedulers did not lead me to expect that some tasks would run uninterrupted for a dozen quanta or more, nor did it lead me to expect a 20–30% variation in completion time between the earliest and latest ones. None of this approaches "completely fair." Observing these actual dynamics can lead to better algorithms.

## Deep Sleep: Too Much Too Soon

Our second study concerns power-saving dynamics. Modern software passes hints to the chip hardware that nothing interesting will be executing for a while, so the hardware might well want to slow down or turn off a core to save (battery) power. For x86 processors, the Linux idle-process code issues `mwait` instructions to suggest sleep states to the hardware. Deep sleep states such as Intel C6 involve turning off a CPU core and its caches (first doing any necessary writebacks). When a subsequent interrupt arrives at that core, the hardware and microcode first crank up the CPU core's clock and voltage, write good parity/ECC bits in the cache(s), and eventually execute the first instruction of the interrupt handler. Coming out of C6 deep sleep in an Intel i3-7100 takes about 30 microseconds, delaying interrupt handling by that amount.

You might not think that this matters much until you observe the dynamics of multiple communicating threads sending interprocessor interrupts to each other just as the receiving core has gone to sleep, and when that one responds, the reply goes back to a core that in turn has just gone to sleep. Rinse and repeat.

Figure 3 shows just such a sequence, at the beginning of launching the group of seven threads in the program in the previous section. Note that Figures 1–6 show everything happening on every CPU core every nanosecond—all idle time and kernel and user execution time for all programs, with nothing missing. For

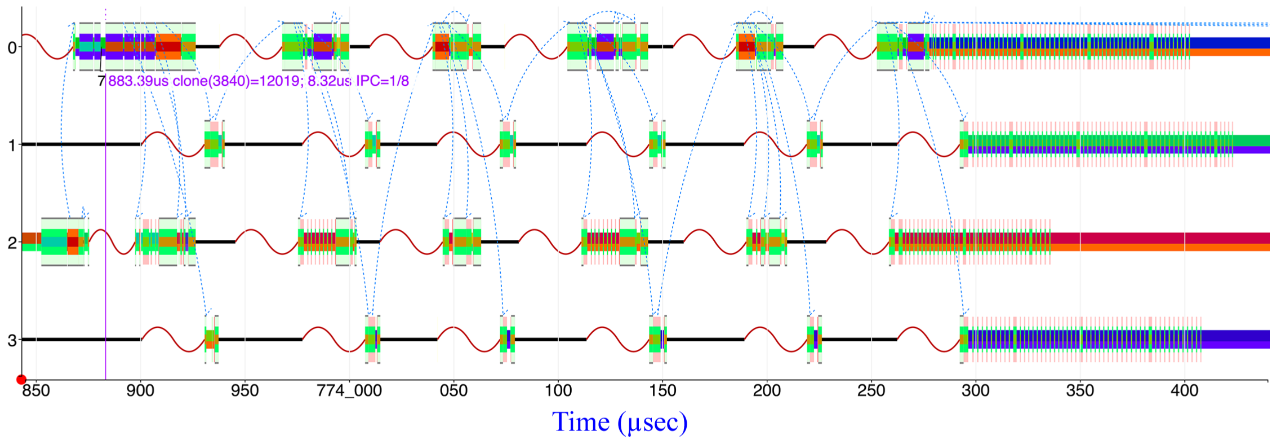


**Figure 3:** Premature sleep in the `intel_idle.c` Linux kernel code causes startup delays for seven spawned threads. Thin black lines are the idle process, and sine waves are the time it takes a chip core in deep sleep to wake up again. Heavier lines on the right are compute-bound execution of four of the seven threads on the four CPU cores.

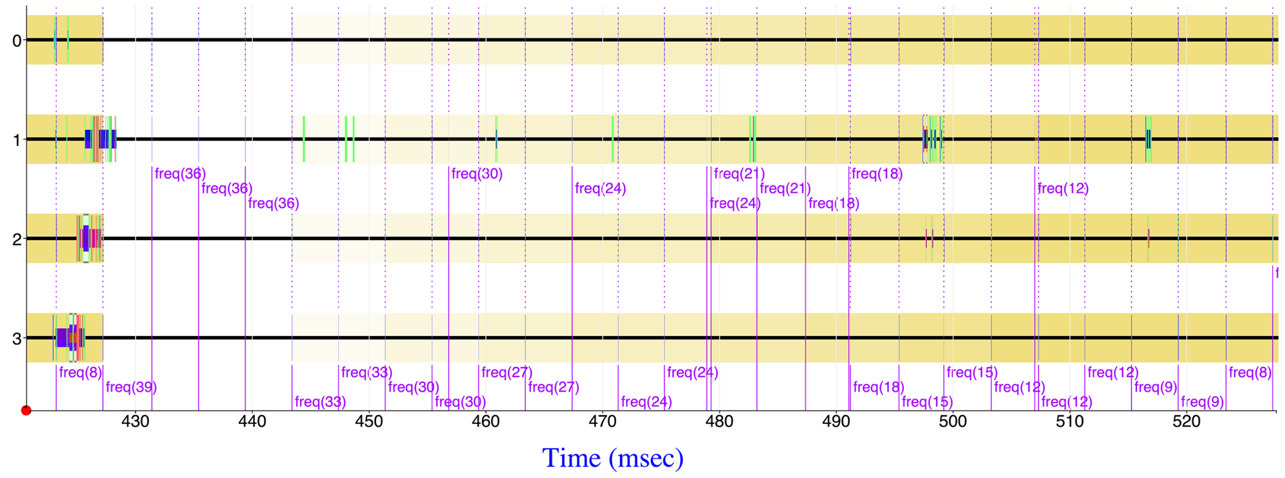


**Figure 4:** Non-idle execution on three CPUs at the left triggers a jump in all four CPU clock frequencies from slowest 800 MHz to fastest 3.9 GHz, which then step back to slow (frequency in units of 100 MHz indicated by the shading surrounding the lines for each CPU).

example, Figure 1 also has threads with names like `systemd-journal`, `cron`, `avahi-daemon`, `sshd`, and `DeadPoolForage`. None of these take any appreciable CPU time, so I cropped out most of them except the three cron jobs that run near time 1.8 seconds and take up a little vertical space between the group of two threads and the group of three threads in that figure.

The thin black lines in Figure 3 are the idle process executing, while the tall gray/colored lines are kernel-mode execution, and the half-height gray/colored lines are user-mode execution. The sine waves are the time coming out of C6 sleep (the time spent in deep sleep is short here, but is often several milliseconds). The dotted arcs show one process waking up another.

The idle threads do an `mwait` instruction to sleep after spinning for only 400–900 nsec, which is much too soon. In the diagram, the first four of seven `clone()` calls are on CPU 0 at the upper left, and the spawned threads start executing on CPUs 3, 2, 2, and 1, respectively, just after and just below. Each child thread blocks almost immediately inside `page_fault`, waiting for the parent to finish setting up shared pages. Full execution of four threads begins only on the right side of the diagram. The bouncing back and forth between parent and child keeps encountering ~50 μs delays because the CPU cores prematurely go into deep sleep.

There are two problems here: (1) 30 μs is a long time to be recovering from a siesta, ruining any expectations of fast interrupt response, for example, and (2) violation of the Half-Optimal Principle [6]:

```
If it takes time T to come out of a state waiting for some
event at unknown time E in future, spin at least time T
before going into that state. This is at least half-optimal
in all cases, even though you don't know the future.
```

In this case, the half-optimal strategy is to spin for 30 μs instead of 0.4–0.8 μs before dropping into a C6 sleep state that takes

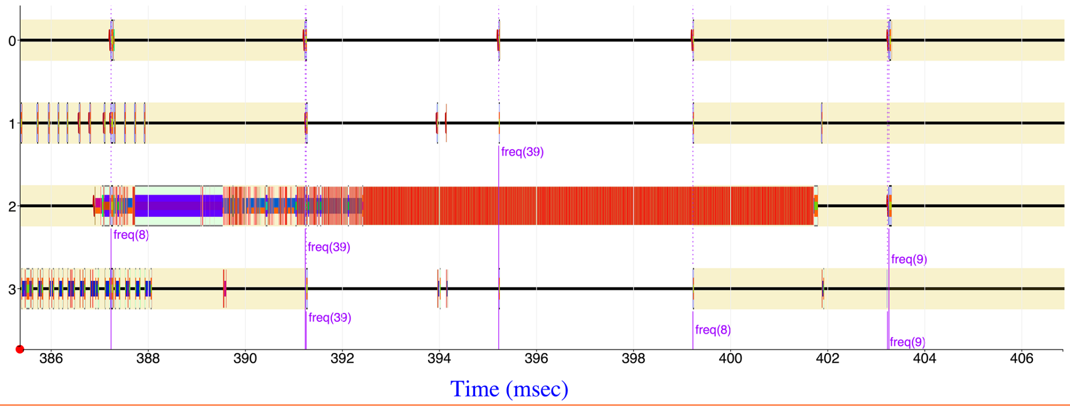


**Figure 5:** Non-idle execution at the left triggers a jump in CPU clock frequencies from slowest to fastest, which prematurely jump back to slow while CPU 2 is still 100% busy.

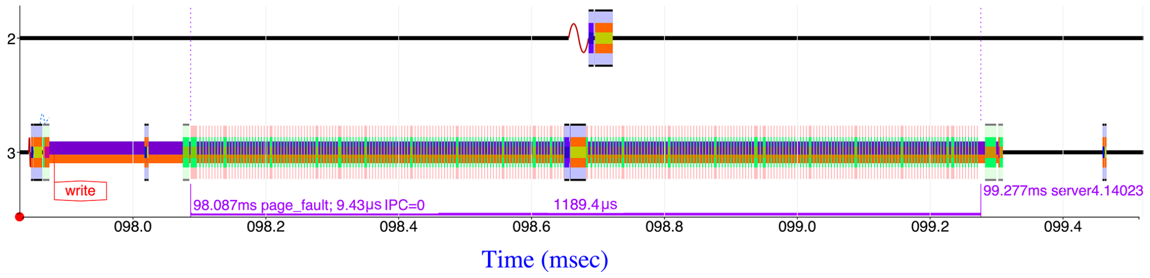


**Figure 6:** The page faults immediately after allocating memory take over 100x more time than the allocation itself.

30 µs to exit. Doing so would completely avoid sleeping in the trace above and would speed up the initial song-and-dance by over 3x. Observing these actual dynamics can lead to better algorithms.

## Fluctuating Frequency: Mismatched to Goal

Our next study looks at another power-saving technique—varying the CPU clock frequency for each core. The goal is to use slow clocks when not much execution is happening and to use fast clocks when doing real computing. The measured Intel i3-7100 chip core clocks vary between 800 MHz and 3.9 GHz. For this processor, Linux allows the chip hardware to dynamically vary the clock frequency—"HWP enabled" in the Linux kernel Intel x86 jargon. Once enabled, no operating system code runs to vary the frequency or even to deliver an event when the frequency changes. However, a machine-specific register can be read to get some hint of the likely upcoming frequency. I added code to read that register at every timer interrupt and add it to the raw KUtrace.

For a computing load to observe, I ran a command to find some files and look for a regular expression in them:

```
    find ~/linux-4.19.19/ -name "*.h" |xargs grep rdmsr
```

and then traced that for 20 seconds. This runs three programs, `find`, `xargs`, and `grep`. The first two mostly wait for disk while reading directories, and the last is mostly CPU-bound scanning a file. I picked this combination because I expected low CPU clock rates while waiting for disk and higher ones while scanning files.

Figure 4 shows an execution timeline on four CPU cores running mostly the `bash`, `find`, and `xargs` programs but with a little bit of other processes such as `jbd2`, `ssh`, and `chrome`. The gray overlay (yellow in the online version) shows CPU clock speeds: dark gray for slow clocks and lighter and lighter for faster clocks. The `freq` numbers are multiples of 100 MHz. Based on the non-idle program execution at the far left on CPUs 1, 2, and 3, the chip switches from 800 MHz to 3.9 GHz on all four CPU clocks, then slowly, over about 100 ms, drops the frequency back to 800 MHz. These are the true clock dynamics and match what one would expect from reading the (sparse) documentation. Note, however, that the execution bursts on CPU 1 in the right half of the diagram do not raise the clock frequency.

In contrast to the intended behavior, Figure 5 shows a region of the same trace eight seconds later. This time the clock frequency jumps up from 800 MHz to 3.9 GHz as expected, but 8 ms later it jumps back to 800 and then 900 MHz, even though CPU 2 is still quite busy running grep.

This dynamic is mismatched to the performance goal of the power-management design. Observing these actual dynamics can lead to better algorithms.

## Cost of Malloc: Not There but Soon Thereafter

Our final study looks at memory allocation. In a client-server environment with the client sending 1,000,000-byte database write messages to the server, the server trace reveals a user-mode allocation of 1,000,000 bytes for receiving the message, followed by 245 page faults (`ceil` of 1,000,000/4096), the repeating blips on CPU 3 in Figure 6. You can see similar page-fault bursts in the completely different program at the far right of Figure 3. The big blips near time 98.7 ms are timer interrupts. You can directly see in the ~30 μs skew in delivering timer interrupts on sleeping CPU 2 and on busy CPU 3.

The allocation takes a few microseconds in the underlying system call just before the page faults, but the page faults themselves take over 1100 microseconds. The (good) Linux design for extending heap allocation simply creates 245 read-only page table entries pointing to the kernel's single all-zero page. As the user-mode program moves data into this buffer, each memory write to a new page takes a page fault, at which time the page-fault handler allocates a real memory page, does a copy-on-write (CoW) to zero it to prevent data security leaks between programs, sets the page table entry to read-write, and returns to redo the memory write. This goes on for 245 pages, taking much, much longer than the allocation time that is visible in many profiling tools. The dominant page-fault time is *invisible* to normal observation tools.

The copy-on-write path itself is inefficient in several ways. First, it could do groups of 4–16 pages at once, saving some page-fault entry/exit overhead without spending excess time in the fault routine and without allocating too many real pages that might never be used. Second, the kernel code does not special-case the Linux `ZERO_PAGE` as source to avoid reading it, by something like:

```
if (src == ZERO_PAGE)
        memset(dst, 0, 4096);
else
        memcpy(dst, src, 4096);
```

Doing so would avoid reading an extra 4 KB of zeros into the L1 cache each time and would avoid half of the memory accesses. It would also speed up the instructions per cycle (IPC) of the CoW inner loop.

A `malloc` call that reuses previously allocated heap space does not have the behavior seen here, but one that extends the heap does. Some programs aggressively extend and then shrink the heap repeatedly, wasting time not only in `malloc/free` but also in page faults. Allocating a buffer once and then explicitly reusing it in user code can be faster, for example. Observing these actual dynamics can lead to better algorithms.

## Conclusion

Careful observation of Linux dynamic behavior reveals surprising anomalies in its schedulers, its use of modern chip power-saving states, and its memory allocation overhead. Such observation can lead to better understanding of how the actual behavior differs from the pictures in our heads. This understanding can in turn lead to better algorithms and better control of dynamic behavior.

As an industry, we have poor nondistorting tools for observing the true dynamic behavior of complex software, including the operating system itself. KUtrace is an example of a better tool. I encourage operating-system designers to provide such extremely-low-overhead, and hence nondistorting, tools in future releases.

### *References*

[1] R. L. Sites, "Benchmarking 'Hello World'," *ACM Queue Magazine,* vol. 16, no. 5 (November 2018): https://queue.acm.org/detail.cfm?id=3291278.

[2] R. L. Sites, "KUTrace: Where Have All the Nanoseconds Gone?" Tracing Summit 2017 (11:00 a.m., slides and video): https://tracingsummit.org/wiki/TracingSummit2017.

[3] Open-source code for KUtrace: https://github.com/dicksites/KUtrace.

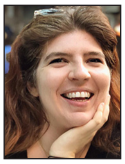
[4] Ftrace function tracer: https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[5] Credit to Lars Nyland at Nvidia for first showing me this.

[6] I have used this principle for many years but only created the name while writing this article. A related Half-Useful Principle applies to disk transfers and other situations with startup delays: if you spend 10 ms doing a seek, then try to spend 10 ms transferring data (1 MB+ these days), so that at least half the time is useful work.

# Revisiting Conway's Law

MARIANNE BELLOTTI



Marianne Bellotti has worked as a software engineer for over 15 years. She built data infrastructure for the United Nations to help humanitarian organizations share crisis data worldwide and spent three and a half years running incident response for the United States Digital Service. While in government she found success applying organizational change management techniques to the problem of modernizing legacy software systems. More recently, she was in charge of Platform Services at Auth0 and currently works as Principal Engineer for System Safety at Rebellion Defense. She has a book on running legacy modernization projects coming out this year from No Starch Press called *Kill It with Fire.* marianne.bellotti@gmail.com

After more than six years helping engineering organizations figure out how to modernize their technology, I've come to realize that Conway's Law is more about how organizational structure creates incentives than where boxes and lines are drawn on an org chart. Misaligned incentives for managers and individual contributors carve their impact into the system design, influencing tool selection and complicating future maintenance.

In 1968 Melvin Conway published a paper titled "How Do Committees Invent?" This paper, originally intended for *Harvard Business Review* but rejected for being too speculative, outlined the ways in which the structure and incentives of an organization influenced the software product it produced. It received little response but eventually made its way to the chair of the University of North Carolina at Chapel Hill's computer science department, Fred Brooks. At the time, Brooks had been pondering a question from his exit interview at IBM: why is it so much harder to manage software projects than hardware projects? Conway's insight linking the structure of software to the structure of the committees that invented them seemed significant enough for Brooks to repackage the thesis as "Conway's Law" when he published his guide on effectively managing software teams—*The Mythical Man-Month*—in 1975.

Yet this was not the only useful observation in Conway's paper. As it has subsequently been referenced by hundreds of computer science texts since Brooks's adoption of it as a universal truth, the more nuanced observations that supported Conway's argument have largely been omitted from the conversation. Conway's Law has become a voodoo curse, something that people believe only in retrospect. Few engineers attribute their architectural success to the structure of their organization, but when a product is malformed the explanation of Conway's Law is easily accepted.

Conway's original paper outlined not just how organizational structure influenced technology but also how human factors contributed to its evolution. Conway felt organizational structure influenced architecture because organizational structure influenced incentives. How individual contributors get ahead in a particular organization determined which technical choices were appealing to them.

Conway's observations are more important in maintaining existing systems than they are in building new systems. Organizations and products both change, but they do not always change at the same pace. Figuring out whether to change the organization or change the design of the technology is just another scaling challenge.

## Individual Incentives

How do software engineers get ahead? What does an engineer on one level need to accomplish for the organization in order to be promoted to another level? Such questions are usually delegated to the world of engineering managers and not incorporated in technical decisions. And yet the answers absolutely have technical impacts.

Most of us have encountered this in the wild: a service, a library, or a piece of a system that is inexplicably different from the rest of the applications it connects to. Sometimes this is an older component of the system reimplemented using a different set of tools. Sometimes this is a new feature. It's always technology that was trendy at the time the code was introduced.

When the organization has no clear career pathway for engineers, software engineers grow their careers by building their reputation externally. This means getting drawn into the race of being one of the first to prove the production scale benefits of a new paradigm, language, or technical product. While it's good for the engineering team to experiment with different approaches as they iterate, introducing new tools and databases, and supporting new languages and infrastructures, increases the complexity of maintaining the system over time.

One organization I worked for had an entire stable of custom-built solutions for things like caching, routing, and message handling. Senior management hated this but felt their complaints—even their instructions that it stop—did little to course correct. Culturally, the engineering organization was flat, with teams formed on an ad hoc basis. Opportunities to work on interesting technical challenges were awarded based on personal relationships, so the organization's regular hack days became critical networking events. Engineering wanted to build difficult and complex solutions in order to advertise their skills to the lead engineers who were assembling teams.

Stern lectures about the importance of choosing the right technology for the job did not stop this behavior. It stopped when the organization hired engineering managers who developed a career ladder. By defining what the expectations were for every experience level of engineering and by hiring managers who would coach and advocate for their engineers, engineers could earn promotions and opportunities without the need to show off.

Organizations end up with patchwork solutions because the tech community rewards explorers. Being among the first with tales of documenting, experimenting with, or destroying a piece of technology builds an individual's prestige. Pushing the boundaries of performance by adopting something new and innovative contributes even more so to one's reputation.

Software engineers are incentivized to forego tried-and-true approaches in favor of new frontiers. Left to their own devices, software engineers will proliferate tools, ignoring feature overlaps for the sake of that one thing tool X does better than tool Y that is only relevant in that specific situation.

Well-integrated, high-functioning software that is easy to understand usually blends in. Simple solutions do not do much to enhance personal brand. They are rarely worth talking about. Therefore, when an organization provides no pathway to promotion for software engineers, the engineers are incentivized to make technical decisions that emphasize their individual contribution over smoothly integrating into an existing system.

Typically this manifests itself in one of three different patterns:

1. Creating frameworks, tooling, and other abstraction layers in order to make code that is unlikely to have more than one use case theoretically "reusable."
2. Breaking off functions into new services, particularly middleware.
3. Introducing new languages or tools in order to optimize performance for the sake of optimizing performance (in other words, without any need to improve an SLO or existing benchmark).

Essentially, engineers are motivated to create named things. If something can be named it can have a creator. If the named thing turns out to be popular, then the engineer's prestige is increased and her career will advance.

This is not to say that good software engineers should never create a new service, or introduce a new tool, or try out a new language on a production system. There just needs to be a compelling reason why these actions benefit the system versus benefit the prospects of the individual engineer.

Most of the systems I work on rescuing are not badly built. They are badly maintained. Technical decisions that highlight individuals' unique contributions are not always comprehensible to the rest of the team. For example, switching from language X to language Z may in fact boost memory performance significantly, but if no one else on the team understands the new language well enough to continue developing the code, those gains will be whittled away over time by technical debt that no one knows how to fix.

The folly of engineering culture is that we are often ashamed of signing our organization up for a future rewrite by picking the right architecture for right now, but we have no misgivings about producing systems that are difficult for others to understand and therefore impossible to maintain. This was a constant problem for software engineers answering the call to public service from organizations like United States Digital Service and 18F. When modernizing a critical government system, when should the team build it using common private sector tools and train the government owners on said tools, and when should the solution be built with the tools the government worker already knows? Wasn't the newest, greatest web application stack always the best option? Conway argued against aspiring for a universally correct architecture. He wrote in 1968, "It is an article of faith among experienced system designers that given any system design, someone someday will find a better one to do the same job. In other words, it is misleading and incorrect to speak of *the* design for a specific job, unless this is understood in the context of space, time, knowledge, and technology."

## Manager Incentives

An engineering manager is a strange creature in the technical organization. How should we judge a good one from a bad one? Unfortunately, far too often managers advance in their careers by managing more people. And if the organization isn't properly controlling for that, then system design will be overcomplicated by the need to broadcast importance. Or as Conway put it: "The greatest single common factor behind many poorly designed systems now in existence has been the availability of a design organization in need of work."

Opportunities to go from an engineering manager and senior engineering manager come up from time to time as the organization grows and changes. It's the difference between handling one team and handling many. Managers leave, new teams form, existing teams grow past their ideal size. A good manager could easily earn those opportunities in the normal course of business. Going from senior manager to director, though, is more difficult. Going from director to vice president or higher even more so. It takes a long time for an organization to reach that level of growth organically.

Organizations that are unprepared to grow talent end up with managers who are incentivized to subdivide their teams into more specialized units before there is either enough people or enough work to maintain such a unit. The manager gets to check off career-building experiences of running multiple teams, hiring more engineers, and taking on more ambitious projects while the needs of the overall architecture are ignored.

Scaling an organization before it needs to be scaled has very similar consequences to scaling technology too early. It restricts your future technical choices. Deciding to skip the monolith phase of development and "build it right the first time" with microservices means the organization must successfully anticipate a number of future requirements and determine how code should be best abstracted to create shared services based on those predictions. Rarely if ever are all of those predictions right, but once a shared service is deployed, changing it is often difficult.

In the same way, a manager who subdivides a team before there is need to do so is making a prediction about future needs that may or may not come true. In my last role, our director of engineering decided the new platform we were building needed a dedicated team to manage data storage. Predictions about future scaling challenges supported her conclusions, but in order to get the head count for this new team, she had to cut it from teams that were working on the organization's existing scaling challenges. Suddenly, new abstractions around data storage that we didn't need yet were being developed while systems that affected our SLAs had maintenance and updates deferred.

Carrying existing initiatives to completion was not as attractive an accomplishment as breaking new ground. But the problem with designing team structure around the desired future state of the technology is that if it doesn't come true the team is thrown into the chaos of a reorganization. Aversions to reorganizations alone often incentivize people to build to their organizational structure.

## Conclusion

Both individual contributors and managers make decisions with their future careers in mind. Those decisions create constraints on possible design choices that drive the organization to design systems that reflect the structure of the organization itself. Those wishing to benefit from the forces of Conway's Law would do well to consider how people within the engineering organization are incentivized before asking them to design a system.

# Decision-Making Using Service Level Objectives

ALEX HIDALGO

Alex Hidalgo is a Site Reliability Engineer and author of the forthcoming *Implementing Service Level Objectives* (O'Reilly Media, September 2020). During his career he has developed a deep love for sustainable operations, proper observability, and use of SLO data to drive discussions and make decisions. Alex's previous jobs have included IT support, network security, restaurant work, t-shirt design, and hosting game shows at bars. When not sharing his passion for technology with others, you can find him scuba diving or watching college basketball. He lives in Brooklyn with his partner Jen and a rescue dog named Taco. Alex has a BA in philosophy from Virginia Commonwealth University. sometimesitsalex@gmail.com @ahidalgosre

Service level objectives, or SLOs, are quickly becoming the latest industry buzzword. Engineers want them, leadership demands them, and job postings increasingly ask for experience with them. However, SLOs are meaningless unless they are understood as more than just the latest industry jargon. There are true, real-world benefits to adopting an SLO-based approach to reliability. I will explain why they are important and how you can use them most effectively to have discussions that lead to better decisions.

Using service level objectives to measure the reliability of services is getting more attention than ever before. This is partly due to the success of the first two Google-authored site reliability engineering (SRE) books. But it also seems that many people actually resonate with the approach and find it an intuitive concept to follow. While it is possible that many organizations are forcing their teams to adopt SLOs via mandate just to ensure they're on board with the latest buzzwords, it also seems likely that many people are finding true value in the approach.

I found only one study tracking the adoption rates of SLO-based approaches in this book: https://www.oreilly.com/library/view/slo-adoption-and/9781492075370/. Instead, I'll have to rely on the general anecdotal evidence I have in terms of how many companies are rolling out products to help people measure SLOs, how many conference talks are focused on them, and how often I personally find myself engaged with people who want to learn more about the process.

While SLO-based approaches to reliability are certainly useful to many people, I also cannot ignore the fact that the very phrase has become a buzzword that is starting to lose meaning. It's not uncommon for words, phrases, and concepts that gain traction and desirability to have their original meanings forgotten. Service level objectives are no different. They provide many benefits, but some of their most important aims have unfortunately become obfuscated by more readily available ones.

SLO-based approaches to reliability give you many benefits, and there are many reasons why organizations may choose to adopt them. Unfortunately, for many they have just become "a thing you do." This is not to say that every organization looking to adopt such an approach has overlooked the benefits of SLOs, but few manage to use them to their full potential.

Let's explore some of the ways you can use the information that service level objectives provide to make better decisions through data. Making better decisions is at the very heart of the SLO approach, and that's often the part that is overlooked.

But first, let's outline how this approach works in a little more detail.

## SLO Components

There are three primary components to an SLO-based approach. The first is *service level indicators*, or SLIs. A good SLI is a measurement that tells you how your service is performing from the perspective of your users. In this case, when I say users, they could be anything from paying customers to coworkers to other services that depend on yours—there doesn't strictly have to be a human attached to the other end. In this article, I'll mostly be talking about human users who interact with web services since they are intuitively accessible concepts that almost all of us interact with on a daily basis; however, the concepts and approaches apply to any service and any type of user, even if those users are just other computer systems.

After SLIs, you have *service level objectives*, which are targets for how you want your SLI to perform. While a service level indicator may tell you how quickly a web page loads, an SLO allows you to do things like set thresholds and target percentages. An example SLI might be "Web pages are fully rendered in the user's browser within 2500 ms." Building off of that, an SLO might read, "The 95th percentile of web page render times will complete within 2500 ms 99.9% of the time." Service level objectives allow you to set reasonable targets. Nothing is ever perfect, and 100% is impossible for just about everything, but by using SLOs, you can ensure that you're striving for a reasonable target and not an unreachable one.

Finally, you have *error budgets*. An error budget is a way of keeping track of how an SLO has performed over time. If you acknowledge that only 99.9% of the 95th percentile of your web page render times have to complete within 2500 ms, you are also acknowledging that 0.1% of them don't have to. Error budgets give you a way to do the math necessary to determine whether your adherence to your SLO target is suitable for your users, not just in the moment but over the last day, week, month, quarter, or year.

SLIs, SLOs, and error budgets are all data—data that allows you to ask important questions that can drive better decision-making.

- Is our SLI adequately measuring what our users need and expect? If not, we need to figure out a new way to measure this.
- Is our SLO target meaningfully capturing the failure rates our users can tolerate? If not, we need to pick a new target or new thresholds.
- What is our error budget status telling us about how our users have actually experienced our service over time? If we've exceeded the error budget, perhaps we drop feature work and focus on reliability instead.

## Decisions about User Experience

A meaningful SLI is one that captures the user experience as closely as possible. Following our simple example from above, it is pretty intuitive to think about the fact that the users of a web service need their pages to load—and to load in an amount of time that won't annoy them. But there is so much more to the user experience than just the concepts of availability and latency. A web service is not doing its job just by being able to render pages in a timely manner. If you're only measuring things like availability and latency, the only data SLO-based approaches can provide you are ones that focus on improving your availability and latency.

A web service is often much more than just serving data to people. Imagine that your web service is a retail site. In such a case, you suddenly have many other user journeys to consider. If you want people to be able to purchase items from you, they need to be able to do exactly that and not just have web pages display in their browser.

For example, a standard retail website often has some sort of *shopping cart* feature—one where a user can add a potential purchase to a list of items they might want to check out with later. This shopping cart feature has to do a lot of things in order to be reliable.

The first is that it needs to do what it is supposed to: if a user wants to add an item to their shopping cart, they should be able to do exactly that. Additionally, it needs to be persistent; a shopping cart isn't much good if it only remains consistent with the wishes of a user for a short amount of time.

It also has to be accurate. There is no sense in allowing customers to add to a list of items they might want to purchase if that list doesn't represent the items they have actually chosen.

Finally, how the user interacts with the shopping cart has to work properly. If an item is added or removed, it should actually be added or removed. If the user expects an icon representing how many items they have in their cart to be updated when they add a new item, that icon should actually update in real-time.

These examples all represent different data points that meaningful SLIs can give you—and these data points help you make decisions. If it's simply the case that your site isn't loading well or quickly enough, you might just need to introduce more resources. However, if the shopping cart isn't working well, the problem could be anything from the JavaScript powering user interactions to the service that talks to the database to the data-storage systems that are ultimately responsible for holding the ones and zeroes. By having the data that multiple meaningful SLIs provide you, you can make better decisions about what you should be measuring in the first place and what areas of your system require the most attention.

## Decisions about Tolerable Failures

One of the most attractive aspects of measuring services with SLOs is that the entire discipline acknowledges the fact that nothing is ever perfect. All complex systems fail at some point in time, and because of this fact it is fruitless to aim for 100%. Additionally, it turns out that people already know and are okay with this—whether they're consciously aware of it or not.

For example, if you start streaming a video via a video-streaming service, you have a certain expectation for how long it should take for such a video to buffer before it begins playing in real time. However, if it takes much longer than normal to buffer every once in a while, you likely won't care too much. Most people won't abandon a streaming video platform if one in every 100 videos takes 10 seconds of buffering time instead of three seconds. Failure in the sense that the streaming platform had to buffer too long occasionally is just fine—it just can't happen too often. If videos take 10 seconds to start every single time, people might become annoyed and look for other options.

A good service level objective lies somewhere just beyond what you need for users of your service to be happy. If people are okay with one in every 100 streaming attempts buffering for longer than normal, you should set your SLO target at something like one in every 200 streaming attempts. Exactly where you set this target is up to the data you have available to you and the feedback you're able to collect from your users. The important part is that your SLO should be more strict than the level at which users might decide to leave and use a different option. No matter how refined your SLO target is, you're not always going to reach it, and you don't want your business or organization to suffer if you don't.

Acknowledging failure and accounting for it are at the very base of how SLO-based approaches work. Understand that nothing is perfect, but use SLO data to help you decide how close to perfect you should attempt to be—or risk losing users.

## Decisions about Work Focus

Once you have a meaningful SLI and a reasonable SLO target, you can produce an error budget. Error budgets are simply just another data point you can use to make decisions. They're the most complicated part of the stack, but once you can find yourself using error budgets to drive your decision-making, you'll truly understand how the entire SLO-based approach works.

Error budgets are the ultimate decision-making tool once you've established SLIs and SLOs. By measuring how you've performed over a time window, you can drive large-scale decisions that could impact anything from the focus of your team for a single sprint to the focus of an entire company for a quarter.

For example, let's say you have a reasonable measurement of how reliable one particular microservice has been. Using your error budget, you can now also see that you haven't been reliable about 10% of the time over the last month. At this point you can use this data to inform a few different discussions that can fuel decisions.

One example is that you simply haven't been performing well enough, and that you believe that your SLI measurement and your SLO target are well-defined. If this is the case, you might choose to pivot one or more members of your team to focusing on reliability work instead of feature work. You could do this for anything, like the length of an on-call shift to a full sprint or even until you've recovered all of your budget. There are no hard-and-fast rules at play here. Error budgets, like everything else, are just data to help you decide what to do.

Another example is that you've completely depleted your error budget but have reason to think this exact situation is unlikely to arise again. An example of this kind of event could be anything like the disruption of power at a datacenter or just a historically bad bug pushed to production. Just because you've depleted your error budget over time doesn't mean you have to take action. Sometimes it absolutely makes sense to do so: perhaps you need to introduce a better testing infrastructure to your deployment pipeline, or maybe you need to install additional circuits or distribute your footprint geographically to avoid further power disruptions.

The point is that it's totally okay to look at how you've performed in terms of reliability over time and say, "This time we can just continue our current work focus." Error-budget statuses are just another data set you should use to make decisions—they shouldn't be rules that need to be followed every single time you examine your status. It doesn't matter if you're looking at the error budget status for a single small microservice that sees very little traffic or your entire service as viewed from your paying customers. Use error budgets as data to help you think about prioritization.

For a larger service, such as an entire customer-facing web service, burning through all of your error budget probably warrants some stricter decision-making. Even if it was due to your ISP that your users experienced an hour of outage last month, it still probably doesn't make sense for you to do things like perform potentially disruptive chaos engineering or experimentation in your production environment until a significant amount of time has passed. Be reasonable about how you make decisions using your error budgets, and certainly feel free to ignore their status from time to time—but never do so at the expense of your users' experience.

## Conclusion

There are many benefits to SLO-based approaches that I don't have room to cover here. They can help you better communicate to other teams about how they should think about the reliability of their own services. They can be excellent tools in reporting to management and product teams. They can also be used for many things outside of computer services, such as examining whether your team's ticket load is too high or whether people aren't taking enough vacation time. An SLO-based approach is simply about thinking about people and users first, acknowledging nothing is perfect, and using some math to help you aim for reasonable targets instead.

But one of the most important parts of this approach is that it allows you to make better data-driven decisions. Don't just implement SLOs because they're popular and a buzzword, or because you heard a conference talk about them, or because upper-management has decided that every team must have one.

Implement SLOs because they give you data you can use to have better discussions and make better decisions—decisions that can help make both your team and your users happier.

# ML for Operations
## Pitfalls, Dead Ends, and Hope

STEVEN ROSS AND TODD UNDERWOOD

Steven Ross is a Technical Lead in site reliability engineering for Google in Pittsburgh, and has worked on machine learning at Google since Pittsburgh Pattern Recognition was acquired by Google in 2011. Before that he worked as a Software Engineer for Dart Communications, Fishtail Design Automation, and then Pittsburgh Pattern Recognition until Google acquired it. Steven has a BS from Carnegie Mellon University (1999) and an MS in electrical and computer engineering from Northwestern University (2000). He is interested in mass-producing machine learning models. stross@google.com

Todd Underwood is a lead Machine Learning for Site Reliability Engineering Director at Google and is a Site Lead for Google's Pittsburgh office. ML SRE teams build and scale internal and external ML services and are critical to almost every product area at Google. Todd was in charge of operations, security, and peering for Renesys's Internet intelligence services that is now part of Oracle's cloud service. He also did research for some early social products that Renesys worked on. Before that Todd was Chief Technology Officer of Oso Grande, an independent Internet service provider (AS2901) in New Mexico. Todd has a BA in philosophy from Columbia University and a MS in computer science from the University of New Mexico. He is interested in how to make computers and people work much, much better together. tmu@goggle.com

Machine learning (ML) is often proposed as the solution to automate this unpleasant work. Many believe that ML will provide near-magical solutions to these problems. This article is for developers and systems engineers with production responsibilities who are lured by the siren song of magical operations that ML seems to sing. Assuming no prior detailed expertise in ML, we provide an overview of how ML works and doesn't, production considerations with using it, and an assessment of considerations for using ML to solve various operations problems.

Even in an age of cloud services, maintaining applications in production is full of hard and tedious work. This is unrewarding labor, or toil, that we collectively would like to automate. The worst of this toil is manual, repetitive, tactical, devoid of enduring value, and scales linearly as a service grows. Think of work such as manually building/testing/deploying binaries, configuring memory limits, and responding to false-positive pages. This toil takes time from activities that are more interesting and produce more enduring value, but it exists because it takes just enough human judgment that it is difficult to find simple, workable heuristics to replace those humans.

We will list a number of ideas that appear plausible but, in fact, are not workable.

## What Is ML?

Machine learning is the study of algorithms that learn from data. More specifically, ML is the study of algorithms that enable computer systems to solve some specific problem or perform some task by learning from known examples of data. Using ML requires training a model on data where each element in the data has variables of interest (features) specified for it. This training creates a model that can later be used to make inferences about new data. The generated model is a mathematical function, which determines the predicted value(s) ("dependent variable(s)") based on some input values ("independent variables"). How well the model's inferences fit the historical data is the objective function, generally a function of the difference between predictions and correct inferences for supervised models. In an iterative algorithm, the model parameters are adjusted incrementally on every iteration such that they (hopefully) decrease the objective function.

## Main Types of ML

In order to understand how we'll apply ML, it is useful to understand the main types of ML and how they are generally used. Here are broad categories:

### *Supervised Learning*

A supervised learning system is presented with example inputs and their desired outputs labeled by someone or a piece of software that knows the correct answer. The goal is to learn a mapping from inputs to outputs that also works well on new inputs. Supervised learning is the most popular form of ML in production. It generally works well if your data consist of a large volume (millions to trillions) of correctly labeled training examples. It can be effective with many fewer examples, depending on the specific application, but it most commonly does well with lots of input data.

Think of identifying fruit in an image. Given a set of pictures that either contain apples or oranges, humans do an amazing job of picking out the right label ("apple" or "orange") for the right object. But doing this without ML is actually quite challenging because the heuristics are not at all easy. Color won't work since some apples are green and some oranges are green. Shape won't work because it's hard to project at various angles, and some apples are exceedingly round. We could try to figure out the skin/texture but some oranges are smooth and some apples are bumpy.

With ML we simply train a model on a few hundred (or a few thousand) pictures labeled "orange" or "apple." The model builds up a set of combinations of features that predict whether the picture has an apple or an orange in it.

### *Unsupervised Learning*

The goal of unsupervised learning is to cluster pieces of data by some degree of "similarity" without making any particular opinion about what they are, i.e., what label applies to each cluster. So unsupervised learning draws inferences without labels, such as classifying patterns in the data.

One easy-to-understand use case is fraud detection. Unsupervised learning on a set of transactions can identify small clusters of outliers, where some combination of features (card-not-present, account creation time, amount, merchant, expense category, location, time of day) is unusual in some way.

Unsupervised learning is particularly useful as part of a broader strategy of ML, as we'll see below. In particular, in the example above, clustering outlier transactions isn't useful unless we do something with that information.

### *Semi-Supervised Learning*

The goal of semi-supervised learning is to discover characteristics of a data set when only a subset of the data is labeled. Human raters are generally very expensive and slow, so semi-supervised learning tries to use a hybrid of human-labeled data and automatically "guessed" labels based on those human labels. Heuristics are used to generate assumed labels for the data that isn't labeled, based on its relationship to the data that is labeled.

Semi-supervised learning is often used in conjunction with unsupervised learning and supervised learning to generate better results from less effort.

### *Reinforcement Learning*

In reinforcement learning (RL), software is configured to take actions in an environment or a simulation of an environment in order to accomplish some goal or cumulative set of values. The software is often competing with another system (which may be a prior copy of itself or might be a human) without externally provided labeled training data, following the rules.

Google's DeepMind division is well known for using RL to solve various real-world problems. Famously, this has included playing (and winning) against humans in the strategy game Go [1] as well as the video game StarCraft [2]. But it has also included such practical and important work as optimizing datacenter power utilization [3].

## ML for Operations: Why Is It Hard?

Given that ML facilitates clustering, categorization, and actions on data, it is enormously appealing as a system to automate operational tasks. ML offers the promise of replacing the human judgment still used in decisions, such as whether a particular new deployment works well enough to continue the roll-out, and whether a given alert is a false positive or foreshadowing a real outage. Several factors make this more difficult than one might think.

ML produces models that encode information by interpreting features in a fashion that is often difficult to explain and debug (especially with deep neural networks, a powerful ML technique). Errors in the training data, bugs in the ML algorithm implementation, or mismatches between the encoding of data between training and inference will often cause serious errors in the resulting predictions that are hard to debug. Below we summarize some common issues.

### *ML Makes Errors*

ML is probabilistic in nature, so it will not always be right. It can classify cats as dogs or even blueberry muffins [4] as dogs a small fraction of the time, especially if the data being analyzed is significantly different from any specific training example. Of course, humans make errors as well, but we are often better able to predict, tolerate, and understand the types of errors that humans make. Systems need to be designed so such occasional gross errors will be tolerable, which sometimes requires sanity tests on the result (especially for numerical predictions).

### *Large Problem Spaces Require Lots of Training Data*

The more possible combinations of feature values that a model needs to deal with, the more training data it requires to be accurate. In other words, where many factors could contribute to a particular labeling or clustering decision, more data is required. But in large feature spaces, there may be a large difference between examples being analyzed and the closest training data, leading to error caused by trying to generalize over a large space. This is one of the most serious issues with using ML in operations, as it is often hard to find sufficient correctly labeled training data, and there are often many relevant variables/features.

Specifically, the problem space of production engineering or operations is much messier than the space of fruit categorization.

In practice, it turns out to be quite difficult to get experts to categorize outages, SLO violations, and causes in a mutually consistent manner. Getting good labels is going to be quite difficult.

### *Training Data Is Biased Relative to Inference Demand*

The data you use to train your model may be too different from the data you're trying to cluster or categorize. If your training data only cover a particular subset of all things the model might need to infer over, all the other slices it wasn't trained on will see higher errors because of their divergence from the training data. Additionally, if the statistical distribution of classifications in the training data differs from the statistical distribution in the real world, the model will make skewed predictions, thinking that things that are more common in the training set are more common in the real world than they really are. For example, if the training data had 10 million dogs and 1000 cats, and dogs and cats are equally likely in the inference examples, it will tend to infer the presence of a dog more often than it should.

### *Lack of Explainability*

Many of the best performing ML systems make judgments that are opaque to their users. In other words, it is often difficult or impossible to know why, in human intelligible terms, an ML model made a particular decision with respect to an example. In some problem domains, this is absolutely not a difficulty. For example, if you have a large number of false positive alerts for a production system and you're simply trying to reduce that, it's not generally a concern to know that an ML model will use unexpected combinations of features to decide which alerts are real. For this specific application, as long as the model is accurate, it is useful.  But models with high accuracy due purely to correlation rather than causation do not support decision making. In other situations aspects of provable fairness and lack of bias are critical. Finally, sometimes customers or users are simply uncomfortable with systems that make decisions that cannot be explained to them.

## Potential Applications of ML to Operations

Given all of these challenges, it will be useful to examine several potential applications of ML to operations problems and consider which of these is feasible or even possible.

### *Monitoring*

For complex systems, the first problem of production maintenance is deciding which of many thousands of variables to monitor. Candidates might include RAM use by process, latency for particular operations, request rate from end users, timestamp of most recent build, storage usage by customer, number, and type of connections to other microservices, and so on. The possibilities of exactly what to monitor seem unbounded.

Systems and software engineering sometimes suggest using ML to identify the most relevant variables to monitor. The objective would be to correlate particular data series with the kinds of events that we are most interested in predicting—for example, outages, slowness, capacity shortfalls, or other problems.

In order to understand why this is a difficult problem, let us consider how to build an ML model to solve it. In order to use ML to create a dashboard that highlights the best metrics to see any current problems with your system, the best approach will be to treat the problem as a supervised multiclass prediction problem. To address that problem we will need:

- A class to predict for every metric of interest
- Labels for all classes that were helpful for millions of production events of concern
- Training and periodic retraining of your model as you fix bugs and create new ones with failure types shifting over time
- Periodic (potentially on page load) inferring with the model over which metrics should be shown to the user.

There are other complexities, but the biggest issue here is that you need millions of labeled training examples of production events of concern. Without millions of properly categorized examples, simple heuristics, for example that operators select the metrics that appear to be the most relevant, are likely to be as or more effective and at a fraction of the cost to develop and maintain. Simple heuristics also have several advantages over ML, as previously mentioned. We hope you don't have millions of serious problematic events to your production infrastructure to train over. However, if your infrastructure is of a scale and complexity that you think that you will, eventually, have an appropriate amount of data for this kind of application, you should begin accumulating and structuring that data now.

### *Alerting*

Most production systems have some kind of manually configured but automated alerting system. The objective of these systems is to alert a human if and only if there is something wrong with the system that cannot be automatically mitigated by the system itself.

The general idea of an ML-centric approach to alerting is that once you have determined which time series of data are worth *monitoring* (see above) it might be possible to automatically and semi-continuously correlate values and combinations of these. To accomplish this we can start with every alert that we have or could easily have and create a class for each.

We then need to create positive and negative labels. Positive labels are applied to the alerts that were both useful and predictive of some serious problem in the system that actually required human intervention. Negative labels are the opposite: either not useful or not predictive of required human intervention. We need to label many events, those where something bad was happening and those where everything was fine, and continuously add new training examples. To scope the effort, we estimate that we will need at least tens of thousands of positive examples and probably even more (millions, most likely) of negative examples in order to have a pure-ML solution that is able to differentiate real problems from noise more effectively than a simple heuristic. We are not discussing potential hybrid heuristic + ML solutions here since, in many practical setups, this will lead to increased complexity from integrating two systems that need to be kept in sync for the intended outcome, which is unlikely to be worth the extra effort.

Even if we had all these labels (and they're correct) and a good model, which we know to be difficult from the monitoring case above, the on-call will still need to know where to look for the problem. While we may be able to correlate anomalous metrics with a confident alerting signal, covering the majority of alert explanations this way would not be enough. For as long as the fraction of "unexplainable" alerts is perceived by alert recipients as high, the explainability problem makes adoption cumbersome at best. This is the problem of explainability.

### *Canarying/Validation*

Pushing new software to production frequently or continuously as soon as it is effectively tested poses risks that new software will sometimes be broken in ways the tests won't catch. The standard mitigation for this is to use a canary process that gradually rolls out to production combined with monitoring for problems and a rapid rollback if problems are detected. The problem is that monitoring is incomplete, so occasionally bad pushes slip through the canary process unnoticed and cause serious issues.

For this reason, production engineers often suggest using ML to automatically detect bad pushes and alert and/or roll them back.

This is a specialized version of the alerting problem; you need positive labels and negative labels, labeling successful pushes with positive labels and broken pushes with negative labels. Much like with alerting, you will probably need thousands of examples of bad pushes and hundreds of thousands of examples of good pushes to differentiate real problems from noise better than a simple heuristic. The main factor that makes canarying a little less hard than general alerting is that you have a strong signal of a high-risk event when your canary starts (as opposed to continuous monitoring for general alerting) and an obvious mitigation step (roll back), but you still need a large number of correctly labeled examples to do better than heuristics. Note that if you have a bad push that you didn't notice in your labeling, because it was rolled back too fast or got blocked by something else and improperly labeled as a good push, it will mess up your data and confuse your ML model.

False-positive canary failures will halt your release (which is usually a preferable outcome to an outage). To maintain release velocity, these need to be kept to a minimum, but that will lower the sensitivity of your model.

### *Root Cause Analysis*

Outages are difficult to troubleshoot because there are a huge number of possible root causes. Experienced engineers tend to be much faster than inexperienced engineers, showing that there is some knowledge that can be learned.

Production engineers would like to use ML to identify the most likely causes and surface information about them in an ordered fashion to the people debugging problems so that they can concentrate on what is likely. This would require classifying the set of most likely causes, and then labeling and training over enough data to rank this list of causes appropriately.

Because you need a fixed list of classes to train over for this problem, if a new type of problem shows up your model won't be able to predict it until it has trained over enough new examples. If you have a case that isn't on your list, then people may spend excessive time looking through the examples recommended by the model even though they're irrelevant. To minimize this risk, you might want to add lots of classes to handle every different possibility you can think of, but this makes the training problem harder as you need more properly labeled training data for every class of problem you want the model to be able to predict. To be able to differentiate between a list of a hundred causes, you'll probably need tens of thousands of properly labeled training examples. It will be difficult to label these examples with the correct root cause(s) without a huge number of incidents, and there is a strong risk that some of the manually determined root cause labels will be incorrect due to the complexity, making the model inaccurate. An additional complexity is that events (potential causes) sequenced in one order may be harmless (capacity taken down for updates after load has dropped), but sequenced in another order may cause a serious outage (capacity taken down for updates during peak load), and the importance of this sequencing may confuse the ML model.

A manually assembled dashboard with a list of the top *N* most common root causes, how to determine them (some of which might be automated heuristics), and related monitoring will probably be more helpful than an ML model for root cause analysis in most production systems today.

## Path Forward

We do not recommend that most organizations use machine learning to manage production operations at this point in the maturity of software services and ML itself. Most systems are not large enough and would do better to focus their engineering effort and compute resources on more straightforward means of improving production operations or expanding the business by improving the product itself. Unless all of your monitoring is well curated, alerting is carefully tuned, new code releases thoroughly tested, and rollouts carefully and correctly canaried, there is no need to expend the effort on ML.

However, in the future as production deployments scale, data collection becomes easier, and ML pipelines are increasingly automated, ML will definitely be useful to a larger fraction of system operators. Here are some ways to get ready:

1. Collect your data. Figure out what data you think you might use to run production infrastructure and collect it.
2. Curate those data. Make sure that the data are part of a system that separates and, where possible, labels the data.
3. Begin to experiment with ML. Identify models that might make sense and, with the full understanding that they will not reach production any time soon, begin the process of prototyping.

## Conclusion

While ML is promising for many applications, it is difficult to apply to operations today because it makes errors, it requires a large amount of high-quality training data that is hard to obtain and label correctly, and it's hard to explain the reasons behind its decisions. We've identified some areas where people commonly think ML can help in operations and what makes it difficult to use in those applications. We recommend using standard tools to improve operations first before moving forward with ML, and we suggest collecting and curating your training data as the first step to take before using ML in operations.

***References***

[1] https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[2] https://www.seas.upenn.edu/~cis520/papers/RL_for_starcraft.pdf.

[3] https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf.

[4] https://www.topbots.com/chihuahua-muffin-searching-best-computer-vision-api/.

COLUMNS

# Site Reliability Engineering and the Crisis/Complacency Cycle

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly Site Reliability Engineering book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura is a production engineer at Slack. laura.nolan@gmail.com

This column will be published in Summer 2020, but I'm writing it in mid-March. In the past week, in a response to the spread of the new SARS-CoV-2 virus, many nations have closed down schools and imposed restrictions on travel and events. Several major technology companies are encouraging most employees to work from home. Stock markets are falling more quickly than in the first stages of the 2008 crash. Nothing is normal.

My social media feeds clearly show that SARS-CoV-2 is a source of fascination for systems engineers and SREs (site reliability engineers) because it has some characteristics of the kinds of systems problems we deal with in our work. The pandemic response is currently centered around preventing the spread of the infection, effectively an attempt to throttle admissions to intensive care in order to avoid saturating scarce medical resources. It involves gathering metrics (which are lagging and sparse due to shortage of test kits) to make analyses and projections. The mathematical analysis of the spread of the illness is very similar to the characteristics of information propagation in a dissemination gossip protocol [1], which will be familiar to anyone who has worked with Cassandra, Riak, Consul, or even BitTorrent—the major difference being that instead of modifying software parameters to adjust the propagation, we all now need to reduce our social interactions, and perhaps to partition our systems with travel restrictions.

I am not an epidemiologist, and I can't predict how this situation will unfold between now and when you read these words. Will we have endured on an international scale the kind of health crisis that northern Italy is experiencing in March, or will most nations succeed in averting the worst consequences of the pandemic, as South Korea seems to have done? If we do succeed, it's possible that many will consider the robust response to the outbreak to be an overreaction, even in light of the evidence from northern Italy and Wuhan that failure to control outbreaks leads to major public health problems.

## The Job Is to Get Ahead of Problems

There is a phenomenon in operations, which I've heard called the "paradox of preparation"—an organization that is effectively managing risks and preventing problems can fail to be recognized as such. Bad outcomes aren't actually occurring, because of this preventative work, so decision-makers may come to believe that the risks are significantly lower than they actually are. Therefore, leaders may conclude that the organization that is preventing the negative events from occurring isn't an efficient use of resources anymore.

This appears to have been the fate of the White House's National Security Council Directorate for Global Health Security and Biodefense, which was set up in 2014 in response to the Ebola outbreaks in Western Africa, then shut down abruptly in 2018. It was tasked with monitoring emerging disease risks and coordinating responses and preparation. According to its former head, Beth Cameron, "The job of a White House pandemics office would have been to get ahead: to accelerate the response, empower experts, anticipate failures, and act quickly and transparently to solve problems" [2]. That is a function very much akin to what a good SRE or resilience engineering team can do within a software engineering organization.

In 2019, before the SARS-CoV-2 virus appeared, the Center for Strategic and International Studies think tank drew attention to the closure of the Directorate.

> When health crises strike—measles, MERS, Zika, dengue, Ebola, pandemic flu—and the American people grow alarmed, the U.S. government springs into action. But all too often, when the crisis fades and fear subsides, urgency morphs into complacency. Investments dry up, attention shifts, and a false sense of security takes hold. The CSIS Commission on Strengthening America's Health Security urges the U.S. government to replace the cycle of crisis and complacency that has long plagued health security preparedness with a doctrine of continuous prevention, protection, and resilience. [3]

This cycle of crisis and complacency is one we see in other kinds of organizations, including software companies—a view that reliability is only worth investing in the wake of problems, and at other times it may be deprioritized and destaffed. The last edition of this column discussed Professor Nancy Leveson's model of operations as a sociotechnical system dedicated to establishing controls over production systems in order to keep them within predefined safety constraints [4]. The crisis/complacency cycle makes it impossible to build a strong sociotechnical system that proactively manages change and emerging risks, because it means that when investment into reliability happens you have to build expertise, standards, processes, and organizations from scratch while already in crisis mode.

## Against the Advice of Their Own Experts

This crisis/complacency cycle is not new, nor is it unique to either software or to pandemic prevention. The Boeing 737 Max has been in the news for most of the past year following two fatal crashes which were the consequence of design flaws in the new aircraft type. The entire 737 Max fleet was grounded in response to the accidents.

The airplane's design was certified by the US Federal Aviation Administration (FAA), a body created in 1958 to manage all aspects of safety in aviation. Air travel has become safer every decade since the FAA was set up, driven by improvements in technology and safety culture. Perhaps not coincidentally, the FAA has come under significant budgetary pressure in recent years. Partly as a result of those budgetary constraints and partly because of a shortage of relevant technical expertise, the FAA delegated much of the technical work of validating the design of the 737 Max aircraft against FAA standards to Boeing itself.

The report of the House Committee on Transportation and Infrastructure paints a clear picture of enormous pressure from Boeing's management to get the aircraft to the market as quickly as possible, at the lowest feasible cost and without any need for existing 737 pilots to take further training—regardless of any safety concerns [5]. Budgets for testing were cut, and multiple suggestions by engineers to incorporate extra alerts and indicators were rejected. Though it isn't in Boeing's commercial interest to develop an unsafe aircraft, the company's management consistently made decisions that compromised safety, contrary to the advice of their own technical experts. That they did this against the backdrop of the safest period in the history of commercial flight strongly suggests the same cycle of crisis and complacency was at work in Boeing and the FAA that led to the shutdown of the White House's pandemics office in 2018.

## Disconnects between Management and Engineers

On January 28, 1986, the Space Shuttle Challenger exploded during liftoff. The accident was triggered by the failure of an O-ring seal in unusually cold weather conditions. The disaster occurred after 24 successful space shuttle launches, and these successes helped to create complacency about safety at NASA. The incident has been studied extensively, most notably by Diane Vaughan, who coined the term "normalization of deviance" to describe the process by which previously unacceptable results and practices can gradually become the norm over time [6]. Despite that phenomenon, the Rogers Commission Report on the disaster found that engineers had raised safety concerns over the design with management.

Richard Feynman, the noted physicist, was a member of the commission that investigated the Challenger accident. Feynman was particularly struck by the difference in perception of risk between the engineers who worked on the shuttle and NASA's management. The engineers mostly believed that the shuttle had a risk of catastrophic failure between 1 in 50 and 1 in 200. NASA's management claimed that the risk was 1 in 100,000. Feynman's assessment was that the engineers' estimate of the risk was far closer to the truth than management's number, which seemed based largely on wishful thinking and misunderstandings [7].

This kind of disconnect seems also to have existed at Boeing in recent years. In 2001, Boeing's executives moved from Seattle, where its engineers are located, to Chicago, and non-engineers moved into many executive roles.

> [T]he ability [was lost] to comfortably interact with an engineer who in turn feels comfortable telling you their reservations, versus calling a manager [more than] 1,500 miles away who you know has a reputation for wanting to take your pension away. It's a very different dynamic. As a recipe for disempowering engineers in particular, you couldn't come up with a better format. [8]

## "Captain Hindsight Suited Up": Outcome Bias

Many of us in the software industry still remember the cautionary tale of Knight Capital, a financial firm that went bust in 2012 as a result of a bug in their trading software. As Knight Capital was an SEC (Securities and Exchange Commission) regulated company, there was an investigation and a report, which recommended that the company should have halted trading as soon as they realized there was something amiss [9].

On July 9, 2015, the New York Stock Exchange discovered a problem in their systems. They halted trading, just as the SEC said that Knight Capital ought to have done. However, as John Allspaw put it, the "clone army of Captain Hindsights suited up, ready to go" decided that the shutdown hadn't been essential and criticized the NYSE for unnecessarily halting over a "glitch" [10].

This is outcome bias, a cognitive bias that leads us to judge decisions based on their results. We can't predict the consequences of decisions perfectly at the time we make them. Many tough decisions have to be made with imperfect information—risks we can't fully quantify, information that's incomplete or missing. Sometimes, you need to make a sacrifice decision to avoid a risk of greater harm. This is likely better than simply reacting according to prevailing conditions of the crisis/complacency cycle. This closely describes the situation that the political leaders of most of the world find themselves in March 2020 with respect to SARS-CoV-2. By the time you read this, outcome bias will likely have declared their actions as overkill (if successful) or insufficient.

## Risk and Rot in Sociotechnical Systems

We work in organizations made up of people, all subject to outcome bias and prone to underestimate or overestimate risks, depending on to what extent normalization of deviance has set in on our team. Executives can become far removed from the reality of life at the front line, and their appreciation of probabilities of adverse events can be strongly affected by recent outcomes.

One of the major functions of an SRE or operations team is to proactively manage risks. This kind of work covers a broad spectrum, from keeping systems patched, rotating certs and tokens, and validating backups, through to less routine things like writing runbooks and recovery tools, running disaster tests, performing production readiness reviews for new systems, and doing thorough reviews of near-miss production incidents. These are also the kinds of work that can fall by the wayside all too easily when a team is overloaded or understaffed. The eventual outcome is likely to be a crisis and the start of a new cycle of investment.

An important part of our job, therefore, is to make the value of our work visible in order to avoid the organizational rot that makes us underestimate risk and underinvest in reliability. We live in a data-driven world, but of course, we can't track the incidents that don't happen because of good preventative work. However, at times when we aren't in crisis mode, there are many other things that we can do to show how our work contributes to increasing reliability.

We can create internal SLOs for the routine jobs we do to manage risks, and set up dashboards to show whether you're meeting those SLOs or not. Write production-readiness standards that you'd like your services to meet—covering areas such as change management, monitoring and alerting, load balancing and request management, failover, and capacity planning. Track how your services meet those standards (or don't). Set up chaos engineering and game days to test how your services deal with failure, and track those results as you would postmortem action items. Load test your systems to understand how they scale, and address bottlenecks you will encounter in the next year or two. Take near-misses and surprises seriously, and track them, along with action items. All of these things help to prevent a slide into normalization of deviance as well as giving visibility into our work.

As engineers, we have a responsibility to clearly communicate about risks in our systems and the proactive work we do to reduce them. But "the fish rots from the head down": engineering leaders ultimately make critical decisions and therefore they must be acutely aware of outcome bias and the risk of disconnects in understanding of risk between front-line engineers and themselves. Most importantly, they must be mindful of the crisis/complacency cycle and maintain an appropriate continuous investment in resilience and reliability in order to avoid crisis.

***References***
[1] K. Birman, "The Promise, and Limitations, of Gossip Protocols": http://www.cs.cornell.edu/Projects/Quicksilver/public_pdfs/2007PromiseAndLimitations.pdf.

[2] B. Cameron, "I ran the White House pandemic office. Trump closed it," *The Washington Post,* March 13, 2020.

[3] J. S. Morrison, K. Ayotte, and J. Gerberding, "Ending the Cycle of Crisis and Complacency in U.S. Global Health Security," November 20, 2019, Center for Strategic International Studies: https://www.csis.org/analysis/ending-cycle-crisis-and-complacency-us-global-health-security.

[4] L. Nolan, "Constraints and Controls: The Sociotechnical Model of Site Reliability Engineering," *;login:*, vol. 45, no. 1 (Spring 2020), pp. 44–48.

[5] The House Committee on Transportation and Infrastructure, "Boeing 737 MAX Aircraft: Costs, Consequences, and Lessons from Its Design, Development, and Certification," March 2020: https://transportation.house.gov/imo/media/doc/TI%20Preliminary%20Investigative%20Findings%20Boeing%20737%20MAX%20March%202020.pdf.

[6] D. Vaughan, *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA* (University of Chicago Press, 1996).

[7] Presidential Commission on the Space Shuttle Challenger Accident, Report, 1986: https://science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/table-of-contents.html.

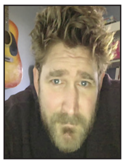
[8] J. Useem, "The Long-Forgotten Flight That Sent Boeing Off Course," *The Atlantic*, November 20, 2019: https://www.theatlantic.com/ideas/archive/2019/11/how-boeing-lost-its-bearings/602188/.

[9] "In the Matter of Knight Capital Americas LLC," SEC Release No. 70694, October 16, 2013: https://www.sec.gov/litigation/admin/2013/34-70694.pdf.

[10] J. Allspaw, "Hindsight and Sacrifice Decisions," March 3, 2019: https://www.adaptivecapacitylabs.com/blog/2019/03/03/hindsight-and-sacrifice-decisions/.

# iVoyeur
## eBPF Tools

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Fastly. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

I spent my high-school years in a tightly entangled group of four friends. We were basically inseparable, formed a horrible rock band, and I think did a lot of typical '90s Los Angeles kid things like throwing powdered doughnuts into oncoming traffic and making a nuisance of ourselves at 7-Eleven and Guitar Center. We smashed against the breakwater of graduation and went different places, but of the four of us, I was the only one who didn't go off to college to study music theory. Opting instead to eject into the Marine Corps, which is a longer story, and irrelevant to the current metaphor.

Anyway, we kept in touch, and in their letters all three of my friends described the process of learning music theory in a very similar way. As a neophyte musician, you typically have some aptitude with one or two instruments, but very little knowledge about how music itself works. Evidently in the first year of music theory, you are presented with myriad complicated rules. From what I understand, in fact, you do little else the first year but learn the rules and some important exceptions to the rules.

Then bit by bit, as the years progress, the rules are stripped away, until you reach some sort of musical enlightenment, where there are no rules and you work in a kind of effortless innovatory fugue where everything you create just clicks.

I vaguely remember feeling this way about computer science. Having written my first Perl script, flush with optimism and newfound aptitude. "So this is what it feels like to have mastered computering at last," I thought to myself, setting aside my Camel book to cross my arms in a self-satisfied way, and cursing whatever company I was working for at the time with whatever abomination I'd just created.

Many—er, well, several years later, I feel strongly that computer science is something like the exact opposite of how my friends described music theory in those hastily scribbled letters all that time ago. The rules do not so much disappear but rather change and reassemble anew every so often, and instead of effortless enlightenment, I find myself splitting my days between confounded frustration and shocked dismay, each of those punctuated by short bouts of relief and semi-comprehension. In our world I sometimes feel like it's a miracle anything works at all, and the more I learn, the less I seem to know.

In my last article I introduced eBPF, the extended Berkeley Packet Filter, along with a shell tool called `biolatency`, which uses eBPF-based kernel probes to instrument the block I/O (or bio) layer of the kernel and return per-device latency data in the form of a histogram. There is a deeply refreshing crispness about delving into the solar system of eBPF, a brisk undercurrent that pulls one down through abstraction layers and toward the metal. There are over 150 tools in the BCC (https://github.com/iovisor/bcc) tools suite, and you can use them all without knowing how they work, of course. I think you'll find, however, that your effectiveness with BCC tools like `biolatency` scales linearly with your knowledge of kernel internals, and the slightest exploration into their inner workings leads one directly into the kernel source.

Let's begin this second article on eBPF, therefore, with a short discussion of the Linux Kernel's "bio" layer [1]. This is the kernel software layer loosely defined as the contents of the `block` subdirectory of the Linux kernel source. The code here resides between file systems like ext3 and device drivers, which do the work of interacting directly with storage hardware.

At this layer, we are below abstraction notions like files and directories. Disks are represented by a small `struct` inside the kernel called `struct_gendisk` [2], for "generic disk," and reads and writes no longer exist as separate entities. Instead, all types of block I/O operations are wrapped inside a generic request wrapper called `struct_bio` [2], the `struct` for which the "bio" layer is named.

Without delving any further into the bio layer, we can already see how ideally situated the bio layer is for trace-style instrumentation. Above us, in the file systems, we would need to probe every kind of disk operation: a different probe for reads, writes, opens, etc. Below bio we will find vendor-specific code and a mountain of historical, related exceptions and complications. But right here inside bio, we have a single, well-defined data-structure that represents every type of disk I/O possible operation. No writes, no reads, just requests, and one probe can summarize them all.

We can also assume that tracing these requests will give us read access to the `struct_bio` data structure, because we'll need it to see what kind of request we're dealing with (e.g., read/write), what block device each request is destined for, and so on.

We now have the necessary information to take our first cursory glance inside `biolatency.py` [3] to intuit what's going on. The first 53 lines are pretty typical preamble for a Python script: documentation, imports, and argument parsing. The arguments are interesting, but we'll set them aside for now to take a look at the large string that begins on line 55:

```
# define BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
typedef struct disk_key {
```

From our last article you'll remember that eBPF is a virtual machine that resides in the kernel. This string (named `bpf_text`) is the payload intended for that in-kernel VM; it takes up about a quarter of the overall code in the Python script and is written in C. It is a program, embedded within our program, that will be compiled to bytecode and loaded into the kernel's eBPF VM. If you look closely, you'll notice that this C code won't compile as is, because of expressions like this one on line 70:

```
BPF_HASH(start, struct request *);
STORAGE
```

These are string-replacement match targets. These will be replaced in this string with valid code, depending on options passed in by the user. These substitutions begin on line 103 and all take the same general form:

```
if args.milliseconds:
    bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000000;')
    label = "msecs"
else:
    bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000;')
    label = "usecs"
```

All of these substring substitutions follow the same basic pattern: if option X was set by the user, then replace MACRO in the payload program with value Y; otherwise, replace MACRO with value Z. In the example above, we're choosing between micro and milliseconds in the payload string. We're also setting a "label" variable to give hints for properly printing the output later on. This process of rewriting sections of the payload string goes on for most of the options the user passes in. The exception is `-Q`, which selects whether we will include the time an I/O request spends queued in the kernel as part of the latency calculation.

This switch affects our choice of which particular kernel functions we ultimately choose to trace. If we don't care about queue-time, we will want to measure latency starting from the moment the I/O request is *issued*. However if `-Q` is set, we will also want to include the time each request spent waiting on the kernel. We can see how this is implemented starting on line 134:

```
b = BPF(text=bpf_text)
if args.queued:
    b.attach_kprobe(event="blk_account_io_start", \
        fn_name="trace_req_start")
else:
    if BPF.get_kprobe_functions(b'blk_start_request'):
        b.attach_kprobe(event="blk_start_request", \
          fn_name="trace_req_start")
    b.attach_kprobe(event="blk_mq_start_request", \
        fn_name="trace_req_start")
b.attach_kprobe(event="blk_account_io_done",
        fn_name="trace_req_done")
```

First, we instantiate a new BPF Python object, passing in our newly rewritten payload in the process. What happens next depends on the `-Q` option. If we care about the latency induced by in-kernel queue time, then we'll insert our kernel probe at the `blk_account_io_start()` kernel function, which is called when an I/O request is first queued in the kernel. However, if we want to measure "pure" block I/O latency—that is, the amount of time a given generic I/O request took to return—we'll instrument `blk_mq_start_request()` and possibly `blk_start_request()` if the latter function exists in the current kernel. No matter what paths we choose, we'll close each trace at `blk_account_io_done()`.

At this point, our payload is inserted into the running kernel, and we are collecting data. Now we are confronted with some bitwise arithmetic beginning with a collection of constants on line 147 and continuing with some bitmask construction, and constants definition on line 157:

```
REQ_OP_BITS = 8
REQ_OP_MASK = ((1 << REQ_OP_BITS) - 1)
REQ_SYNC = 1 << (REQ_OP_BITS + 3)
REQ_META = 1 << (REQ_OP_BITS + 4)
REQ_PRIO = 1 << (REQ_OP_BITS + 5)
```

This is necessary to understand the data we're collecting. The bit-specifics correspond to the `bi_opf` [4] attribute (bio operational flags) inside `struct_bio`, the central block I/O request struct I mentioned above in the bio layer. The attribute is an unsigned `int` that's used to track metadata about a given block I/O request. You can see the constant defs for this bitmask a few lines down [5] from the `struct_bio` definition in the kernel source. In short, these flags tell us whether a given request was a read, write, cache-flush, etc. and provide some additional metadata about the operation, whether it was priority, backgrounded, read-ahead, etc.

If you continue down to line 171 in biolatency, you'll see that we `AND` the flags value, given to us from the probe, against a bitmask with bit 7 set to determine an integer value that corresponds to the top-level operation type (read, write, flush, discard, etc.). We then proceed to individually check for flags in the bitmask which correspond to subcategories. Prepending these to the top-level operation type:

```
if flags & REQ_SYNC:
    desc = "Sync-" + desc
if flags & REQ_META:
    desc = "Metadata-" + desc
if flags & REQ_FUA:
    desc = "ForcedUnitAccess-" + desc
```

So if the flags mask `AND`'d to a value of 0, which equates to "Read," and then we subsequently discovered that bit 11 was set in the flags mask corresponding to "Sync," we'd wind up filing this bio-request under "Sync-Read." Biolatency can use this data to plot histograms of I/O latency *per category of I/O operation* with the `-F` flag.

The last section in the script deals with printing our output. The script stays in the foreground until it encounters a keyboard interrupt from the user, and then dumps its output depending on how the user specified they wanted to see it in the argument flags. Unfortunately, these all use functions defined deeper inside the BCC Python library code, and scratching at them requires us to understand the eBPF data model, and a little bit more about the line between kernel and userspace, all of which we will get into in our next article.

If you're feeling like you know less than you did when you came in, then you are in a pretty good place. As I said in the intro, studying eBPF internals brings you close to the kernel in short order, which is a refreshing place to be. If you'd like to read a little more about the kernel's bio layer, there is an excellent set of introductory articles at LWN [1], and Brendan Gregg's BCC Python Development Tutorials [6] are another great resource for those wanting to read ahead.

Take it easy.

## *References*

[1] "A block layer introduction part 1: The bio layer": https://lwn.net/Articles/736534/.

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/genhd.h?h=v4.13#n171.

[3] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py.

[4] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/blk_types.h?h=v4.14-rc1#n54.

[5] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/blk_types.h?h=v4.14-rc1#n182.

[6] https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md.

# Who Will Pay the Piper for Open Source Software Maintenance?

## Can We Increase Reliability as We Increase Reliance?

DAN GEER AND GEORGE P. SIENIAWSKI

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

George P. Sieniawski is a technologist at In-Q-Tel Labs, which develops open source tools and data sets that address challenges at the intersection of national security, the public interest, and the private sector. He specializes in data visualization research and prototype development for a wide variety of use cases. GSieniawski@iqt.org

"A little neglect may breed mischief ...
for want of a nail, the shoe was lost;
for want of a shoe, the horse was lost;
and for want of a horse, the rider was lost."

—*Benjamin Franklin, Poor Richard's Almanac (1758)*

As software eats the world and open source eats software, IT supply chains and enterprise risk management postures are evolving. Top-down, CIO-led commercial software procurement is shifting towards bottom-up, developer-driven choices that increasingly involve open source software (OSS) [1]. Security in this context requires visibility, starting with a comprehensive inventory (software bill of materials) as well as an understanding of code provenance (software composition analysis). It also entails application testing, automated vulnerability scanning, instrumentation, and observability, which can provide insights for defenders. For organizations that plan over longer time horizons, however, mitigating OSS risk sometimes means taking on direct responsibility for software maintenance. Little by little, organizations are empowering staff to perform upstream code improvements that the rest of the world can access. When implemented thoughtfully, this pragmatic form of software stewardship can help avoid broken builds, obsolescence, and other potential failure modes.

In a rough count by the authors, we found that at least one-third of Fortune 500 firms have a public Git presence for company-sanctioned OSS activity [2]. While proprietary software use remains widespread, and while many more companies use private repositories for internal collaboration projects, or inner-source, many high-profile enterprise software development efforts are now happening in the open under permissive license terms. A similar pattern appears to be unfolding in the public sector, albeit at a more gradual pace. NASA, the GSA, the Department of Transportation, and the Department of Energy, for instance, have earned high marks on the code.gov agency compliance dashboard for their performance under the Federal Source Code Policy. Other federal agencies are taking more incremental steps in adapting OSS to their missions, and these initiatives are likely to remain a continual work-in-progress. With commercial and governmental enterprises mostly consuming but increasingly producing OSS, and with shared source code resources circulating across both types of Git repos, knowledge spillovers [3] appear to be reshaping a wide variety of software development communities. Silicon Valley is playing a prominent role in this arena, and as the Linux Foundation's Core Infrastructure Initiative recently noted, "some of the most active OSS developers contribute to projects under their Microsoft, Google, IBM, or Intel employee email addresses" [4].

Whether public or private, funding for OSS can help underwrite open innovation, reduce security costs, and amortize technical debt, but Red Hat's Gordon Haff reminds us: "Open source today is not peace, love, and Linux" [5]. Fiscal sponsorship can skew incentives in

| Top 50 Packages (for each package manager) | Primary Language | Language Rank,* 2019 | Language Rank,* 2018 | Average Dependent Projects | Average Direct Contributors |
|---|---|---|---|---|---|
| npm | JS | 1 | 1 | 3,500,000 | 35 |
| Pip | Python | 2 | 3 | 78,000 | 204 |
| Maven | Java | 3 | 2 | 167,000 | 99 |
| NuGet | .NET/C++ | 6 | 5 | 94,000 | 109 |
| RubyGems | Ruby | 10 | 10 | 737,000 | 146 |

**Table 1:** Concentration of GitHub contributions. *Popularity ranked by number of unique contributors to public and private GitHub repositories tagged with the corresponding primary language. Source: GitHub, *State of the Octoverse* (https://octoverse.github.com/#average-package-contributors-and-dependencies), released Nov. 6, 2019 (a few months before GitHub acquired npm).

unexpected ways since OSS backers are in a position to influence feature prioritization and project governance. As organizations start treating user-driven open source development as a regular operating expense, some developers worry about ecosystem fragmentation, value capture, and selective appropriation of benefits. Indeed, the advent of new software funding vehicles and managed open source subscription plans has drawn comparisons to gentrification and gerrymandering [6]. Consequently, organizations looking to engage with OSS communities around the world need to understand developer motivations, which are distinct from ownership and contract [7] and which involve a mix of pecuniary, reputational, and "own-use"/DIY reasons.

As Internet researcher Nadia Eghbal rightly recognizes, the OSS community's "volunteer culture discourages talk of money" [8]. Moreover, "The pervasive belief, even among stakeholders such as software companies, that open source is well-funded, makes it harder to generate support" for fledgling projects. It also highlights the need to find a balance between bearing private cost and conferring public benefit, which is the crux of open source stewardship. In the years since Eghbal's magisterial study of OSS, developers have become increasingly vocal about funding. Researchers are also beginning to look more closely at the individual contributors whose work underpins today's OSS ecosystem. These efforts have started to shed light on the complex symbiosis—or perhaps commensalism—between community-developed OSS and corporate-backed OSS.

Among other companies, Netflix, JP Morgan, and Airbnb have reaped significant benefits from company-sponsored community-maintained open source, not only in terms of demonstrating technical prowess and cultivating talent, but also in terms of operational impact. Other groups, like the world's largest automakers collaborating on Automotive Grade Linux or the financial sector companies embracing the Hyperledger project, seem to be following suit by forming consortia. GitLab's effort to establish a clear set of principles that enable a diverse OSS contributor community to work as one is another compelling case in point. The company's management promises not to "remove features from the open source codebase in order to make the same feature paid." GitLab also stresses contributors' right to the integrity of their work: "If the wider community contributes a new feature they get to choose if it is open source or source-available (proprietary and paid)" [9]. By explicitly recognizing the value volunteer developers bring to the platform, the company has been able to promote high-quality code contributions while avoiding cannibalization.

GitLab's rivals also appear to be taking a long-term view of OSS risk [10]. In February 2019, Microsoft took a snapshot of the top active public GitHub repositories, depositing physical copies of some of the world's most widely used software in a decommissioned coal mine in the Svalbard archipelago of Norway. The company has already stored copies of the source code for the Linux and Android operating systems in this remote region, along with 6,000 other OSS libraries it considers significant. Part gene bank and part library, this mega-repository is now the largest tenant in the Arctic World Archive, with additional redundancies planned for other locations. Backing up this treasure trove of software is a significant resilience and data loss prevention measure. However, building a nest for Coase's Penguin [11] in Svalbard is by no means sufficient for the vitality of the open source economy. On the contrary, as OSS becomes ever more ubiquitous, active maintenance becomes an increasingly pressing priority. Which brings us to the maintainers.

## OSS Maintenance

Although there is "a high correlation between being employed and being a top contributor to" OSS [12], sustaining it takes more than a regular income stream. Long-term commitment to open source stewardship is also essential, as is budgeting time for periodic upkeep. For perspective, consider that 36% of professional developers report never contributing to open source projects, with another 28% reporting less than one open source contribution per year (2019 Stack Overflow Developer Survey). Thus, despite more direct enterprise engagement with open source, risk-averse attitudes towards licensing risk and potential loss of proprietary advantage endure by and large. Consider further Table 1, which shows how concentrated contribution patterns are, particularly in JavaScript, and thus where additional OSS maintenance support could have an outsized impact.

For additional context, Figures 1 and 2 show the geographic and technological mix of contemporary OSS development worldwide. Note that this is not an exhaustive account of OSS growth, merely an indicative snapshot at a single point in time. In addition, keep in mind that this data, sourced from the Open Source Compass, excludes GitHub projects with fewer than 10 watchers. For more detail on these smaller open source projects, which are enjoying intense growth outside of the US, see the *State of the Octoverse* report mentioned in the caption of Table 1.

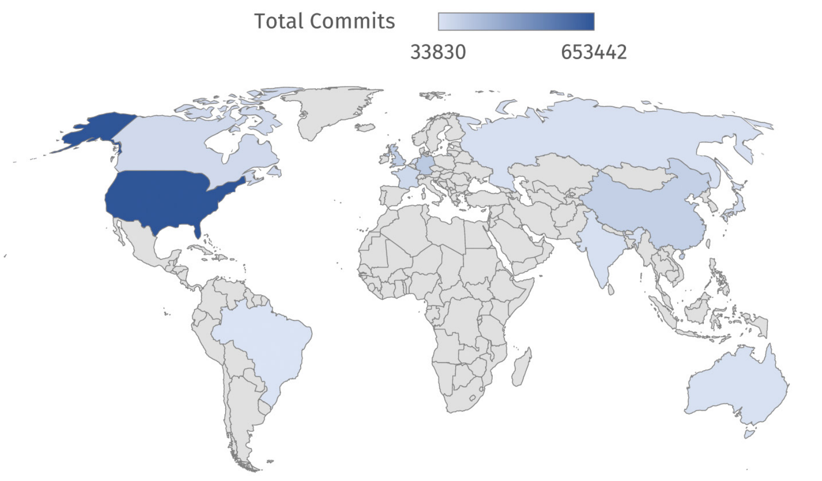


**Figure 1:** Geographic mix of OSS contributors on GitHub, 1Q19. Source: Open Source Compass (https://opensourcecompass.io/locations); note that this map excludes countries with fewer than 5,000 commits.

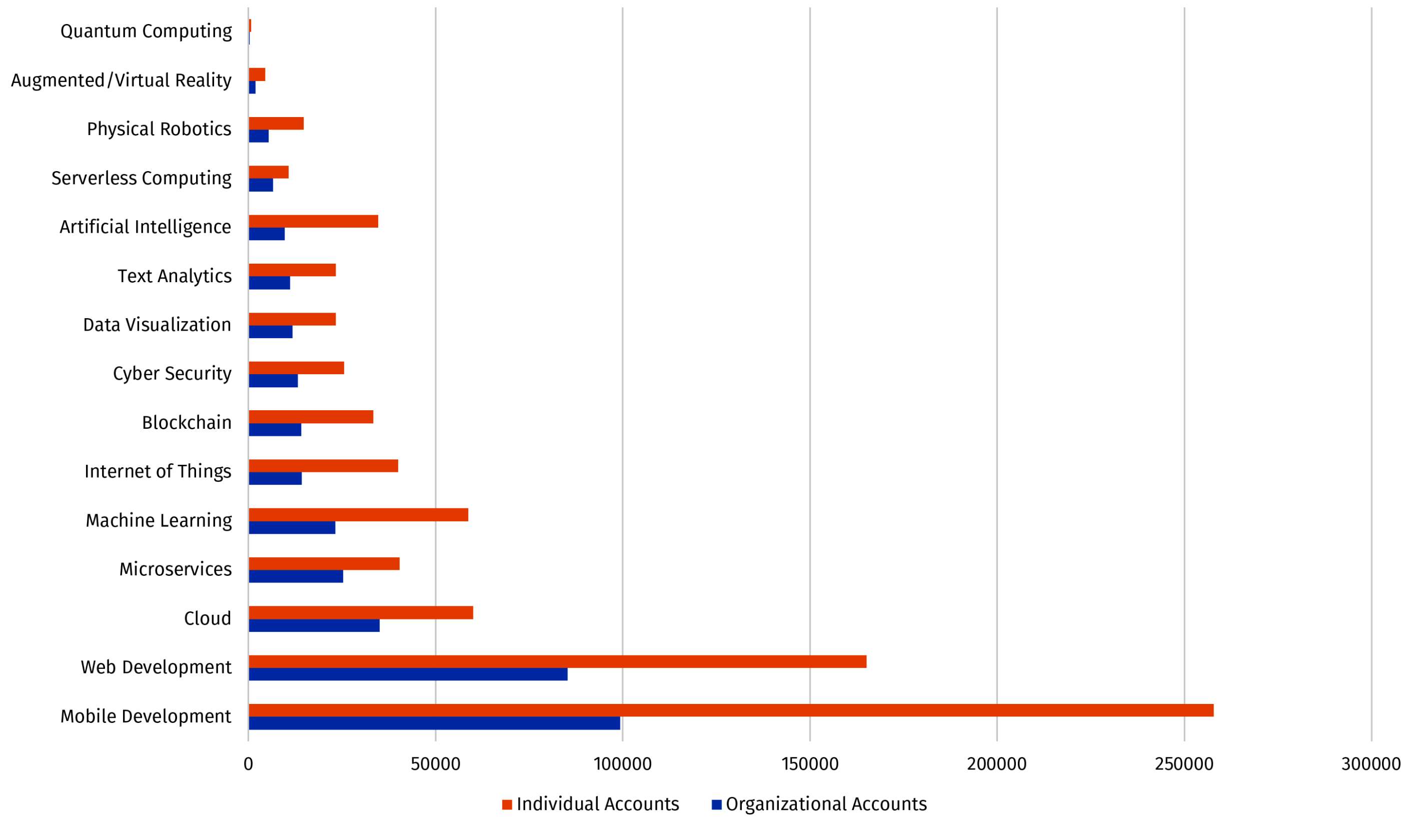


**Figure 2:** Technological mix of GitHub contributions, 1Q19. Source: Open Source Compass (https://opensourcecompass.io/domains/#which-domains-have-the-most-contributors), which uses data from the GH Torrent project, a research initiative led by Georgios Gousios of Delft University of Technology. GH Torrent monitors the GitHub public event timeline and retrieves and stores the contents and dependencies of each event.

## Conclusion

Each year, the Augean task of patching OSS vulnerabilities falls to small groups of solitary maintainers who generally rise to the occasion but who also have to balance competing commitments. This developer dynamic has unfortunate security ramifications for widely used software like bash, OpenSSL, and Apache Struts, the latter of which played a significant role in the Equifax breach. In parallel, bitsquatting and typosquatting (e.g., the `python3-dateutil` library masquerading as the popular `dateutil` tool) as well as developer infrastructure exploits (such as the `event-stream` hack) are opening up new attack vectors that undermine trust in OSS. In addition, with "rage-quit" takedowns (like the npm `left-pad` deletion [13], which briefly impacted React and Babel) and with maintainer withdrawal on libraries like `core-js` and `jsrsasign`, enterprise risk managers are increasingly attuned to the risk of broken builds. Given these challenges, federated package registries, cryptographically signed software packages, and reproducible builds are all steps in the right direction.

In the long run, however, establishing a *modus vivendi* between IT risk managers and open source developers will be critical to open source innovation, security, and competitiveness. Such an outcome will be as much a function of cultural adjustment as of technological advancement. Organizations paying the open source piper need to remain attuned to developer trust and transparency issues, and while there are few easy answers for how to sustain and secure OSS, paying it forward on maintenance is likely to generate outsized benefits, not only for end users, but also for society at large.

***References***

[1] P. Ford, "What Is Code?" *Bloomberg Businessweek*, June 11, 2015 (describing this secular shift in detail from the perspective of non-technical company managers): https://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/.

[2] In early 2020, the list includes media companies (Disney, CBS), insurers (State Farm, Liberty Mutual, and Northwestern Mutual), asset managers (JP Morgan, Goldman Sachs, BNY Mellon, BlackRock), industrial firms (3M, GE, and Emerson Electric), energy giants (Halliburton, DCP Midstream, and NRG), retailers (Walmart, Nordstrom, and Home Depot), airlines (Alaska Air and American Airlines), tractor OEMs (John Deere and AGCO), and automakers (Tesla and Ford), among others.

Another ≈20% of the Fortune 500 appears to have OSS placeholder pages for brand integrity and/or developer recruiting purposes.

[3] See generally T. Wang, "Knowledge Spillovers in the Open Source Community," Toulouse Digital Seminar, 2017; see also J. Meinwald, "Why Two Sigma Contributes to Open Source" (January 29, 2018): https://www.youtube.com/watch?v=5lk7LJU_zZM.

[4] F. Nagle, J. Wilkerson, J. Dana, and J. L. Hoffman, "Vulnerabilities in the Core Preliminary Report and Census II of Open Source Software," The Linux Foundation & The Laboratory for Innovation Science at Harvard, February 18, 2020: https://www.coreinfrastructure.org/wp-content/uploads/sites/6/2020/02/census_ii_vulnerabilities_in_the_core.pdf.

[5] G. Haff, *How Open Source Ate Software* (Apress, 2018), p. 172.

[6] C. Aniszczyk, "Open Source Gerrymandering," Oct. 8, 2019: https://www.aniszczyk.org/2019/10/08/open-source-gerrymandering/; B. Scott, "The Hacker Hacked," *Aeon*, August 10, 2015: https://aeon.co/essays/how-yuppies-hacked-the-original-hacker-ethos.

[7] See generally Y. Benkler, "Coase's Penguin, or, Linux and *The Nature of the Firm*," *Yale Law Journal*, vol. 112, no. 3 (December 2002), pp. 369–446 : https://www.yalelawjournal.org/article/coases-penguin-or-linux-and-the-nature-of-the-firm.

[8] N. Eghbal, *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure* (Ford Foundation, 2016): https://www.fordfoundation.org/work/learning/research-reports/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure/.

[9] "Our Stewardship of GitLab": https://about.gitlab.com/company/stewardship/.

[10] A. Vance, "Open Source Code Will Survive the Apocalypse in an Arctic Cave," *Bloomberg Businessweek*, November 13, 2019: https://www.bloomberg.com/news/features/2019-11-13/microsoft-apocalypse-proofs-open-source-code-in-an-arctic-cave.

[11] See [7], citing Y. Benkler, 2002. As Benkler notes, "the geek culture that easily recognizes 'Coase' doesn't [always] recognize the 'Penguin,' and vice versa. 'Coase' refers to Ronald Coase, who originated the transactions costs theory of the firm that provides the methodological template for the positive analysis of peer production…The penguin refers to the fact that the Linux kernel development community has adopted the image of a paunchy penguin as its mascot/trademark. One result of this cross-cultural conversation is that [discussions of open source require one to] explain in some detail concepts that are well known in one community but not in the other."

[12] See [4], citing, Nagle et al., 2020.

[13] D. Haney, "NPM & left-pad: Have We Forgotten How to Program?" (March 23, 2016): https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/.

# /dev/random
## Rewind Your Mind

ROBERT G. FERRELL

Robert G. Ferrell, author of *The Tol Chronicles*, spends most of his time writing humor, fantasy, and science fiction.
rgferrell@gmail.com

In the course of writing a speculative fiction short story about the direction human intellectual evolution might take (it doesn't involve giant melon-shaped foreheads with pulsing veins, if that's what you were visualizing), I found myself ruminating on the intersection between human and artificial intelligence. If we are to consider that, sooner or later, we and machines will become competitors for the same resources (electricity and self-direction), then it might be logical to presume that evolutionary fitness principles will also apply.

*Which is to be master, that's all.*

It seems probable to me that carbon and silicon will eventually merge, although perhaps not in the way many people envision. One of the first points of intersection may well be solid state biological memory. Not SSDs with our neural connections imprinted on them (we'll get to that later), but rather onboard computing of physiological data derived from embedded sensors, the results of which may be downloaded by your friendly neighborhood medical professional whether you like it or not. Taking your blood pressure or assaying your CBC might soon happen anytime you wander too near an RFI (Radio Frequency Intrusion) hub. That certainly puts the "Portability" into HIPAA.

Since we've brushed lightly past the subject, how practical is the "store your complete neural identity in electronic form" pipe dream/nightmare? Given that each of your 16 billion or so cortical neurons can have thousands of connections—which makes your neocortex a neural network of neural networks—we're talking about a level of convolution that would impress even a tax code author. I've seen a plethora of thought experiments on "post-humanity" that reduce us to digitized entities streaming Douglas Adams-style across the universe as a series of ones and zeroes. I think this is about as far-fetched as Star Trek teleportation, to be brutally honest (or honestly brutal, which, not to be brutal, I honestly prefer). Reducing our cognition to a collection of binary impulses seems beyond impractical.

I think neurons in the neocortex communicate not only using simple point-to-point connections, but also by interpreting patterns generated by attenuation of depolarization signals traveling those connected nerve fibers. Axons aren't just "on" or "off," in other words: they can demonstrate different signal strengths, which can then be used to overlay more information onto the binary connection map. This adds another layer of complexity, the depth of which is at least partially dependent on the minimum pattern size needed for constructing meaningful data objects.

Let's say memories are stored like multimedia files, with video, audio, olfactory, and gustatory tracks. Rather than a simple bitwise image map, however, we'll pretend the optical component is compressed by some form of pattern-based encoding that is then decoded by the visual cortex when a memory is replayed. That encoding relies on a large collection of "primitives" or stored data archetypes stitched together from the individual's past experiences. When we remember a scene containing a tree, for example, we don't need to visualize a specific tree unless that specificity is integral to the memory. How much space a memory requires depends on the number of unique moieties it contains and the array of "facets" each of these exhibits. Accessing a memory containing only a few modifications from an existing template is, after all, a lot less processor-intensive than building the entire scene from scratch. Think of it as "clipping" for the memory.

While the process by which it is accomplished is even less clear to me, the brain may also use the archetype approach for smells, sounds, and tastes. Tastes are probably the simplest, since they are all some combination of the five identified base sensations (20% sour / 15% sweet / 5% bitter / 35% salty / 25% umami, for example). This scheme is no doubt overly simplistic (especially since science recognizes seven, not five, basic tastes), but you get the idea. Odors, being closely associated with tastes, are likely stored in much the same manner. The audio track has to encode, at a bare minimum, pitch, timbre, rhythm, balance, dynamics, and several other characteristics. There are doubtless archetypes for all of these, too. Percussion, strings (plucked and bowed), winds, and voice must have their own sets of primitives that can be mixed and matched to create any music. This presumably goes as well for sounds of nonmusical origin (such as my singing).

The longer I think about this, the more it seems to me that the algorithms for data storage and retrieval in the human memory are probably even more subtly complex than we currently imagine. I expect some sophisticated sorting goes on, such that each data object can trigger a variety of different patterns depending on the contextual filtering it experiences along the way to the area where the memory is rendered. The brain in this respect works more like an analog music synthesizer than a digital computer. I think memories could well be categorized as waves, rather than particles; perhaps there's even a photon-like duality at work. Maybe thoughts are themselves packetized in quanta, giving the term "neuron" another meaning altogether: the intelligence particle. Its anti-particle is, then, the "moron."

Storing ourselves electronically may require a continuous recording medium like magnetic tape, as opposed to a lattice of discrete bits. Future humans might need to carry around some kind of analog-to-digital converter in order to back up to or restore memories from hard drives. After all, thoughts are not exactly binary in nature. What do you see in your mind's eye when you hear your favorite music: zero or one? Not a useful descriptor.

Mapping and storing a human's mental landscape would, realistically, require a lot more than just bit-flipping. I believe that our brains use those patterns we discussed as fundamental storage tokens. Sensory input is formed into multidimensional objects that are then stored ad hoc in some pseudo-hierarchical matrix. Specific memories are composed of pattern fragments pulled from this cache using a linked index created by ranking those fragments by frequency of appearance and something representationally equivalent to color or texture, along with other metadata.

Perhaps the brain employs a QR code-like mechanism to assemble complex memories from disparate archives scattered around wherever those moieties could be fitted in (hence the "ad hoc"). It does seem that something akin to disk fragmentation occurs in my own memory from time to time, which leads to attention headache. People with true long-duration eidetic recollection may keep all the fragments of a memory object in much closer logical proximity to one another than do the rest of us. I'm pretty certain my sensory input tumbles immediately into a neural woodchipper, to be blown across the perceptual lawn like gale-driven autumn leaves. My memory is more pathetic than eidetic.

Or maybe this whole line of reasoning is utter nonsense. Perhaps it turns out we store our memories on a very long VHS tape looping in the hippocampus. If we forget to rewind, it takes a lot longer the next time we want to access that memory. I'm pretty sure that somewhere on my personal VHS tape there is a memory of flunking neuroanatomy, so you might think I would avoid tossing around terms like "gyrus," "sulcus," "nucleus," and "ganglion," but it makes me feel like a stable genius.

# Book Reviews

MARK LAMOURINE

**Docker in Action, 2nd Edition**
Jeff Nickoloff and Stephen Kuenzli
Manning Publications, 2019, 310 pages
ISBN 978-1-61-729476-1

I guess you could say that when a tech book reaches a second edition the software it describes has reached some kind of maturity. Docker has inspired a whole new type of software infrastructure, and today there are a variety of resources for the beginner building and using containerized software. It's still a niche, however large, and it's still an advanced topic. Running containerized software requires the skills of a software developer, systems and network administrator, and operator.

The first edition of *Docker in Action* was one of the early books to market. A lot has happened since 2016, and they've added a few chapters and updated the rest.

*Docker in Action* follows the common narrative path for tutorial style references. They start with justification, show basics, and add features until they've covered the topic. Containers are easier to start with than some things because of the presence of public repositories of working images. With Docker, you can create a functional default configured database or web server in a few minutes. That's enough to hook a reader early and give a sense of what is possible. Nickoloff and Kuenzli use the first section to teach the reader how to run single containers on a single host. This includes adding storage, network communications, and customized configuration to make a useful service.

The second section is devoted to creating new container images. The chapter on creating containers really only touches on the basics, as there are lots of good references on the details. The section is about more than just building images. The succeeding chapters show how to interact with public and private image repositories and how to automate the production, testing, and publication of new container images, all triggered from public source code repositories. When combined, these capabilities form a software development and delivery chain.

I like the authors' writing style. They are clear and concise. The theoretical exposition is balanced nicely with the practical elements. I do wish there were more external references, either in the text or in the chapter summaries. I know from experience that the Docker website has detailed references describing all of the keywords available for creating Dockerfiles. The authors only demonstrate the basics needed to get started, which is adequate as they have limited space. However, I would have liked to see reference callouts to those well-known stable resources.

In the final section, the authors introduce container orchestration. This is the idea of describing and automating clusters of coordinating containers to form larger applications. It is possible to start a database container, a front-end web server, and a middleware container to implement some kind of business logic, and to do all this manually, a step at a time. Applications like this form patterns, though, and the patterns make it possible to build services to manage the deployment of these complex sets of containers.

The authors use Docker Swarm to show the possibilities of container orchestration. Swarm is an integral part of the Docker application system and so is available anywhere that Docker itself is. The alternatives, such as Kubernetes or the commercial cloud offerings, each have whole books devoted to them, so Swarm is a good choice for a first look. The authors admit that Swarm probably isn't suitable for large-scale deployments, but perhaps it has a place in production in smaller shops.

Likewise, the authors make no mention of alternative container runtime systems or tool sets. I used to liken the Docker suite to the BASIC programming language. It is a good easy starting point to engage and learn concepts, but it is possible to outgrow its capabilities and its limits. The Open Container Foundation describes a standard container format and a standard runtime behavior. Docker is one compliant system, but there are others.

For a moderately experienced system administrator, this second edition of *Docker in Action* will be a good introduction to container systems. Like VMs, container management requires an understanding of underlying storage and complex networking that this book only glosses over. To go deeper, the reader will have to keep learning, but this is enough to get started doing useful work.

**Microservices and Containers**
Parminder Singh Kocher
Addison-Wesley Professional, 2018, 283 pages
ISBN: 978-0-13-459838-3

I'm the kind of geek who likes a mix of theory and practice in a tech book. For some reason, most of the books I've seen on software containers and microservices tend to be tutorials for specific technologies. In *Microservices and Containers* Kocher does discuss the tools, but he doesn't stick to just the syntax and behavior. The first section is devoted to an overview of Microservices.

The flexibility that microservices offer comes with some up-front cost. People who first hear about how easy Docker is to use for simple containers want to jump right in and port their applications to single containers. I like that Kocher doesn't give in to the temptation to get right to the sexy tech.

The term "microservice" refers to the components that are used to make up a conventional application stack. In the original LAMP (Linux, Apache, MySQL, PHP) stack, the components are installed directly onto a host computer. Using software containers, it is possible to implement the same behavior running the service components in containers rather than installing them directly on the host.

Containers impose boundaries that conventional host installations do not. Porting an application to microservices tends to expose the boundaries that are often neglected or left implicit in a conventional deployment. Kocher does a good job of addressing the challenges that porting an application poses.

Inevitably, when Kocher starts to talk about the implementation of individual microservices, he is forced to revert to expressing it in terms of an existing container system. Despite the existence of a number of alternative runtime and container image build tools, Docker remains the overwhelmingly dominant environment. In the middle section of the book he provides the same catalog of Docker commands that you'll find in other books.

This book is one of the unfortunate cases where the print and ebook versions are significantly different in appearance. The ebook has color graphics that don't convert well to grayscale. Furthermore, the code examples in the print version are compressed to fit the pages to the point that they are nearly unreadable.

The final chapter of this section covers container orchestration, and Kocher returns to implementation agnosticism. There are whole books about Kubernetes, Mesos, and Swarm, and he doesn't try to go into depth about any of them before returning to their common features: automation, service discovery, and global metrics.

In the final section, Kocher distinguishes himself again with a set of case studies in implementation and migration. Again, this book isn't long enough to be a comprehensive guide, but it is sufficient to give the experienced reader a sense of the different challenges that microservice design, deployment, and management present. Three cases are used to explore and then contrast a monolithic deployment and a fully containerized one. He includes an intermediate case where the application is in the process of migration. Together, these case studies expose the assumptions underlying a monolithic deployment and the common misconceptions about containerization that can undermine a project.

I liked Kocher's perspective and his approach to microservice applications. He shows a thorough understanding of the issues that I often see downplayed by other authors in their enthusiasm for the tech. I don't think the full potential of microservice architecture has made it to the mainstream yet. In *Microservices and Containers*, Kocher presents a realistic path for application designers to explore the possibilities.

## An Illustrated Book of Bad Arguments, 2nd Edition

Ali Almossawi, illustrated by Alejandro Giraldo
The Experiment LLC, 2014, 56 pages
ISBN 978-1-61-519225-0

First Edition: http://bookofbadarguments.com
Creative Commons BY-NC license
ISBN 978-1-61-519226-7

It's hard to swing a syllogism these days without hitting a bad argument. It's one thing, though, to know that something isn't right and another to know *what's* not right about it. Aristotelian logic was required for the engineering students where I went to college, but most of the focus was on how to create and evaluate good arguments. The most illustrative lesson on bad arguments was the 10-minute comedy set at the beginning of the first lecture in which the professor enumerated the ways students would try to persuade him to give them a better grade, and why he wouldn't be swayed by any of them.

I also remember that most of the other students in the class were intimidated by the professor and the topic. Logic has a reputation for being difficult and the province of nerds. Logic is like grammar—people who make a big deal about rigor in daily life are mostly annoying to others.

Making logic palatable, even amusing, is the challenge that Almossawi took on in 2013 when he published the first edition of *An Illustrated Book of Bad Arguments* as an online book. He released it under a Creative Commons Non-Commercial license then, and this second edition was published the following year in print. As the title indicates, he focuses on how arguments go bad. You won't find more than the most basic definition of terms needed to understand what a good argument is and is not.

Most of the arguments made in the public sphere today are constructed rather informally, and most of the ways they are broken are informal as well. A formal argument is literally one that has the correct form. There are logical fallacies related to the form of an argument, that is, where the failure of the argument comes from the failure of the structure of the argument, but most of the fallacies you find in discourse today are not of this type. In fact, Almossawi offers only one formal fallacy. The rest of the 19 total examples are informal fallacies. This makes them no less significant.

Each pair of facing pages describes and demonstrates one form of logical fallacy. The footer includes the fallacy's place in the taxonomy of bad argument. Yes, fallacies have families. I hadn't realized until I saw the diagram in the front of the book that most fallacies are a variation of a red herring. They divert attention away from the actual argument by offering something unrelated to the point. All of the informal fallacies are a form of *non sequitur,* or "does not follow."

The text for each page is brief and clear. The illustrations have the style of 19th- and early-20th-century woodcuts. They remind me of the illustrations from *Alice in Wonderland* or the animals from my mother's "Laughing Brook" books by Thornton W. Burgess. The cover and pages are printed to look antiqued.

You're not going to make any friends by pulling out this book and pointing at a page the next time you're on Facebook. It is useful for understanding the myriad ways what you see there can be wrong. It's really important to understand that an invalid argument does not mean that the conclusion is false. It just means you can't prove it *that way*. It is good to have a taxonomy and a name for each of the ways that an argument can go wrong, and it's most helpful for me to recognize when I find myself leaning on these when my own biases and wishes try to lead me off the path. *Bad Arguments* is a slim volume or URL to keep handy when you find yourself thinking "Hey, wait a minute…"

## Running Virtual PC (vPC) Meetings

*Erez Zadok and Ada Gavrilovska*
*2020 USENIX Annual Technical Conference (USENIX ATC '20) Program Co-Chairs*

As the co-chairs of the USENIX ATC '20 PC, our original plans to hold an in-person PC meeting pivoted to virtual PC (vPC) meeting planning due to COVID-19. Along with our very helpful submission chairs (Dongyoon Lee from Stony Brook University and Ketan Bhardwaj from Georgia Institute of Technology), we experimented with three solutions to see what would work best: Webex, BlueJeans, and Zoom. We have now concluded running the vPC meeting, with over 70 participants for at least part of the meeting. Below we describe our experiences in planning and running the vPC.

Ultimately, we settled on Zoom, but it did not solve all of our problems. At this point, we are mainly interested in reporting our experiences while they are still fresh in our memory, in hopes you will find it useful. It would take more time and experimentation to turn this document into a concrete set of recommendations.

Running USENIX ATC is a relatively complex operation for many reasons, including the number of submissions (in the hundreds), and the three tiers of reviewers (numbering almost 120). The two co-chairs and the two submission chairs all need administrative privileges in the online paper reviewing system (HotCRP.com).

### vPC Meeting Requirements

1. Our key need for the PC meeting is how to handle conflicts of interest (CoI). In a physical PC meeting, any PC members with a conflict are kicked out of the room, and called back in after the conflicted paper's discussion is over. This requires a waiting room feature.
2. There are numerous tasks that all four of us have to handle efficiently: watching and moving the discussions along, marking decisions, reviewing paper summaries, picking and assigning shepherds, and of course managing conflicts of interest (CoIs). As a result, all four of us need to have admin privileges when running the meeting, not just in HotCRP.
3. We need to verify the identity of PC members, and ensure that only invited individuals can join the meeting after proper authentication.

### Webex

Webex allows the host to define alternate hosts. Alas, only one of the alternates at a time can be an active host: once person A delegates host privileges to person B, person A loses host privileges and can't get them back. What we need is true co-hosting, and Webex doesn't seem to support that at the moment.

Webex does have a decent waiting room feature: we were able to manually move attendees to the waiting room and verify that they could not hear or see anything, and could not get back in on their own.

Webex has a very nice registration feature: you invite N people with specific emails and names to a Webex meeting. They are required to register with the email they were invited with, and they cannot change their name.

### BlueJeans

BlueJeans supports multiple co-hosts. It also supports a "breakout room," and we were able to move people to it. Alas, people in the breakout room could rejoin the main meeting on their own—clearly undesired. (I guess it's like a conflicted PC meeting member who is outside the main room barging right back in.)

We didn't test BlueJeans's registration feature, as the breakout room problem was a showstopper for us.

### Zoom

Zoom has a rudimentary role-based access control system, and allows one host and multiple co-hosts at a time. It allows true co-hosts, which the host can define when creating the meeting, but they need to have a Zoom account. If they don't, the host can easily promote them to co-hosts after the meeting starts. Only hosts can declare others as co-hosts, and the host can even hand off actual host privileges to another co-host, but cannot take them back. All co-hosts have the same admin control over the meeting: they can admit people in/out, un/mute all, etc.

In our experience, Zoom's waiting room worked very well. Participants with conflicts could be kicked out of the meeting and sent into the waiting room, where they could not hear or see anything. We could then re-admit them all with a single click of the admit-all button, and go on to remove the next set of CoI out of the meeting. The key here is that all co-hosts were able to manage these conflicts and the waiting room, allowing us to better parallelize (and double check) this complex task.

Zoom's registration feature is not as good as Webex's. We had to send the Zoom URL to all of our PC members, who then had to register with a valid email and enter their names. They received an email with a personal link to join the meeting—thankfully not a shared URL that could be easily zoombombed. However, they were able to enter any valid email and any first/last name. In theory, someone could create a new dummy email and masquerade as another PC member if they got their hands on the invitation URL.

In the future, we will need to ask PC members to use their proper names and emails that are registered in HotCRP. When the meeting starts, all PC members will be in the waiting room by default, and we'll have to verify one by one whom we are admitting into the meeting—otherwise we can chat privately with them in Zoom to establish their identity. Once we admit everyone, we can turn off the "participants can rename themselves" feature.

Registration becomes even more important for people who will dial in by phone to the meeting. They will still have to register with a per-participant link; then they will receive an email with instructions for connecting to the meeting with a personal phone code that identifies them. When dial-in users connect, they are shown as "Phone User N." We have to identify them by voice and rename them in the Zoom participants list so everyone knows who they are.

### Other Solutions?

We heard that at least one PC meeting via Microsoft Teams worked well. Given that we were reasonably pleased with the Zoom setup, and were not sure we had access to test Teams, we did not investigate it. Erez did have the opportunity to join a Teams meeting recently, described below, and we are interested to hear from anyone who has detailed experience with it.

Erez recently joined three different back-to-back meetings with about 6–8 people each, using Microsoft Teams, Zoom, and Google Meet. Overall, he felt that Zoom worked much better and doubted that Teams or Meet would have met our vPC needs.

Microsoft Teams does seem to have a waiting room feature, as Erez had to wait to be admitted, but it's unclear how well it would work for running a vPC. Video and audio quality was lower for some participants; while it might have been their Internet connections, we fear that it might not have scaled to our PC size. Only four people's videos were visible at a time, limiting the ability to feel inclusive and see more people. After examining all the buttons and menu options during the meeting, it seemed to Erez that Teams had far fewer features.

Google Meet also has very few features compared to Zoom, and even fewer than Microsoft Teams. The worst part was that the audio and video quality in Google Meet was considerably poorer for everyone participating. Even turning off everyone's video and streaming audio only, the quality was still fairly choppy.

Webex Teams, which we did not have access to test, reportedly supports multiple concurrent co-hosts.

### Experiences from Running the Actual Virtual PC

With a virtual PC, there was more to manage at once. It was important that each organizer use a computer with a large screen—even two screens. We had to have the conference paper management window open as well as the Zoom window, with sub-windows for chat, the participant list, and the waiting room list, our email and messaging client (or cell phone), since people were emailing or texting us with various issues, and a private Slack chat window for the organizers.

When streaming media for hours, some people's computers overheated and shut down after a few hours. It is important to have a sufficiently powerful computer for long-running CPU-hog processes like video and audio streaming.

We used Slack as a side channel for private communications among the meeting organizers. We could have used Zoom's chat feature, but it was too risky—participants could inadvertently broadcast something publicly unintentionally. So we allowed participants to chat only with the host(s) in Zoom. It was useful as people had to tell us about last-minute schedule changes or other requests. The Zoom messaging feature was not very convenient, however, when we needed to send the same message to a few participants (but not all, so as not to violate conflicts), for instance, that their paper would need to be reshuffled in the schedule. Also, Zoom let participants chat with one of the co-hosts but not all of them as a group. Lastly, there was no way to clear the chat history between paper discussions in order to avoid leaking information to other participants once they rejoined.

While Zoom permitted us to manage conflicts as described above, it took time to do so: we had to look up the conflicts in HotCRP, then scroll or search for the right participant in the participant list, then move them to the waiting room one by one. There is no feature for participants to take themselves into the waiting room the way they would during an in-person PC meeting. Zoom, perhaps under network stress, had a delay of 2–3 seconds between when you kicked someone off the meeting and they actually showed up in the waiting room. So it took 1–2 minutes per paper just to manage those conflicts, precious time when you are under a tight schedule. Conversely, in a physical PC meeting, you quickly call the names of all conflicted members, and they all get up at once and leave the room in parallel.

Zoom shows at most 25 participants' video at once, and not all of our participants used their video. (One insisted on calling in from an anonymous phone number due to reports of Zoom privacy concerns.) This made it harder for PC members to know when they could jump in and speak. We tried to manage the order as best we could, calling on people in turn, and we also used the "raise hand" feature a bit, but it still took longer than with an in-person meeting. There were also natural delays in people's audio/video stream and a few people with poor connections. All this added another 1-2 minutes of time when discussing each paper.

When a PC meeting is held in person, people come from all over the world and are present at the start of the meeting at the designated time. But with a virtual PC meeting spanning 12–15 time zones, it was impossible to expect people to be at the meeting at ridiculous early/late hours. So our meeting was scheduled for the middle of the day. We sent a Doodle survey to see what times people could attend, and we tried our best to group papers based on people's time constraints—not an easy task. Worse, because of COVID19, people had day job duties they couldn't ignore, childcare duties, last-minute schedule changes, and more. We had to adapt to people's changing schedules dynamically. This added more "context switching" time between papers.

A few other aspects made the process challenging. First, it was more difficult to control inadvertent leakage of information about paper reviewers—we had cases where either one of us or reviewers themselves asked if we could do paper #X before they left, or when we waited to discuss a paper because of a missing reviewer, but now that information was visible to others—they saw who just joined the meeting. Likely some of this exists in an in-person PC meeting, but probably less so. Second, managing the discussions to wrap up in a fixed amount of time was more difficult, given the lack of other options. PC voting as an option really doesn't work in an online format. We rarely had the full PC, and with people coming and going and videos switched off, it was difficult to tell who was around, whether they would listen in a brief summary of the discussion before voting, etc. As a result, in cases when the PC discussion was "deadlocked" and it was obvious that a reviewer's vote wouldn't resolve it (e.g., an even number of reviewers split 50/50), asking the PC to vote could not resolve the paper's status.

In addition, it was harder to ask the PC members to take the conversation offline and report back—something that's commonly done during in-person PC meetings—because of the above-mentioned issue with time zones and daytime duties. Taking a conversation "offline" meant pushing papers to be decided at some undetermined later point, likely after the actual PC meeting. These two issues made it harder to cut discussions short, which again added to the meeting time.

We already expected that our virtual PC meeting wasn't going to be as effective as an in-person one would have been. So for weeks leading to the meeting, we pushed our PC hard to try and reach a decision on as many papers as possible. That certainly helped a lot (and we have even heard of some PC Chairs who canceled their online PC meeting so they didn't have to deal with the complexities of running it virtually). Still, all these complications caused our PC meeting, originally scheduled for five hours, to take seven hours. And we still had a few of the discussed papers to finalize offline after the meeting.

Finally, a word about security and privacy. Since Zoom saw its user base grow 20-fold in just a few months, it has attracted a lot of media attention and reporting of serious security and privacy concerns. (This is not to suggest that Zoom's competitors' security and privacy practices are perfect and their software bug-free.) As a result, a few high profile communities (e.g., school districts) banned or abandoned Zoom altogether. To their credit, Zoom has apologized publicly, has begun to address these concerns, and has already released several security fixes and new features, promising more. Still, some of our PC members, understandably, preferred not to run the Zoom client or accept their privacy policy as there are reports of numerous Zoom users' credentials sold on the dark web. These users called in via phone instead.

# Writing for *;login:*

We are looking for people with personal experience and expertise who want to share their knowledge by writing. USENIX supports many conferences and workshops, and articles about topics related to any of these subject areas (system administration, programming, SRE, file systems, storage, networking, distributed systems, operating systems, and security) are welcome. We will also publish opinion articles that are relevant to the computer sciences research community, as well as the system adminstrator and SRE communities.

Writing is not easy for most of us. Having your writing rejected, for any reason, is no fun at all. The way to get your articles published in *;login:*, with the least effort on your part and on the part of the staff of *;login:*, is to submit a proposal to login@usenix.org.

## PROPOSALS

In the world of publishing, writing a proposal is nothing new. If you plan on writing a book, you need to write one chapter, a proposed table of contents, and the proposal itself and send the package to a book publisher. Writing the entire book first is asking for rejection, unless you are a well-known, popular writer.

*;login:* proposals are not like paper submission abstracts. We are not asking you to write a draft of the article as the proposal, but instead to describe the article you wish to write. There are some elements that you will want to include in any proposal:

- What's the topic of the article?
- What type of article is it (case study, tutorial, editorial, article based on published paper, etc.)?
- Who is the intended audience (syadmins, programmers, security wonks, network admins, etc.)?
- Why does this article need to be read?
- What, if any, non-text elements (illustrations, code, diagrams, etc.) will be included?
- What is the approximate length of the article?

Start out by answering each of those six questions. In answering the question about length, the limit for articles is about 3,000 words, and we avoid publishing articles longer than six pages. We suggest that you try to keep your article between two and five pages, as this matches the attention span of many people.

The answer to the question about why the article needs to be read is the place to wax enthusiastic. We do not want marketing, but your most eloquent explanation of why this article is important to the readership of *;login:*, which is also the membership of USENIX.

## UNACCEPTABLE ARTICLES

*;login:* will not publish certain articles. These include but are not limited to:

- Previously published articles. A piece that has appeared on your own Web server but has not been posted to USENET or slashdot is not considered to have been published.
- Marketing pieces of any type. We don't accept articles about products. "Marketing" does not include being enthusiastic about a new tool or software that you can download for free, and you are encouraged to write case studies of hardware or software that you helped install and configure, as long as you are not affiliated with or paid by the company you are writing about.
- Personal attacks

## FORMAT

The initial reading of your article will be done by people using UNIX systems. Later phases involve Macs, but please send us text/plain formatted documents for the proposal. Send proposals to login@usenix.org.

The final version can be text/plain, text/html, text/markdown, LaTeX, or Microsoft Word/Libre Office. Illustrations should be PDF or EPS if possible. Raster formats (TIFF, PNG, or JPG) are also acceptable, and should be a minimum of 1,200 pixels wide.

## DEADLINES

For our publishing deadlines, including the time you can expect to be asked to read proofs of your article, see the online schedule at www.usenix.org/publications/login/publication_schedule.

## COPYRIGHT

You own the copyright to your work and grant USENIX first publication rights. USENIX owns the copyright on the collection that is each issue of *;login:*. You have control over who may reprint your text; financial negotiations are a private matter between you and any reprinter.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

**POSTMASTER**
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
**PAID**
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES