HACKERMONTHLY

Issue 56 January 2015

Curator
Lim Cheng Soon

Contributors
Doug Bierend
Zach Holman
Lawrence Kesteloot
Jeffrey Ventrella
Damien Katz
Carlos Bueno
Correl Roush
Yunong Xiao

Proofreader
Emily Griffin

Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

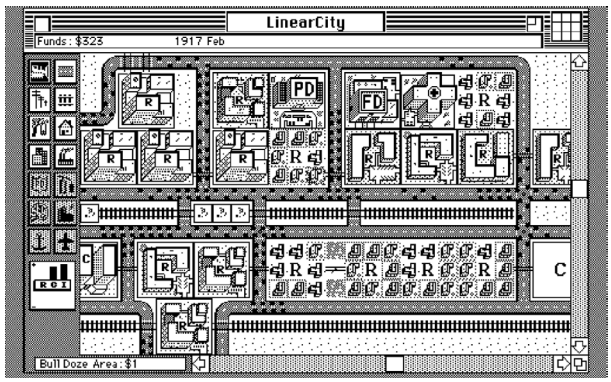
Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

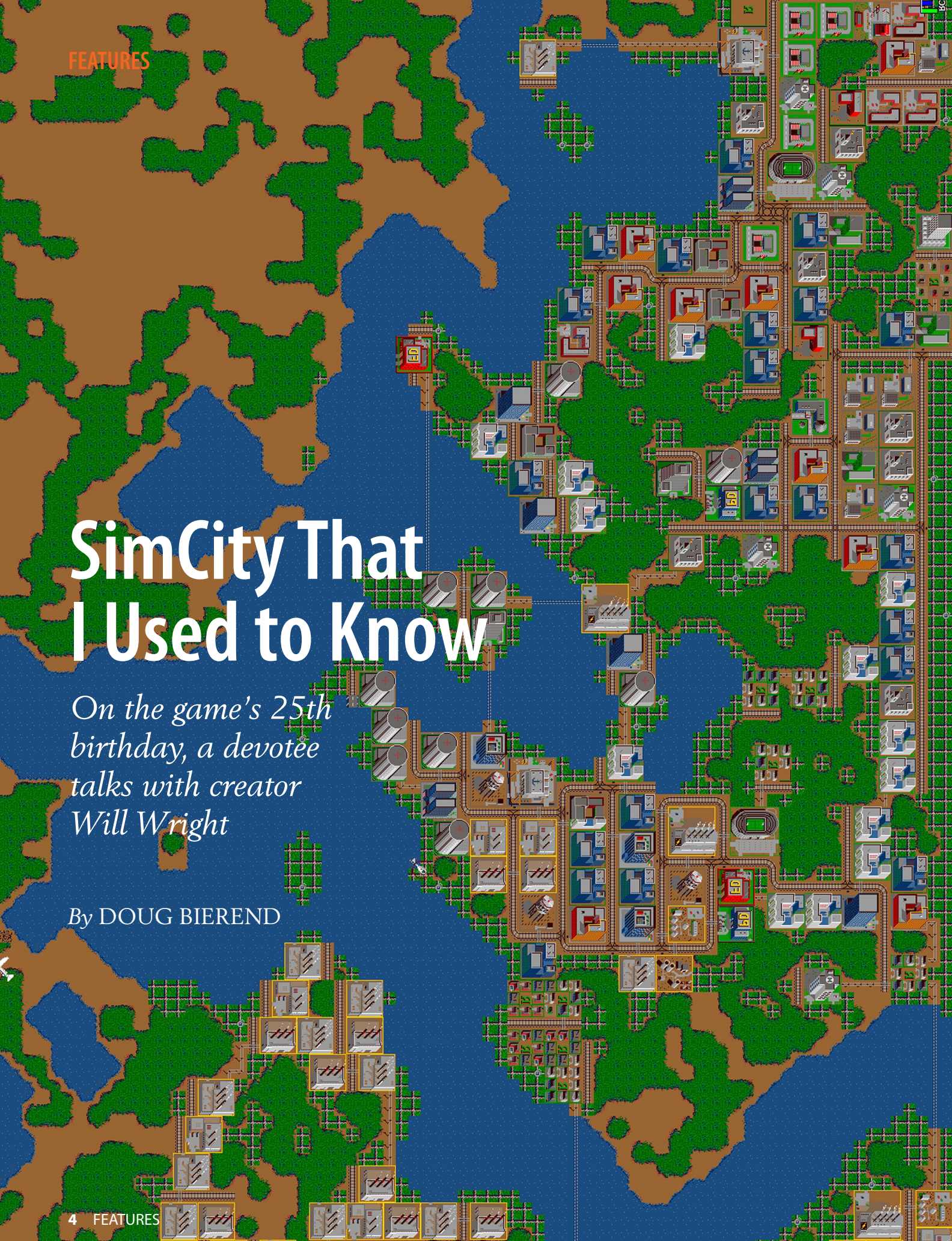
Contents

FEATURES



PROGRAMMING

For links to Hacker News dicussions, visit *hackermonthly.com/issue-56*



SimCity That I Used to Know

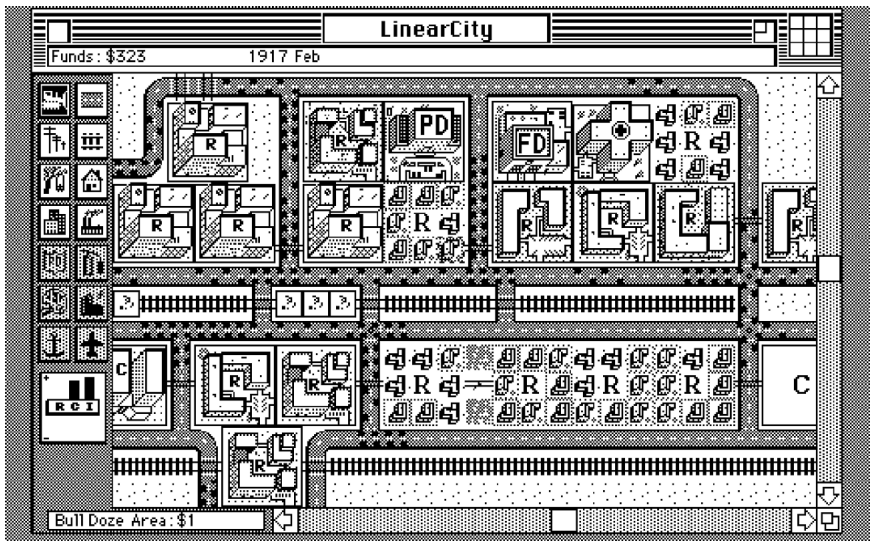
On the game's 25th birthday, a devotee talks with creator Will Wright

By DOUG BIEREND

SimCity, the classic PC game that makes mayors out of middle schoolers, turned 25 last week. Well, actually that's a common misconception — the IBM version of SimCity was released in October of '89, but the original (for Mac and Amiga), came out in February. I found this out from Will Wright, the game design guru behind SimCity and the genre of games it spawned, whose mental history of the legendary game is far more accurate than the Internet's. "I think everybody just puts too much trust in Wikipedia," he said.

Regardless of its precise birthday, SimCity was a hugely influential game, popularizing a genre of "software toys" that presented players with an interactive, complex world. Little gamers growing up in the '90s may remember a time when any new PC title from Maxis (the company Wright founded with partner Jeff Braun) bearing the prominent "SIM" prefix promised endless hours of play time that wasn't about winning or losing, but experimentation and discovery. The Sim series also represents a philosophy about design, and the role of play in our learning process.

"I think that play, in a more general sense, is fundamentally one of the ways that we understand the world, the real world," says Wright, "as is storytelling. I think the two are both kind of educational technologies, and that's the part that interests me; basically, how we take these things — whether it's storytelling, or play, or games — and use those to increase our understanding and our engagement with the real world, not pull us away from it."



A screenshot from the original SimCity game for the Macintosh.

SimCity gives the player macro- and micro-managerial control of a voxelated urban terrarium. Along with its subsequent titles — SimCopter, SimTower, SimAnt, Sim et cetera ad nauseam — SimCity kicked off a series of digital sandboxes that put complex systems within the grasp of anyone with a computer mouse.

Every decision has a consequence in the balance of dozens of variables. The RCI (residential, commercial, industrial) balance that guides the city's economy, the layering of transportation options and power lines, the funding of schools or location of prisons — each reacts in subtle or overt ways based on a simplified system of logic devised by Wright and his team. "How do we take these big complex things we're embedded in, and bring them into such a focus that we can now apply our natural instincts and intuitions to it?" Wright asks. Seeing cars begin to drive down the roads you've built, watching as neighborhoods flux and gentrify when a new commercial zone is established nearby, all while getting familiar with each little wrinkle of the city and its geography — you can start to imagine what it's like to live there.

Keeping track of a budget may sound more like a simulation of accounting than urban planning, but the beauty of the game is that it manages to turn things like fiscal policy into a feature of play. That's largely because their effects are visible. Unlike a "replicative" simulation (say, a baking simulator), which recreates an experience you might actually have in life, the scales of size and time are variable. With the added power to call up natural (and unnatural) disasters, this essentially transforms the player from mayor to god, and allows them to watch as the long term consequences of their choices unfold. "All of a sudden you get this totally different view of it. It feels like this organic picture in front of you.…That's kind of what I would call turning something into a toy that we can now play with."

These toys were especially effective for kids, who were at an age when the real and the imaginary seem less distinct. Watching as the little cities exhibited behavior in reaction to the player's actions

created a link between us the game. That link was also an intentional part of the game's design.

"They're starting to understand its behavior and you're getting kind of an instinct or an intuition for how it operates," he says, "in a totally different way than if you're reading a book about classical economic theory, which is entirely abstract." Wright is fond of the notion of seeing a player's brain as the second processor — its proclivities and responses feeding and responding to the invisible gears and pulleys behind the simulation.

For the link to work, the interface has to be intuitive enough that anyone from a preschooler to a PhD in urban systems can get their hands and heads around what it takes to grow their city. The array of variables they control must be dramatically simplified from reality of course, but complex enough that the models they create exhibit dynamic "behavior." A critical feature of the design of these games is, in fact, that they allow for unpredictable phenomena to arise. Unexpected harmonic convergences of circumstances occur in the city that lead to explosions of growth, or collapses of neighborhoods, say, from causes that aren't as easy to trace back as pointing out that, say, lowering police funding meant an increase in crime.

"Most of the simulation is really built up of rather simple rules, if you look under the hood, and it's really interesting how these simple rules, when they interact with each other, give rise to great complexity," Wright says. "You can't even really sit back and engineer it or blueprint it. It's more like you have to discover it, because the emergent systems are inherently, by definition, unpredictable."

Will Wright

The way Wright sees games, players occupy and explore what he calls "possibility spaces." Simply put, possibility spaces are all the potential arrangements a system (or game) might find itself in. The whole tree of possible movements of pieces on a chessboard, or the countless ways you might reach a destination in Grand Theft Auto, each are a kind of possibility space.

These spaces often intersect along numerous dimensions — in the Sims, for example, the interplay between social success and professional success created a jointed set of possibility spaces that a player could work to maximize (the game was designed around an ideal, not unlike in real life, that lay in achieving a balance between the two).

The possible choices faced by a player of SimCity include the aesthetic priorities, economic models, level of environmental concern, and other more subjective dimensions. Depending on the player, a city might be a green oasis or a Koyaanisqatsi-esque nightmare; some might try to make the most visually pleasing city they can, or simply have fun wreaking havoc. Their decisions in these spaces — which can be measured, by the way — are often a reflection of their own values and sensibilities.

"Players right off the bat were forced to sit down and in fact pick their goals," Wright says. "They had to first of all decide what their values were, what kind of city would they like to live in. That was part of it, and the other part of it was that at some point, invariably, the people who played it enough would start arguing with the assumptions of the simulation. They would start saying, "I don't think mass transit's that effective, I don't think pollution really would drive away that many residents." At that point, they're also having to clarify their internal model of the way a city operates…all of a sudden your assumptions become clear to you."

A screenshot from the first Windows version of SimCity.

A screenshot of the 2013 version of SimCity, its fifth major installment.

Wright's games — if you can call them that — were uniquely influential for a generation of kids with access to computers in the '90s. One could guide the complex course of events within a continent, a neighborhood, or beneath a picnic table, and leave with a more systemic understanding of each. An imaginative player could also weave their own stories between these layers — many hours were spent imagining stories taking place within and between these worlds.

"Whenever you see a kid that's really motivated and into something, it's entertaining to them, they love it," says Wright. "But at the same time that's also probably the most effective process of education."

"Fun and educational" is an aspirational combination of words, one that many products claim but few live up to. I certainly emerged from my hunched sessions with my pet cities carrying a new appreciation for the world around me. With all the talk of gamifying education, and with a new generation of teachers — the first in history — raised on video games, the value in approaching learning with games may get the real-world traction it deserves.

"I really think our brain is wired to consume entertainment and enjoy entertainment, precisely because of the fact that it's inherently educational," says Wright. "And we've made this artificial distinction between the two, we've almost kind of put a chasm there that didn't exist….I think SimCity was just a simple example that for a lot of people started to remove the wall between the two." ■

Doug Bierend is a writer. He writes about futurism and technology at VICE's Motherboard. Doug also write about design for Medium's ReForm, as well as photography and visual culture for Vantage. Previously, he wrote for WIRED and their Raw File blog.

Reprinted with permission of the original author. First appeared in *hn.my/simcity* (medium.com)

MOVE FAST AND BREAK NOTHING

A Talk About Code, Teams and Process

By ZACH HOLMAN

Moving Fast and Breaking Things

Let's start with the classic Facebook quote, *Move fast and break things*. Facebook's used that for years: it's a philosophy of trying out new ideas quickly so you can see if they survive in the marketplace. If they do, refine them; if they don't, throw them away without blowing a lot of time on development.

Breaking existing functionality is acceptable. It's a sign that you're pushing the boundaries. Product comes first.

Facebook was known for this motto, but in early 2014 they changed it to *Move fast with stability*, among other variations on the theme. They caught a lot of flak from the tech industry for this: something something "they're running away from their true hacker roots" something something. I think that's horseshit. Companies need to change and evolve. The challenges Facebook faces today aren't the same challenges they faced ten years ago. A company that's not changing is probably as innovative as tepid bathwater.

Around the time I started thinking about this talk, my friend sent me an IM:

> *Do you know why kittens and puppies are so cute?*
>
> *It's so we don't fucking eat them.*

Maybe it was the wine I was drinking or the mass quantity of Cheetos® Puffs™ I was consuming, but what she said both amused me and made me think about designing unintended effects inside of a company. A bit of an oxymoron, perhaps, but I think the best way to get things done in a company isn't to bash it over your employees' heads every few hours, but to instead build an environment that helps foster those effects. Kittens don't wear signs on them that explicitly exclaim "DON'T EAT ME PLS," but perhaps their cuteness helps lead us toward being kitten-carnivorous-averse. Likewise, telling your employees "DON'T BREAK SHIT" might not be the only approach to take.

I work at GitHub, so I'm not privy to what the culture is like at Facebook, but I can take a pretty obvious guess as to the external manifestations of their new motto: it means they break fewer APIs on their platform. But the motto is certainly more inward-facing than external-facing. What type of culture does that make for? Can we still move quickly? Are there parts of the product we can still break? Are there things we absolutely can't break? Can we build product in a safer manner?

This talk explores those questions. Specifically I break my talk into three parts: **code**, internal **process** in your development team and company, and the **talk**, discussion, and communication surrounding your process.

Code

I think *move fast and break things* is fine for many features. But the first step is identifying **what you cannot break**. These are things like **billing code** (as much as I'd like to, I probably shouldn't accidentally charge you a million dollars and then email you later with an "oops, sorry!"), **upgrades** (hardware or software upgrades can always be really dicey to perform), and **data migrations** (it's usually much harder to roll-back data changes).

The last two years we've been upgrading GitHub's permissions code to be faster, safer, cleaner, and generally better. It's a scary process, though. This is an absolute, 100% can't-ever-break use case. The private repository you pay us for can't suddenly be flipped open to the entire internet because of a bug in our deployed code. 0.02% failure isn't an option; 0% failure is mandatory.

But we like to move fast. We love to deploy new code incrementally hundreds of times a day. And there's good reason for that: it's safer overall. Incremental deploys are easier to understand and fix than one gigantic deploy once a year. But it lends itself to those small bugs, which, in this permissions case, are unacceptable.

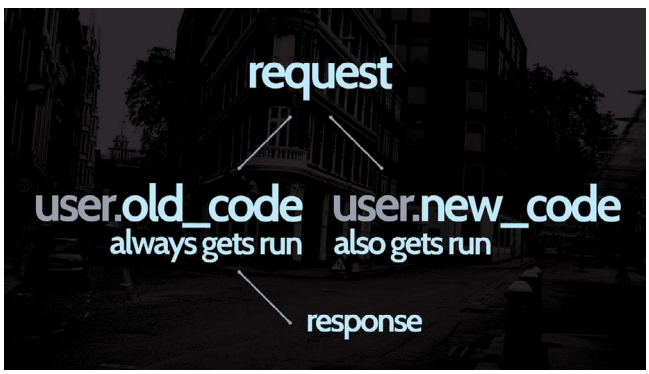
So tests are good to have. This is unsurprising to say in this day and age; everyone generally understands now that testing and continuous integration are absolutely critical to software development. But that's not what's at stake here. You can have the best, most comprehensive test suite in the world, but *tests are still different from production*.

There are a lot of reasons for this. One is data: you may have flipped some bit (accidentally or intentionally) for some tables for two weeks back in December of 2010, and you've all but forgotten about that today. Or your cognitive model of the system may be idealized. We noticed that while doing our permissions overhaul. We'd have a nice, neat table of all the permissions of users, organizations, teams, public and private repositories, and forks, but we'd notice that the neat table would fall down on very arcane edge cases once we looked at production data.

And that's the rub: you need your tests to pass, of course, but you also need to verify that you don't change production behavior. Think of it as another test suite: for better or worse, the behavior deployed now is the state of the system from your users' perspective. You can then either fix the behavior or update your tests; just make sure you don't break the user experience.

Parallel Code Paths

One of the approaches we've taken is through the use of **parallel code paths.**



What happens is this: a request will come in as usual and run the existing (old) code. At the same time (or just right after it executes), we'll also run the new code that we think will be better/faster/harder/stronger (pick one). Once all that's done, return whatever the existing (old) code returns. So, from the user's perspective, nothing has changed. They don't see the effects of the new code at all.

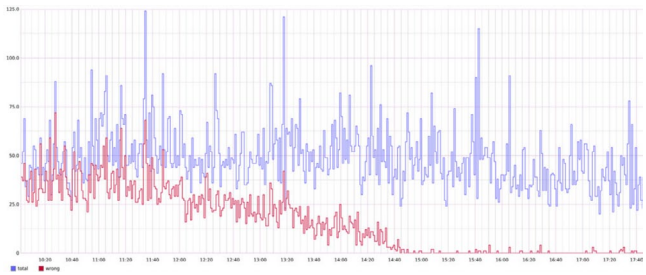
There are some caveats, of course. In this case, we're typically performing read-only operations. If we're doing writes, it takes a bit more smarts to either write your code to make sure it can run both branches of code safely, or you can rollback the effects of the new code, or the new code is a no-op or otherwise goes to a different place entirely. Twitter, for example, has a very service-oriented architecture, so if they're spinning up a new service they redirect traffic and dual-write to the new service so they can measure performance, accuracy, catch bugs, and then throw away the redundant data until they're ready to switch over all traffic for real.

We wrote a Ruby library named Science to help us out with this. You can check it out and run it yourself in the github/dat-science repository. The general idea would be to run it like this:

```
science "my-cool-new-change" do |e|
  e.control   { user.existing_slow_method }
  e.candidate { user.new_code_we_think_is_
great }
end
```

It's just like when you Did Science™ in the lab back in school growing up: you have a control, which is your existing code, and a candidate, which is the new code you want to introduce. The `science` block makes sure both are run appropriately. The real power happens with what you can do after the code runs, though.

We use Graphite literally all over the company. If you haven't seen Coda Hale's Metrics, Metrics Everywhere talk [hn.my/metrics], do yourself a favor and give it a watch. Graphing behavior of your application gives you a ton of insight into your entire system.



Attempts vs. mismatches

Science (and its sister library, github/dat-analysis) can generate a graph of the number of times the code was run (the top blue bar to the left) and compare it to the number of mismatches between the control and the candidate (in red, on the bottom). In this case you see a downward trend: the developer saw that their initial deploy might have missed a couple use cases, and over subsequent deploys and fixes the mismatches decreased to near-zero, meaning that the new code is matching production's behavior in almost all cases.

75th and 99th percentile performance

What's more, we can analyze performance, too. We can look at the average duration of the two code blocks and confirm if the new code we're running is faster, but we can also break down requests by percentile. In the slide to the right, we're looking at the 75th and 99th percentile, i.e. the slowest of requests. In this particular case, our candidate code is actually quite a bit slower than the control: perhaps this is acceptable given the base case, or maybe this should be huge red sirens that the code's not ready to deploy to everyone yet...it depends on the code.

All of this gives you evidence to prove the safety of your code before you deploy it to your entire userbase. Sometimes we'll run these experiments for weeks or months as we widdle down all the (sometimes tricky) edge cases. All the while, we can deploy quickly and iteratively with a pace we've grown accustomed to, even on dicey code. It's a really nice balance of speed and safety.

Build Into Your Existing Process

Something else I've been thinking about a lot lately is how your approach to building product is structured.

Typically process is added to a company vertically. For example: say your team's been having some problems with code quality. Too many bugs have been slipping into production. What a bummer. One way to address that is to add more process to your process. Maybe you want your lead developers to review every line of code before it gets merged. Maybe you want to add a layer of human testing before deploying to production. Maybe you want a code style audit to give you some assurance of new code maintainability.

These are all fine approaches, in some sense. It's not problematic to want to achieve the goal of clean code; far from it, in fact. But I think this vertical layering of process is really what can get aggravating or just straight-up confusing if you have to deal with it day in, day out.

I think there's something to be said for scaling the breadth of your process. It's an important distinction. By limiting the number of layers of process, it becomes simpler to explain and conceptually understand (particularly for new employees). "Just check continuous integration" is easier to remember than "push your code, ping the lead developers, ping the human testing team, and kick off a code standards audit."

We've been doing more of this lateral process scaling at GitHub informally, but I think there's more to it than even we initially noticed. Since continuous integration is so critical for us, people have been adding more tests that aren't necessarily tests in the classic sense of the word. Instead of "will this code break the application?", our tests are more and more measuring "will this code be maintainable and more resilient towards errors in the future?"

For example, here are a few tests we've added that don't necessarily have user-facing impact but are considered breaking the build if they go red:

- Removing a CSS declaration without removing the associated `class` attribute in the HTML
- ...and vice versa: removing a `class` attribute without cleaning up the CSS
- Adding an `<img>` tag that's not on our CDN, for performance, security, and scaling reasons
- Invalid SCSS or CoffeeScript (we use SCSS-Lint and CoffeeLint)

None of these are world-ending problems: an unspecified HTML class doesn't really hurt you or your users. But from a code quality and maintainability perspective, yeah, it's a big deal in the long term. Instead of having everyone focus on spotting these during code review, why not just shove it in CI and let computers handle the hard stuff? It **frees our coworkers up from gruntwork** and lets them focus on what really matters.

Incidentally, some of these are super helpful during refactoring. Yesterday I shipped some new dashboards on github.com, so today I removed the thousands of lines of code from the old dashboard code. I could remove the code in bulk, see which tests fail, and then go in and pretty carelessly remove the now-unused CSS. Made it much, much quicker to do because I didn't have to worry about the gruntwork.

And that's what you want. You want your coworkers to *think less about bullshit that doesn't matter and spend more consideration on things that do.* Think about consolidating your process. Instead of layers, ask if you can merge them into one meeting. Or one code review. Or automate the need away entirely. The layers of process are what get you.

Process

In bigger organizations, the number of people that need to be involved in a product launch grows dramatically. From the designers and developers who actually build it, to the marketing team that tells people about it, to the ops team who scales it, to the lawyers that legalize it™... there are a lot of chefs in the kitchen. If you're releasing anything that a lot of people will see, there's a lot you need to do.

Coordinating that can be tricky.

Apple's an interesting company to take a look at. Over time, a few interesting tidbits have spilled out of Cupertino. The Apple New Product Process (ANPP) is, at its core, a simple checklist. It goes into great detail about the process of releasing a product, from beginning to end, from who's responsible to who needs to be looped into the process before it goes live.

The ANPP tends to be at the very high-level of the company (think Tim Cook-level of things), but this type of approach sinks deeper down into individual small teams. Even before a team starts working on something, they might make a checklist to prep for it: do they have appropriate access to development and staging servers, do they have the correct people on the team, and so on. And even though they manage these processes in custom-built software, what it is at its core is simple: it's a checklist. When you're done with something, you check it off the list. It's easy to collaborate on, and it's easy to understand.

Think back to every single sci-fi movie you've ever watched. When they're about to launch the rocket into space, there's a lot of "Flip `MAIN SERIAL BUS A` to *on*." And then the dutiful response: "Roger, `MAIN SERIAL BUS A` is *on*." You don't see many responses of, "uh, Houston, I think I'm more happier when `MAIN SERIAL BUS A` is at like, 43% because `SECONDARY SERIAL BUS B` is kind of a jerk sometimes and I don't trust goddamn serial busses what the hell is a serial bus anyway yo Houston hook a brother up with a serial limo instead."

And there's a reason for that: checklists remove ambiguity. All the debate happens before something gets added to the checklist... not at the end. That means when you're about to launch your product — or go into space — you should be worrying less about the implementation and rely upon the process more instead. Launches are stressful enough as-is.

Ownership

Something else that becomes increasingly important as your organization grows is that of code ownership. If the goal is to have clean, relatively bug-free code, then your process should help foster an environment of responsibility and ownership of your piece of the codebase.

If you break it, you should fix it.

At GitHub, we try to make that connection pretty explicit. If you're deploying your code and your code generates a lot of errors, our open source chatroom robot, Hubot, [hubot.github.com] will notice and message you in chat with a friendly "hey, you were the last person to deploy and something is breaking. Can you take a look at it?" This reiterates the idea that you're responsible for the code that you put out. That's good because, as it turns out, **the people who wrote the code are typically the people who can most easily fix it.** Beyond that, forcing your coworkers to always clean up your mess is going to really suck over time (for them).

There are plenty of ways to keep people responsible. Google, for example, uses `OWNERS` files in Chrome. This is a way of making explicit the ownership of a file or entire directories of the project. The format of an actual `OWNERS` file can be really simple — shout out to simple systems like flat files and checklists — but they serve two really great purposes:

- They **enforce quality**. If you're an owner of an area of code, any new contribution to your code requires your signoff. Since you are in a somewhat elevated position of responsibility, it's on you to fight to not allow potentially buggy code into your area.
- It **encourages mentorship**. Particularly in open source projects like Chromium, it can be intimidating to get started with your first contribution. `OWNERS` files make it explicit about who you might want to ask about your code or even about the high-level discussion before you get started.

You can tie your own systems together closer, too. In Haystack, our internal error tracking service at GitHub, we have pretty deep hooks into our code itself. In a controller, for example, we might have code that looks like this:

```
class BranchesController
  areas_of_reponsibility :git
end
```

This marks this particular file as being the responsibility of the `@github/git` team, the team that handles Git-related data and

infrastructure. So, when we see a graph in Haystack like the one to the right, we can see that there's something breaking in a particular page. We can quickly see which teams are responsible for the code that's breaking, since Haystack knows to look into the file with the error and bubble up these areas of responsibility. From here, it's a one-click operation to open an issue on our bug tracker about it, mentioning the responsible team in it so they can fix it.

Look: bugs do happen. Even *if* you move fast and break nothing, well, you're still bound to break something at some point. Having a culture of responsibility around your code helps you address those bugs quickly in an organized manner.

Talking & Communicating

I've given a lot of talks and written a lot of blog posts about software development and teams and organizations. Probably one way to sum them all up is: *more communication*. I think companies function better by being more transparent, and if you build your environment correctly, you can end up with better code, a good remote work culture, and happier employees.

But god, more communication means a ton of shit. Emails. Notifications. Text messages. IMs. Videos. Meetings.

If everyone is involved with everything…does anyone really have enough time to actually do anything?

Having more communication is good. Improving your communication is even better.

Be Mindful Of Your Coworker's Time

It's easy to feel like you deserve the time of your coworkers. In some sense, you do: you're trying to improve some aspect of the company, and if your coworker can help out with that, then the whole company is better off. But every interaction comes with a cost: your coworker's time. This is dramatically more important in creative and problem solving fields like computer science, where being in the zone can mean the difference between a really productive day and a day where every line of code is a struggle to write. Getting pulled out of the zone can be jarring, and getting back into that mental mindset can take a frustratingly long time.

This goes doubly so for companies with remote workers. It's easy to notify a coworker through chat or text message or IM that you need their help with something. Maybe a server went down, or you're having a tough problem with a bug in code you're unfamiliar with. If you're a global company, time zones can become a factor, too. I was talking to a coworker about this, and after enough days of being on-call, she came up with a hilarious idea that I love:

- You find you need help with something.
- You page someone on your team for help.
- They're sleeping. Or out with their kids. Or any level of "enjoying their life."
- They check their message and, in doing so, their phone takes a selfie of them and pastes it into the chat room.
- You suddenly feel worse.

We haven't implemented this yet (and who knows if we will), but it's a pretty rad thought experiment. If you could see the impact your actions on your coworker's life, would it change your behavior? Can you build something into your process or your tools that might help with this? It's interesting to think about.

I think this is part of a greater discussion on empathy. And empathy comes in part from seeing real pain. This is why many suggest that developers handle some support threads. A dedicated support team is great, but until you're actually faced with problems up-close, it's easy to avoid these pain points.

Institutional Teaching

> *We have a responsibility to be teachers — that this should be a central part of [our] jobs…it's just logic that someday we won't be here.*
> *— Ed Catmull, co-founder of Pixar*

I really like this quote for a couple reasons. For one, this can be taken literally: we're all going to fucking die. Bummer, right? Them's the breaks, kid.

But it also means that people move around. Sometimes people will quit or get fired from the company, and sometimes it just means people moving around the company. The common denominator is that our presence is merely temporary, which means we're obligated, in part, to spread the knowledge we have across the company. This is great for your bottom line, of course, but it's also just a good thing to do. Teaching people around you how to progress in their careers and being a resource for their own growth is a very privileged position to be in, and one we shouldn't take lightly.

So how do we share knowledge without being lame? I'm not going to lie: part of the reason I'm working now is because I don't have to go to school anymore. Classes are so *dullllllllll*. So the last thing I want to have to deal with is some formal, stuffy process that ultimately doesn't even serve as a good foundation to teach anyone anything.

Something that's grown out of how we work is a concept we call "ChatOps". @jnewland has a really great talk [hn.my/chatops] about the nitty-gritty of ChatOps at GitHub, but in short: it's a way of handling devops and systems-level work at your company in front of others so that problems can be solved and improved upon collaboratively.

If something breaks at a tech company, a traditional process might look something like this:

1. Something breaks.
2. Whoever's on-call gets paged.
3. They SSH into... something.
4. They fix it... somehow.

There's not a lot of transparency. Even if you discuss it after the fact, the process that gets relayed to you might not be comprehensive enough for you to really understand what's going on. Instead, GitHub and other companies have a flow more like this:

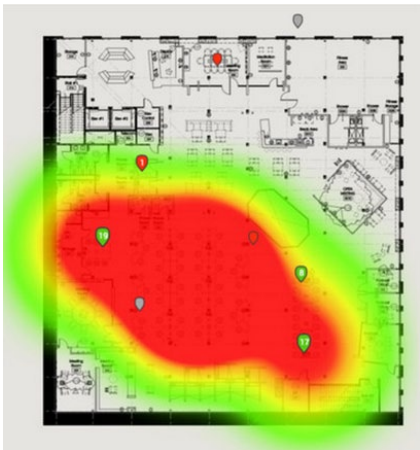
1. Something breaks.
2. Whoever's on-call gets paged.
3. They gather information in a chat room.
4. They fix it through shared tooling, in that chat room, in front of (or leveraging the help of) other employees.

This brings us a number of benefits. For one, you can **learn by osmosis**. I'm not on the Ops team, but occasionally I'll stick my head into their chat room and see how they tackle a problem, even if it's a problem I won't face in my normal day-to-day work. I gain context around how they approach problems.

What's more, if others are studying how they tackle a problem in real-time, the **process lends itself to improvement**. How many times have you sat down to pair program with someone and were blown away by the one or two keystrokes they use to solve a process that takes you three minutes? If you code in a vacuum, you don't have the opportunity to make quick improvements. If I'm watching you run the same three commands in order to run diagnostics on a server, it's easier as a bystander to think, hey, why don't we wrap those commands up in one command that does it all for us? Those insights can be incredibly valuable and, in time, lead to massive, massive productivity and quality improvements.

This requires some work on tooling, of course. We use Hubot to act as a sort of shared collection of shell scripts that allow us to quickly address problems in our infrastructure. Some of those scripts include hooks into PagerDuty [pagerduty.com] to trigger pages, code that leverages APIs into AWS or our own datacenter, and, of course, scripts that let us file issues and work with our repositories and teams on GitHub. We have hundreds or thousands of commands now, all gradually built up and hardened over time. The result is an incredible amount of tooling around automating our response to potential downtime.

This isn't limited to just working in chatrooms, though. Recently the Wi-Fi broke at our office on a particular floor. We took the same approach to fix it, except it was in a GitHub issue instead of real-time chat. Our engineers working on the problem pasted in screenshots of the status of our routers, the heatmaps of dead zones stemming from the downtime, and eventually traced cables through switches until we found a faulty one, taking photos each step of the way and adding them to the issue so we had a paper trail of which cabinets and components were affected. It's amazing how much you can learn from such a non-invasive process. If I'm not interested in learning the nitty-gritty details, I can skip the thread. But if I do want to learn about it... it's all right there, waiting for me.



Feedback

The Blue Angels are a United States Navy demonstration flight squadron. They fly in air shows around the world, maneuvering in their tight six-fighter formations 18 inches apart from one another. The talent they exhibit is mind-boggling.

Earlier this year I stumbled on a documentary on their squadron from years back. There's a specific 45-second section in it that really made me think. It describes the process the Blue Angels go through in order to give each other feedback.

So first of all, they're obviously, patently, completely nuts. The idea that you can give brutally honest feedback without worrying about interpersonal relationships is, well, not really relevant to the real world. They're superhuman. It's not every day you can tell your boss that she fucked up and skip out of the meeting humming your favorite tune without fear of repercussions. So they're nuts. But it does make sense: a mistake at their speeds and altitude is almost certainly fatal. A mistake for us, while writing software that helps identify which of your friends liked that status update about squirrels, is decidedly less fatal.

But it's still a really interesting *ideal* to look up to. They do feedback and retrospectives that take twice as long as the actual event itself. And they take their job of giving and receiving feedback seriously. How can we translate this idealized version of feedback in our admittedly-less-stressful gigs?

Part of this is just **getting better at receiving feedback.** I'm fucking horrible at this. You do have to have a bit of a thicker skin. And it sucks! No one wants to spend a few hours — or days, or *months* — working on something, only to inevitably get the drive-by commenter who finds a flaw in it (either real or imagined). It's sometimes difficult to not take that feedback personally. That you failed. That you're not good enough to get it perfect on the first or second tries. It's funny how quickly we forget how iterative software development is, and that computers are basically stacking the deck against us to never get anything correct on the first try.

Taking that into account, though, it becomes clear **how important giving good feedback** is. And sometimes this is just as hard to do. I mean, someone just pushed bad code! To *your* project! To your code! I mean, you're even in the damn `OWNERS` file! The only option is to rain fire and brimstone and hate and loathing on this poor sod, the depths of which will cause him to surely think twice about committing such horrible code and, if you're lucky, he'll quit programming altogether and become a dairy farmer instead. Fuck him!

Of course, this isn't a good approach to take. Almost without fail, if someone's changing code, *they have a reason for it*. It may not be a *good* reason, or the implementation might be suspect, but it's reason nonetheless. And being cognizant of that can go a long way towards pointing them in the right direction. How you piece your words together is terribly important.

And this is something you should at least think about, if not explicitly codify across your whole development team entirely. What do you consider good feedback? How can you promote understanding and positive approaches in your criticism of the code? How can you help the submitter learn and grow from this scenario? Unfortunately these questions don't get asked enough, which creates a self-perpetuating cycle of cynics and aggressive discussion.

That sucks. Do better.

Move Fast With A Degree Of Caution

Building software is hard. Because yeah, moving quickly means you can do more for people. It means you can see what works and what doesn't work. It means your company sees real progress quicker.

But sometimes it's just as important to know what not to break. And when you work on those things, you change how you operate so that you can try to get the best of both worlds.

Also, try flying fighter jets sometimes. It can't be that hard. ■

Zach joined GitHub in 2010 as one of their first engineering hires. Initially working on what would become GitHub Enterprise, he now hacks on new features and frequently gives talks about building products and growing startups. He also writes about public speaking on *speaking.io*

Reprinted with permission of the original author. First appeared in *hn.my/mfbn* (zachholman.com)

Photo: Cock the Hammer by Kyle May [flickr.com/photos/kylemay/1430449350]

Java for Everything

By LAWRENCE KESTELOOT

I used to ask interviewees, "What's your favorite programming language?" The answer was nearly always, "I just choose the right language for the job." Duh. Does anyone ever deliberately pick the wrong language? This was clearly a way to avoid actually naming a language for fear of picking one I didn't like.

If the interviewee gave an answer at all, it was, "I'm most familiar with language X," which didn't answer my question either.

At the time I would myself have replied something like, "I like Python best because it makes me happy to program in it, but I only use it in such-and-such a situation. The rest of the time I use XYZ…"

About a year ago, though, I started to form a strange idea: That Java is the right language for all jobs. (I pause here while you vomit in your mouth.) This rests on the argument that what you perceive to be true does not match reality, and that's never a popular approach, but let me explain anyway.

Python really is my favorite language, and it truly makes me happy when I code in it. It pushes the happy spot in my head. It matches pseudo-code so well that it's a genuine pleasure to work in it.

Years ago I read Bruce Eckel's influential Strong Type vs. Strong Testing. [hn.my/strong] In it he argued that static typing (what he calls strong typing) is one of the many facets of program correctness, and that if you're going to check the other facets (such as the algorithm and the logic) with unit tests, then the types will also get checked, so you may as well go for dynamic typing and benefit from its advantages.

Bruce used Python to illustrate his code, and that clinched it: I decided that I would from then on write everything in Python. Unfortunately I was half-way through a large Java program at work, but my co-worker and I agreed that it should have been written in Python, and perhaps one day we'd get a good excuse to rewrite it all that way.

Several things changed my mind 180° in less than a year:

- At one company I wrote a simulator that allowed me to run my Java services without a fully-functional site. In this simulator I ran scripts that tested various scenarios including failures. For these scripts I decided to use JavaScript, primarily because it's included in Java 6 and secondarily because many people know it. I reasoned that a scripting language would allow us and Q/A to write tests easily. An intern, Justin Lebar, argued that we should simply use Java. The simulator is in Java, so why not write the scripts in it, too? It's sitting right there and we all know it. I went ahead with JavaScript, which forced me to write various code to bridge the two. It also meant that stack traces were much harder to read, since they didn't point to the line in the script that was being executed. Q/A never wrote any tests. Overall we gained nothing from JavaScript and Justin had been right.
- At the same company we stored our logs in JSON format (which is a great idea, by the way), and a co-worker wrote a Python program called logcat to parse the logs and generate the standard columnar output, with many nice features and flag (including a binary search for timestamp). On OurGroceries, my personal project, we needed something similar and I suggested again to use Python. My partner Dan Collens suggested Java, since it's right there and we know it and it's fast. He wrote it and he was right: it's blazing fast. I've since compared the Python logcat to a Java one and the latter is about ten times faster. Whatever time was saved by the developer when writing the Python code (if any) were lost many times over as dozens of users had to wait ten times longer each time they fished through the logs.
- And finally, I went to write a simple program that put up a web interface. I considered using Python, but this would have required me to figure out how to serve pages from its library. I had already done this in Java (with Jetty), so I could be up and running in Java in less time. I realized that as I accumulate knowledge about 3rd party Java libraries and grow my own utility library, it becomes increasingly expensive to use any other language. I have to figure those things out again and write them again, instead of copying and pasting the code from the previous project. Note that this doesn't argue for Java, but it does argue for using a single language.

The big argument against Java is that it's verbose. Perhaps, but so what? I suppose the real argument is that it takes longer to write the code. I doubt this is very much true after the first 10 minutes. Sure you have to write public static void main, but how much time does that take? Sure you have to write:

```
Map<String,User> userIdMap =
new HashMap<String,User>();
```

instead of:

```
userIdMap = {}
```

but in the bigger scheme of things, is that so long? How many total minutes out of a day is that, two? And in Python the code more realistically looks like this anyway:

```
# Map from user ID to User
object.
userIdMap = {}
```

(If it doesn't, then you have bigger problems. Undocumented Python programs are horrendously difficult to maintain.) The problem is that programmers perceive mindless work as painful and time-consuming, but the reality is never so bad. Here's a quote from a forum about language design:

It really sucks when you have to add type declarations for blindingly obvious things, e.g. Foo x = new Foo(). — @pazsxn

No, actually, typing `Foo` one extra time does not "really suck." It's three letters. The burden is massively overstated because the work is mindless, but it's really pretty trivial. Programmers will cringe at writing some kind of command dispatch list:

```
if command = "up":
    up()
elif command = "status":
    status()
elif command = "revert":
    revert()
...
```

so they'll go off and write some introspecting auto-dispatch cleverness, but that takes longer to write and will surely confuse future readers who'll wonder how the heck `revert()` ever gets called. Yet the programmer will incorrectly feel as though he saved himself time. This is the trap of the dynamic language. It feels like you're being more productive, but aside from the first 10 minutes of a new program, you're not. Just write the stupid dispatch manually and get on with the real work.

So why are dynamic languages ever chosen? If you and I have a contest to write a simple blogging system and you're using (say) Python, you'll have something interesting in 30 minutes using pickling and whatnot, and it'll take me two days to build something with MySQL. Many language choices are based on trivial contests like these. But after two weeks of development, when we both have to add a feature, mine will take at most as long as yours, and I won't be spending any time figuring out how to get my system to handle so many users, or tracking down why some obscure if clause breaks because you misspelled the name of a function, or figuring out what the heck this request parameter contains.

The classic hacker disdain for "bondage and discipline languages" is short sighted; the needs of large, long-lived, multi-programmer projects are just different than the quick work you do for yourself.

And you don't think you'll struggle with scalability sooner than I? Every year the NaNoWriMo website goes down on October 31st. It's unresponsive for days. About 60,000 people hit it over a period of several hours, so maybe four requests per second. It's written in PHP. The OurGroceries backend is written in Java. It handles (currently) about 50 complex requests per second and the CPU rarely goes above 1% for the Java process.

Twitter tripled their search speed by switching their search engine from Ruby to Java.

A few years earlier, Joel Spolsky tweeted:

Digg: 200MM page views, 500 servers. Stack Overflow: 60MM page views, 5 servers. What am I missing?

The reply from @GregB was:

That's the PHP factor.

StackOverflow uses ASP.NET. So you can complain all day about `public static void main`, but have fun setting up 500 servers. The downsides of dynamic languages are real, expensive, and permanent.

And what about the unit testing argument? If you have to unit test your code anyway, what does static typing buy you? Well for one thing it buys you speed, and lots of it. But also writing and maintaining unit tests takes time. Most importantly, the kinds of bugs that people introduce most often aren't the kind of bugs that unit tests catch. With few exceptions (such as parsers), unit tests are a waste of time. To quote a friend of mine, "They're a tedious, error-prone way of trying to recapture the lost value of static type annotations, but in a bumbling way in a separate place from the code itself."

So here's my new approach: *Do everything in Java*. Don't be tempted to write some quick hack in Python because:

- You can't copy and paste code from other projects in your primary programming language.
- It may feel faster to develop, but that's an illusion. The actual time saved is small, though admittedly annoying.
- It's one more language, platform, and set of libraries that I and my co-workers have to learn and master.
- And here's the important one: Chances are good that this quick hack will grow and become an important tool, and I won't have the bandwidth to rewrite it, yet I'll suffer the performance and maintenance penalty every time I use it.

I agree it's fun to develop in Python. I love it. When I'm writing a Sudoku solver, I reach for Python. But it's the wrong tool for anything larger, and it's the wrong tool for code of any size written for pay, because you're doing your employer a disservice.

I'm even taking this to an extreme and using Java for shell scripts. I've found that anything other than a simple wrapper shell script eventually grows to the point where I'm looking up the arcane syntax for removing some middle element from an array in bash. What a crappy language! Wrong tool for the job! Write it in Java to start with. If shelling out to run commands is clumsy, write a utility function to make it easy.

I've also written a java_launcher shell script that allows me to write this at the top of Java programs:

```
#!/usr/bin/env java_launcher
# vim:ft=java
# lib:/home/lk/lib/teamten.jar
```

I can make the Java programs executable and drop the `.java` extension. The script strips the header, compiles and caches the class file, and runs the result with the specified jars. It provides one of the big advantages of Python: the lack of build scripts for simple one-off programs.

This focus on a single language has had an interesting effect: It has encouraged me to improve my personal library of utility functions [github.com/lkesteloot/teamten] (teamten.jar above), since my efforts are no longer split across several languages. For example, I wrote a library that contains image processing routines. They're both faster and higher quality than anything you can find in Java and Python. This took a while, but I know it's worth it because I won't find myself writing some Python script and wishing I could resize an image nicely. I can now confidently invest in Java as an important part of my professional and personal technical future.

There remains the question of why choosing Java specifically, out of the set of compiled statically-typed languages. The advantages of C and C++ (slight performance gains, embeddability, graphics libraries) don't apply to my work. C# is nice but not cross-platform enough. Scala is too complex. And other languages like D and Go are too new to bet my work on.

When I tell people that I now write everything in Java, they look horrified. One friend had a visible look of disgust. But you know, Java's a pretty nice language, and when my code compiles, which is often the first time, it'll usually also run correctly. I don't have that peace of mind with any other language. Java just works like a horse and is useful across a very broad range of applications. ■

Lawrence Kesteloot writes software in San Francisco. He has worked for DreamWorks and various start-ups and is now making and selling mobile apps for himself.

Reprinted with permission of the original author.
First appeared in *hn.my/java* (teamten.com)

The Case for Slow Programming

By JEFFREY VENTRELLA

My dad used to say, "Slow down, son. You'll get the job done faster."

I've worked in many high-tech startup companies in the San Francisco Bay area. I am now 52, and I program slowly and thoughtfully. I'm kind of like a designer who writes code; this may become apparent as you read on.

Programming slowly was a problem for me when I recently worked on a project with some young coders who believe in making really fast, small iterative changes to the code. At the job, we were encouraged to work in the same codebase, as if it were a big cauldron of soup, and if we all just kept stirring it continuously and vigorously, a fully-formed thing of wonder would emerge.

It didn't.

Many of these coders believed in the fallacy that all engineers are fungible, and that no one should be responsible for any particular aspect of the code; any coder should be able to change any part of the code at any time. After all, we have awesome services like GitHub to manage and merge any number of asynchronous contributions from any number of coders. As long as everyone makes frequent commits, and doesn't break anything, everything will come out just fine.

Bullshit.

You can't wish away Design Process. It has been in existence since the dawn of civilization. And the latest clever development tools, no matter how clever, cannot replace the best practices and real-life collaboration that built cathedrals, railroads, and feature-length films.

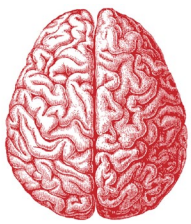
Nor can any amount of programming ever result in a tool that reduces the time of software development to the speed at which a team of code monkeys can type.

Dysrhythmia

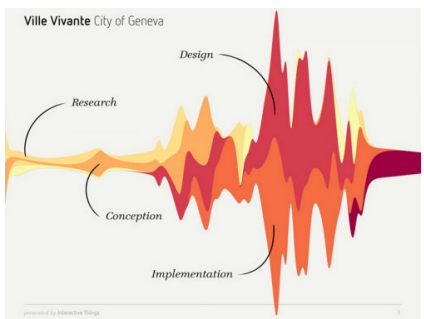
The casualty of my being a slow programmer among fast programmers was a form of dysrhythmia — whereby my coding rhythm got aliased out of existence by the pummeling of other coders' machine gun iterations. My programming style is defined by organic arcs of different sizes and timescales, Each arc starts with exploration, trial and error, hacks, and temporary variables. Basically, a good deal of scaffolding. A picture begins to take shape. Later on, I come back and dot my i's and cross my t's. The end of each arc is something like implementation-ready code. ("Cleaning my studio" is a necessary part of finishing the cycle). The development arc of my code contribution is synonymous with the emergence of a strategy, a design scheme, an architecture.

And sometimes, after a mature organism has emerged, I go back and start over, because I think I have a better idea of how to do it. Sometimes I'm wrong. Sometimes I'm right. There is no way to really know until the organism is fully formed and staring me in the face.

Anyway, back to the cauldron-soup-programmers. The problem is this: with no stasis in the overall software ecosystem — no pools of stillness within which to gain traction and apply design process, how can anyone, even a fast coder, do good design?



Any coder who claims that fast programming is the same as slow programming (except that it's fast), doesn't understand Design Process. For the same reason that many neuroscientists now believe that the fluid-like flow of neuronal firing throughout the brain has a temporal reverberation which has everything to do with thought and consciousness, good design takes time.



The Slow Programming Movement

According to Wikipedia: "The slow programming movement is part of the slow movement. It is a software development philosophy that emphasizes careful design, quality code, software testing and thinking. It strives to avoid kludges, buggy code, and overly quick release cycles.

Wikipedia also says this about "Slow Software Development": "As part of the agile software development movement, groups of software developers around the world look for more predictive projects, and aiming at a more sustainable career and work-life balance. They propose some practices such as pair programming, code reviews, and code refactorings that result in more reliable and robust software applications."

Venture-backed software development here in the San Francisco Bay area is on a fever-pitch fast-track. Money dynamics puts unnatural demands on a process that would be best left to the natural circadian rhythms of design evolution. Fast is not always better. In fact, slower sometimes actually means faster — when all is said and done. The subject of how digital technology is usurping our natural temporal rhythm is addressed in Rushkoff's Present Shock.

There's another problem: the almost religious obsession with technology — and a fetish-like love for tools. People wonder why software sucks (and yes, it sucks). Software sucks because of navel-gazing. Fast programmers build hacky tools to get around the hacky tools that they built to get around the hacky tools that they built to help them code.

This is why I believe that we need older people, women, and educators INSIDE the software development cycle. More people-people, fewer thing-people. And I don't mean on the outside, sitting at help desks or doing UI flower arranging. I mean on the INSIDE — making sure that software resonates with humanity at large.

I'm Glad I'm Not a Touch-Typist.

A friend of mine who is a mature, female software engineer made an interesting quip: "software programming is not typing." Everyone knows this, but it doesn't hurt to remind ourselves every so often. Brendan Enrick discusses this. The fact that we programmers spend our time jabbing our fingers at keyboards makes it appear that this physical activity is synonymous with programming. But programming is actually the act of bringing thought, design, language, logic, and mental construction into a form that can be stored in computer memory.

My wife often comes out into the yard and asks me: "are you coding?" Often my answer is "yes." Usually I am cutting twigs with a garden clipper or moving compost around.

Plants, dirt, and clippers have just as much to do with programming as keyboards and glowing screens.

We are transitioning from an industrial age and an economic era defined by growth to an age of sustainability. Yes, new software and new businesses need to grow. But to be sustainable, they need to grow slowly and with loving care. Like good wine. Like a baby. ■

Jeffrey Ventrella is an artist/programmer who lives in the San Francisco Bay area. A graduate of the MIT Media Lab, Jeffrey founded Wiggle Planet, LLC to develop autonomous characters with augmented reality. Jeffrey has presented and published works on artificial life, virtual worlds and computer art internationally.

Reprinted with permission of the original author.
First appeared in *hn.my/slowp* (ventrellathing.wordpress.com)

The Unreasonable Effectiveness of C

By DAMIEN KATZ

For years I've tried my damnedest to get away from C. Too simple, too many details to manage, too old and crufty, too low level. I've had intense and torrid love affairs with Java, C++, and Erlang. I've built things I'm proud of with all of them, and yet each has broken my heart. They've made promises they couldn't keep, created cultures that focus on the wrong things, and made devastating tradeoffs that eventually make you suffer painfully. And I keep crawling back to C.

C is the total package. It is the only language that's highly productive, extremely fast, has great tooling everywhere, a large community, a highly professional culture, and is truly honest about its tradeoffs.

Other languages can get you to a working state faster, but in the long run, when performance and reliability are important, C will save you time and headaches. I'm painfully learning that lesson once again.

Simple and Expressive

C is a fantastic high level language. I'll repeat that. C is a fantastic **high level language.** It's not as high level as Java or C#, and certainly nowhere near as high level as Erlang, Python, or JavaScript. But it's as high level as C++, and far, far simpler. Sure, C++ offers more abstraction, but it doesn't present a high level of abstraction away from C. With C++ you still have to know everything you knew in C, plus a bunch of other ridiculous shit.

> *"When someone says: 'I want a programming language in which I need only say what I wish done,' give him a lollipop."*
> *— Alan J. Perlis*

That we have a hard time thinking of lower level languages we'd use instead of C isn't because C is low level. It's because C is so damn successful as an abstraction over the underlying machine and making that high level, it's made most low level languages irrelevant. C is that good at what it does.

The syntax and semantics of C is amazingly powerful and expressive. It makes it easy to reason about high level algorithms and low level hardware at the same time. Its semantics are so simple and the syntax so powerful it lowers the cognitive load substantially, letting the programmer focus on what's important.

It's blown everything else away to the point it's moved the bar and redefined what we think of as a low level language. That's damn impressive.

Simpler Code, Simpler Types

C is a weak, statically typed language and its type system is quite simple. Unlike C++ or Java, you don't have classes where you define all sorts of new runtime behaviors of types. You are pretty much limited to structs and unions and all callers must be very explicit about how they use the types, callers get very little for free.

> *"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."*
> *— Joe Armstrong*

What sounds like a weakness ends up being a virtue: the "surface area" of C APIs tend to be simple and small. Instead of massive frameworks, there is a strong tendency and culture to create small libraries that are lightweight abstractions over simple types.

Contrast this to OO languages where codebases tend to evolve massive interdependent interfaces of complex types, where the arguments and return types are more complex types and the **complexity is fractal**, each type is a class defined in terms of methods with arguments and return types or more complex return types.

It's not that OO type systems force fractal complexity to happen, but they encourage it, they make it easier to do the wrong thing. C doesn't make it impossible, but it makes it harder. C tends to breed simpler, shallower types with fewer dependencies that are easier to understand and debug.

Speed King

C is the fastest language out there, both in micro and in full stack benchmarks. And it isn't just the fastest in runtime, it's also consistently the most efficient for memory consumption and startup time. And when you need to make a tradeoff between space and time, C doesn't hide the details from you, it's easy to reason about both.

> *"Trying to outsmart a compiler defeats much of the purpose of using one."*
> *— Kernighan & Plauger, The Elements of Programming Style*

Every time there is a claim of "near C" performance from a higher level language like Java or Haskell, it becomes a sick joke when you see the details. They have to do awkward backflips of syntax, use special knowledge of "smart" compilers and VM internals to get that performance, to the point that the simple expressive nature of the language is lost to strange optimizations that are version specific, and usually only stand up in micro-benchmarks.

When you write something to be fast in C, you know why it's fast, and it doesn't degrade significantly with different compilers or environments the way different VMs will, the way GC settings can radically affect performance and pauses, or the way interaction of one piece of code in an application will totally change the garbage collection profile for the rest.

The route to optimization in C is direct and simple, and when it's not, there are a host of profiler tools to help you understand why without having to understand the guts of a VM or the "sufficiently smart compiler." When using profilers for CPU, memory and IO, C is best at not obscuring what is really happening. The benchmarks, both micro and full stack, consistently prove C is still the king.

Faster Build-Run-Debug Cycles

Critically important to developer efficiency and productivity is the "build, run, debug" cycle. The faster the cycle is, the more interactive development is, and the more you stay in the state of flow and on task. C has the fastest development interactivity of any mainstream statically typed language.

> *"Optimism is an occupational hazard of programming; feedback is the treatment."*
> *— Kent Beck*

Because the build, run, debug cycle is not a core feature of a language, it's more about the tooling around it, this cycle is something that tends to be overlooked. It's hard to overstate the importance of the cycle for productivity. Sadly it's something that gets left out of most programming language discussions, where the focus tends to be only on lines of code and source writability/readability. The reality is the tooling and interactivity cycle of C is the fastest of any comparable language.

Ubiquitous Debuggers and Useful Crash Dumps

For pretty much any system you'd ever want to port to, there are readily available C debuggers and crash dump tools. These are invaluable to quickly finding the source of problems. And yes, there will be problems.

> *"Error, no keyboard — press F1 to continue."*

With any other language there might not be a usable debugger available and less likely a useful crash dump tool, and there is a really good chance for any heavy lifting you are interfacing with C code anyway. Now you have to debug the interface between the other language and the C code, and you often lose a ton of context, making it a cumbersome, error-prone process, and often completely useless in practice.

With pure C code, you can see call stacks, variables, arguments, thread locals, globals, basically everything in memory. This is ridiculously helpful especially when you have something that went wrong days into a long-running server process and isn't otherwise reproducible. If you lose this context in a higher level language, prepare for much pain.

Callable from Anywhere

C has a standardized application binary interface (ABI) that is supported by every OS, language, and platform in existence. And it requires no runtime or other inherent overhead. This means the code you write in C isn't just valuable to callers from C code, but to every conceivable library, language, and environment in existence.

> *"Portability is a result of few concepts and complete definition."*
> *— J. Palme*

You can use C code in standalone executables, scripting languages, kernel code, embedded code, as a DLL, even callable from SQL. It's the Lingua Franca of systems programming and pluggable libraries. If you want to write something once and have it usable from the most environments and use cases possible, C is the only sane choice.

Yes. It has Flaws

There are many "flaws" in C. It has no bounds checking, it's easy to corrupt anything in memory, there are dangling pointers and memory/resource leaks, bolted-on support for concurrency, no modules, no namespaces. Error handling can be painfully cumbersome and verbose. It's easy to make a whole class of errors where the call stack is smashed and hostile inputs take over your process. Closures? HA!

> *"When all else fails, read the instructions."*
> *— L. Lasellio*

Its flaws are very, very well known, and this is a virtue. All languages and implementations have gotchas and hang-ups. C is just far more upfront about it. And there are a ton of static and runtime tools to help you deal with the most common and dangerous mistakes. That some of the most heavily used and reliable software in the world is built on C is proof that the flaws are overblown, and easy to detect and fix.

At Couchbase we recently spent easily 2+ man/months dealing with a crash in the Erlang VM. We wasted a ton of time tracking down something that was in the core Erlang implementation, never sure what was happening or why, thinking perhaps the flaw was something in our own plug-in C code, hoping it was something we could find and fix. It wasn't, it was a race condition bug in core Erlang. We only found the problem via code inspection of Erlang. This is a fundamental problem in any language that abstracts away too much of the computer.

Initially for performance reasons, we started increasingly rewriting more of the Couchbase code in C, and choosing it as the first option for more new features. But amazingly it's proven much more predictable when we'll hit issues and how to debug and fix them. In the long run, it's more productive.

I always have it in the back of my head that I want to make a slightly better C. Just to clean up some of the rough edges and fix some of the more egregious problems. But getting everything to fit, top to bottom, syntax, semantics, tooling, etc., might not be possible or even worth the effort. As it stands today, C is unreasonably effective, and I don't see that changing any time soon. ■

Damien Katz is a recovering C++ fanatic, a founder and the former CTO and Chief Architect at Couchbase Inc, the creator of Apache CouchDB, a senior engineer on MySQL and Lotus Notes, and was once the "Erlanger of the Year". He is taking time off to spend with his 3 children, but is available for short term consulting and speaking engagements. [damienkatz.net]

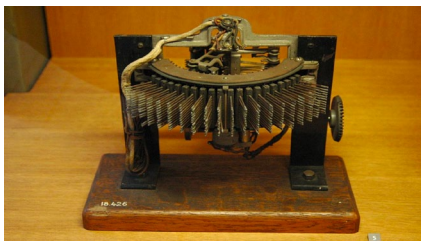
Reprinted with permission of the original author.
First appeared in *hn.my/c* (damienkatz.net)

Cache is the New RAM

By CARLOS BUENO

This is a talk given at Defrag 2014.

One of the (few) advantages of being in technology for a long time is that you get to see multiple tech cycles from beginning to end. You get to see how breakthroughs actually propagate. If all you have seen is a part of the curve, it's hard to extrapolate correctly. You either overshoot the short-term progress or undershoot the long. What's surprising is not how quickly the facts on the ground change, but how slowly engineering practice changes in response. This is a Strowger switch, an automated way to connect phone circuits. It was invented in 1891.



In 1951, right on the cusp of digital switching, the typical central switching office was basically a super-sized version of the Victorian technology. There was a strowger switch for every digit of every phone call in progress.

From the perspective of the time, this was the highest of high technology. Of course from our perspective, it was the world's largest Steampunk art installation.

It's probably a mistake to feel superior about that. It's been 65 years since the invention of the integrated circuit, but we still have billions of these guys around, whirring and clicking and breaking. It's only now that we are on the cusp of the switch to fully solid-state computing.

The most exciting kinds of technological shifts are when a new model finally becomes feasible, or when an old restriction falls away. Both kinds are happening right now in our industry.

Distributed computing is becoming the dominant programming model throughout the entire software stack. The so-called "Central Processing Unit" is no longer central, or even a unit. It's only one of many bugs crawling over a mountain of data. The database is the last holdout.

At the same time, the latency gap between RAM and hard drive storage is becoming irrelevant. For 30 years the central fact of database performance was the gigantic difference in the time it takes to access a random piece of data in RAM versus on a hard drive. It's now feasible to skip all that heartache by placing your data entirely in RAM. It's not as simple as that, of course. You can't just take a btree, mmap it, and call it a day. There are a lot of implications to a truly memory-native design that have yet to be unwound.

These two trends are producing an entirely new way to think about, design, and build applications. So let's talk about how we got here, how we're doing, and hints about where the future will take us.

Back in the day, every component in the architecture diagram had a definite article attached to it. Each thing was a separate function: "the" database and "the" web server, characters in a one-room drama. Incidentally, this is where the term "the cloud" came from. A fluffy cloud was the standard symbol for an external WAN whose details you didn't have to worry about.

Distributed computing hit the mainstream with the lowest-hanging fruit. Multiple identical application servers were hidden behind a "load balancer" which spread the work more or less evenly. Load-balancing only the stateless bits of the architecture

sidestepped a lot of philosophical problems. As the system scaled up, those components outflanked and eventually surrounded "the" database. We told ourselves that it was normal to spend more on special database hardware with fast disks and a faster CPU, and it was only one machine anyway. The hardware vendors were happy to take our money.

Eventually, database replication became reasonable and we salved our consciences by adding a hot spare database. We then told ourselves there were no longer any single points of failure. It was even true — for a few minutes.

That hot spare was too tempting to leave sitting idle, of course. Once the business analysts realized they could run gigantic queries on live production data without touching production, the so-called "hot spare" became nearly as busy and mission-critical as the production copy. We told ourselves it would be fine because if the spare is ever needed we can just take it from them for the duration of the emergency. But that's like saying you don't really need to carry a spare tire because you can always steal one from another car.

Then Brad Fitzpatrick released memcached, a daemon that caches data in memory. (Hence the name.) It was amazingly pragmatic software, a simplified version of the distributed hash tables then in vogue in academia. It had lots of features: a form of replication, sharding, load balancing, simple math operators. We told ourselves that most of our load was reads, so why make the database thrash the disk running the same queries over and over again? All you needed was a bunch of small-caliber servers with tons of RAM, and of course the hardware vendors were happy to take our money.

And…maybe you have to write some cache invalidation code. That doesn't sound too hard. Right?

To its credit, the memcached design took things a pretty long way. It replaced the random IO performance of a hard drive with the random IO performance of multiple banks of RAM. Even so, the database machine kept getting bigger and busier. We realized that caching cost at least as much RAM as the working set (otherwise it was ineffective), plus the nearly unbearable headache of cache consistency. But we told ourselves that was the cost of "web scale."

More worrisome was that applications were getting more sophisticated and chattier. Multiple database writes were being performed on almost every hit. Writes, not reads, became the bottleneck. This is when we finally got serious about sharding the database. Facebook initially sharded its user data by university and got away with concepts like "The Harvard Database" for a surprisingly long time. Flickr is another good example. They hand-built a sharding system in PHP that split the database up by a hash of the user ID, in much the same way that memcached shards on the key. In their tech talks there are jolly hints about having to denormalize their tables and double-write objects such as comments, messages, and favorites.

But that's a small price to pay for infinite scaling that solves everything ever. Right?

The problem with sharding a relational database by hand is that you no longer have a relational database. The API that orchestrates the sharding has in effect become your query language. Your operational headaches didn't get better either; the pain of altering schemas across the fleet was actually worse.

This was the point at which a lot of people took a deep breath, catalogued all the limitations and warts of their chosen implementation of SQL…and for some reason decided to blame SQL. A flood of hipster NoSQL and refugee XML databases appeared, all promising the moon. They offered automatic sharding, flexible schemas, some replication…and not much else at first. But it was less painful than writing it yourself.

You know things are really desperate when "less painful than writing it yourself" is the main selling point.

Moving to NoSQL wasn't worse than hand-sharding because we'd already given up any hope of using the usual client tools to manipulate and analyze our data. But it wasn't much better either. What used to be a SQL query written by the business folks turned into hand-written reporting code maintained by the developers.

Remember that "hot spare" database we used to use for backups and analytics? It came back with a vengeance in the form of Hadoop filestores and Hive querying on top. Now this worked, and largely got the business folks off our backs. The biggest problem is the operational complexity of these systems. Like the Space Shuttle, they were sold as reliable and nearly maintenance-free but turn out to need a ton of hands-on attention. The second biggest problem is getting the data in and out; a lag time of one day (!) was considered pretty good. The third problem is that it manages to

be I/O-bound on both network and disk at the same time. We told ourselves that was the price of graduating to BIG DATA.

Anyway, that's how Google does it. Right?

As various NoSQL databases matured, a curious thing happened to their APIs: they started looking more like SQL. This is because SQL is a pretty direct implementation of relational set theory, and math is hard to fool.

To paraphrase Paul Graham's unbearably smug comment about Lisp: once you add group by, filter, & join, you can no longer claim to have invented a new query language, only a new dialect of SQL. With worse syntax and no optimizer.

Because we had taken this strange detour away from SQL, crucial bits missing from most of the systems are a storage engine and query optimizer designed around relational set theory. Bolting that on later led to severe performance hits. For the ones that got it right (or papered it over by being resident in RAM) there were other bits missing like proper replication.

I know of one extremely successful web startup you've definitely heard of that uses four, count 'em, FOUR separate NoSQL systems to cover the gaps.

It's pretty clear that there's no going back to "the" database and 10-million-nanosecond random seek times. Underneath the endless hype cycles in search of the One True Thing To Solve Everything Ever is an interesting pattern: a pain point relieved by a clever approach that comes with a new pain point.

So what's the next complex gadget to add to this dog's breakfast? Maybe the real trick is to make things simpler.

For instance, RAM: You have lots of RAM in the "database" machines, for caching and calculation. You also have lots of RAM in the Memcached machines. The sum of RAM in those systems should be at least equal to the size of your working data set. If it isn't then you've under-bought. Also, I very much doubt that your caching layers are 100% efficient. I'll bet money you have plenty of data that are cached and never read again before eviction. I'll bet more money you don't even track that. That doesn't mean you're a bad person. It means that caching is often more trouble than it's worth.

> "Those who ignore computer history are condemned to GOTO 1"

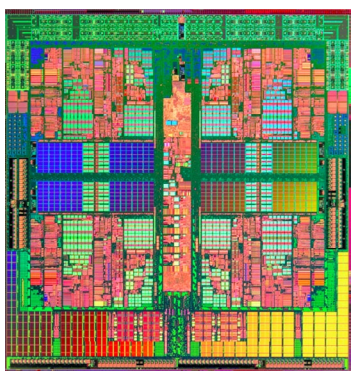
A lot of the features each of these components provides seem to be composable and complementary to one other. If only they could be arranged better.

Once you take it as axioms that the system will be distributed and the data will always be solid-state, a curious thing happens: it all gets much simpler. The "temporary" memory data structures you'd normally only use during query invocation becomes the only structure there is. Random access is no longer a cardinal sin; it's the normal course of business. You don't have to worry about splitting pages, or rebalancing, or data locality.

This is a nice, simple architecture. Just as load balancers abstract away the application servers, SQL "aggregators" abstract away the greasy details of orchestrating the reading and writing of data. This keeps the guts of the data placement strategies behind a stable API, which allows both sides to make changes with less disruption.

So it's all good now, right? We're finally arrived at the happy place at the end of history. Right?

It's a mistake to feel complacent about the state of the art of computing, no matter when you live. There's always another bottleneck.



This is the AMD "Barcelona" chip, a relatively modern design. It has four cores but the majority of the surface is taken by the cache and I/O areas surrounding cores, like a giant parking lot around a Walmart. In the Pentium era cache was only about 15% of the di. The third, quieter, revolution in computing is how much faster the CPU has gotten relative to memory. There's a reason all this expensive real estate is now reserved for cache.

The central fact of database performance used to be the latency gap between RAM and disk. At the moment we're kidding ourselves that the latency gap between CPU cache and RAM isn't exactly the same kind of problem. But it is.

And as much as we like to pretend that shared memory actually exists, it doesn't. With lots of cores and lots of RAM, inevitably some cores will be closer to some parts of RAM.

When you get right down to it, a computer really does only two things: read symbols and write symbols. Performance is a function of how much data the computer must move around, and where it goes. The happiest possible case is an endless sequential stream of data that's read once and dealt with quickly, never to be needed again. GPUs are a good example of this. But most interesting workloads aren't like that.

Every random pointer that's chased almost guarantees a cache miss. Every contention for the same area of memory (e.g. a write lock) causes huge coordination delay. Even if your CPU cache-hit rate was 99%, which it isn't, time spent waiting on RAM would still dominate.

Or put it this way: if disk is the new tape, and RAM is the new disk, then the CPU cache is the new RAM. Locality still matters.

So what will solve this problem? It seems that there's the same old fundamental conflicts: do we optimize for random or serial access? Do we take the performance penalty on writes or reads? Can we just sit tight and let the hardware catch up? Maybe memristors or other technology will make all of this irrelevant. Well, I want a pony, too.

The good news is that the gross physical architecture of distributed databases seems to be settling down. Data clients no longer need to deal with the guts and entrails of 4 or 5 separate subsystems. It's not perfect yet; it's not even mainstream yet. Breakthroughs take a while to propagate.

But if the next bottleneck really is memory locality, that means the rest of it has become mature. New innovations will tend to be in data structures and algorithms. There will be fewer sweeping architectural convulsions that promise to fix everything ever. If we're lucky, the next 15 years will be about SQL databases quietly getting faster and more efficient while exposing the same API.

But then again, our industry has never been quiet. ■

> "Throughput and latency *always* have the last laugh."

Carlos Bueno is an engineer at the database company MemSQL. Most recently he was a performance engineer at Facebook, where he helped save the company bags of cash through careful measurement and mature optimization.

Reprinted with permission of the original author. First appeared in *hn.my/cache* (memsql.com)

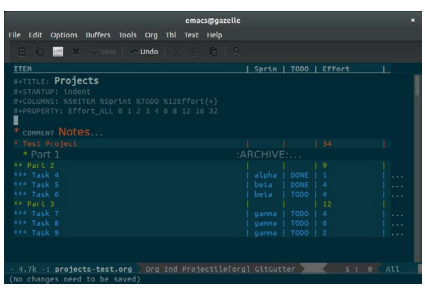
Getting Organized with Org Mode

By CORREL ROUSH

I've been using Emacs Org mode [orgmode.org] for nearly a year now. For a while I mostly just used it to take and organize notes, but over time I've discovered it's an incredibly useful tool for managing projects and tasks, writing and publishing documents, keeping track of time and to-do lists, and maintaining a journal.

Project Management

Most of what I've been using Org mode for has been breaking down large projects at work into tasks and subtasks. It's really easy to enter projects in as a hierarchy of tasks and task groupings. Using Column View, I was able to dive right into scoping them individually and reporting total estimates for each major segment of work.



Because Org Mode makes building and modifying an outline structure like this so quick and easy, I usually build and modify the project org document while planning it out with my team. Once done, I then manually load that information into our issue tracker and get underway. Occasionally I'll also update tags and progress status in the org document as well as the project progresses, so I can use the same document to plan subsequent development iterations.

Organizing Notes and Code Exercises

More recently, I've been looking into various ways to get more things organized with Org mode. I've been stepping through Structure and Interpretation of Computer Programs with some other folks from work, and discovered that Org mode was an ideal fit for keeping my notes and exercise work together. The latter is neatly managed by Babel, which let me embed and edit source examples and my exercise solutions right in the org document itself, and even export them to one or more scheme files to load into my interpreter.

Exporting and Publishing Documents

Publishing my notes with Org is also a breeze. I've published project plans and proposals to PDF to share with colleagues, and exported my SICP notes to html and dropped them into a site built with Jekyll. Embedding graphs and diagrams into exported documents using Graphviz, Mscgen, and PlantUML has also really helped with putting together some great project plans and documentation. A lot of great examples using those tools can be found here. [hn.my/orgex]

Emacs Configuration

While learning all the cool things I could do with Org mode and Babel, it was only natural I'd end up using it to reorganize my Emacs configuration. Up until that point, I'd been managing my configuration in a single init.el file, plus a directory full of mode or purpose-specific elisp files that I'd loop through and load. Inspired primarily by the blog post, "Making Emacs Work For Me", and later by others such as Sacha Chua's Emacs configuration, I got all my configs neatly organized into a single org file that gets loaded on startup. I've found it makes it far easier to keep track of what I've got configured, and gives me a reason to document and organize things neatly now that it's living a double life as a published document on GitHub. I've still got a directory lying around with autoloaded scripts, but now it's simply reserved for tinkering and sensitive configuration.

Tracking Habits

Another great feature of Org mode that I've been taking advantage of a lot more lately is the Agenda. By defining some org files as being agenda files, Org mode can examine these files for TODO entries, scheduled tasks, deadlines and more to build out useful agenda views to get a quick handle on what needs to be done and when. While at first I started by simply syncing down my google calendars as Org-files (using ical2org.awk), I've started managing TODO lists in a dedicated org file. By adding tasks to this file, scheduling them, and setting deadlines, I've been doing a much better job of keeping track of things I need to get done and (even more importantly) when I need to get them done.

This works not only for one-shot tasks, but also habits and other repetitive tasks. It's possible to schedule a task that should be done every day, every few days, or maybe every first Sunday of a month. For example, I've set up repeating tasks to write a blog post at least once a month, practice guitar every two to three days, and to do the dishes every one or two days. The agenda view can even show a small, colorized graph next to each repeating task that paints a picture of how well (or not!) I've been getting those tasks done on time.

Keeping a Journal and Tracking Work

The last thing I've been using (which I'm still getting a handle on) is using Capture to take and store notes, keep a journal, and even track time on tasks at work.

```
(setq org-capture-templates
   '(("j" "Journal Entry" plain
      (file+datetree "~/org/journal.org")
      "%U\n\n%?" :empty-lines-before 1)
     ("w" "Log Work Task" entry
      (file+datetree "~/org/worklog.org")
      "* TODO %^{Description}  %^g\n%?\n\
nAdded: %U"
      :clock-in t
      :clock-keep t)))

(global-set-key (kbd "C-c c") 'org-capture)

(setq org-clock-persist 'history)
(org-clock-persistence-insinuate)
```

For my journal, I've configured a capture template that I can use to write down a new entry that will be stored with a time stamp appended into its own Org file, organized under headlines by year, month, and date.

For work tasks, I have another capture template configured that will log and tag a task into another Org file, also organized by date, which will automatically start tracking time for that task. Once done, I can simply clock out and check the time I've spent, and can easily find it later to clock in again, add notes, or update its status. This helps me keep track of what I've gotten done during the day, keep notes on what I was doing at any point in time, and get a better idea of how long it takes me to do different types of tasks.

Conclusion

There's a lot that can be done with Org mode, and I've only just scratched the surface. The simple outline format provided by Org mode lends itself to doing all sorts of things, be it organizing notes, keeping a private or work journal, or writing a book or technical document. I've even written this blog post in Org mode! There's tons of functionality that can be built on top of it, yet the underlying format itself remains simple and easy to work with. I've never been great at keeping myself organized, but Org mode is such a delight to use that I can't help trying anyway. If it can work for me, maybe it can work for you, too!

There's tons of resources for finding new ways for using Org mode, and I'm still discovering cool things I can track and integrate with it. I definitely recommend reading through Sacha Chua's Blog [sachachua.com], as well as posts from John Wiegley [hn.my/wiegley]. I'm always looking for more stuff to try out. Feel free to drop me a line if you find or are using something you think is cool or useful!

Correl is a 32 year old software developer residing in the Philadelphia area with his wife, dog, and cat. Predominantly self-taught, he has been coding professionally for the past 8 year and is always finding new things to learn. His other interests include watching anime, playing video games, and learning the bass guitar.

Reprinted with permission of the original author.
First appeared in *hn.my/org* (phoenixinquis.net)

Node.js in Flames

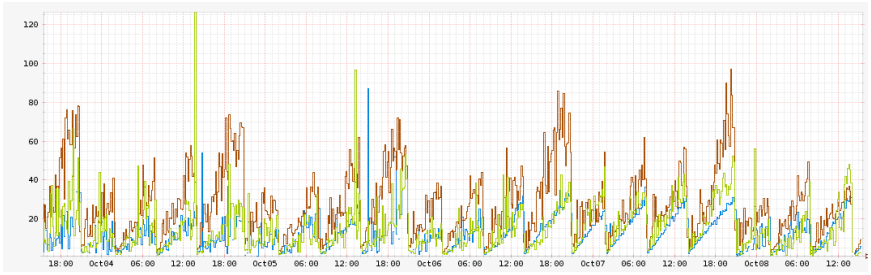
By YUNONG XIAO

We've been busy building our next-generation Netflix.com web application using Node.js. You can learn more about our approach from the presentation we delivered [hn.my/nodeflix] at NodeConf.eu a few months ago. Today, I want to share some recent learnings from performance tuning this new application stack.

We were first clued in to a possible issue when we noticed that request latencies to our Node.js application would increase progressively with time. The app was also burning CPU more than expected, and closely correlated to the higher latency. While using rolling reboots as a temporary workaround, we raced to find the root cause using new performance analysis tools and techniques in our Linux EC2 environment.

Flames Rising

We noticed that request latencies to our Node.js application would increase progressively with time. Specifically, some of our endpoints' latencies would start at 1ms and increase by 10ms every hour. We also saw a correlated increase in CPU usage.



This graph plots request latency in ms for each region against time. Each color corresponds to a different AWS AZ. You can see latencies steadily increase by 10 ms an hour and peak at around 60 ms before the instances are rebooted.

Dousing the Fire

Initially we hypothesized that there might be something faulty, such as a memory leak in our own request handlers that was causing the rising latencies. We tested this assertion by load-testing the app in isolation, adding metrics that measured both the latency of only our request handlers and the total latency of a request, as well as increasing the Node.js heap size to 32 GB.

We saw that our request handler's latencies stayed constant across the lifetime of the process at 1 ms. We also saw that the process's heap size stayed fairly constant at around 1.2 GB. However, overall request latencies and CPU usage continued to rise. This absolved our own handlers of blame, and pointed to problems deeper in the stack.

Something was taking an additional 60 ms to service the request. What we needed was a way to profile the application's CPU usage and visualize where we're spending most of our time on CPU. Enter CPU flame graphs and Linux Perf Events to the rescue.

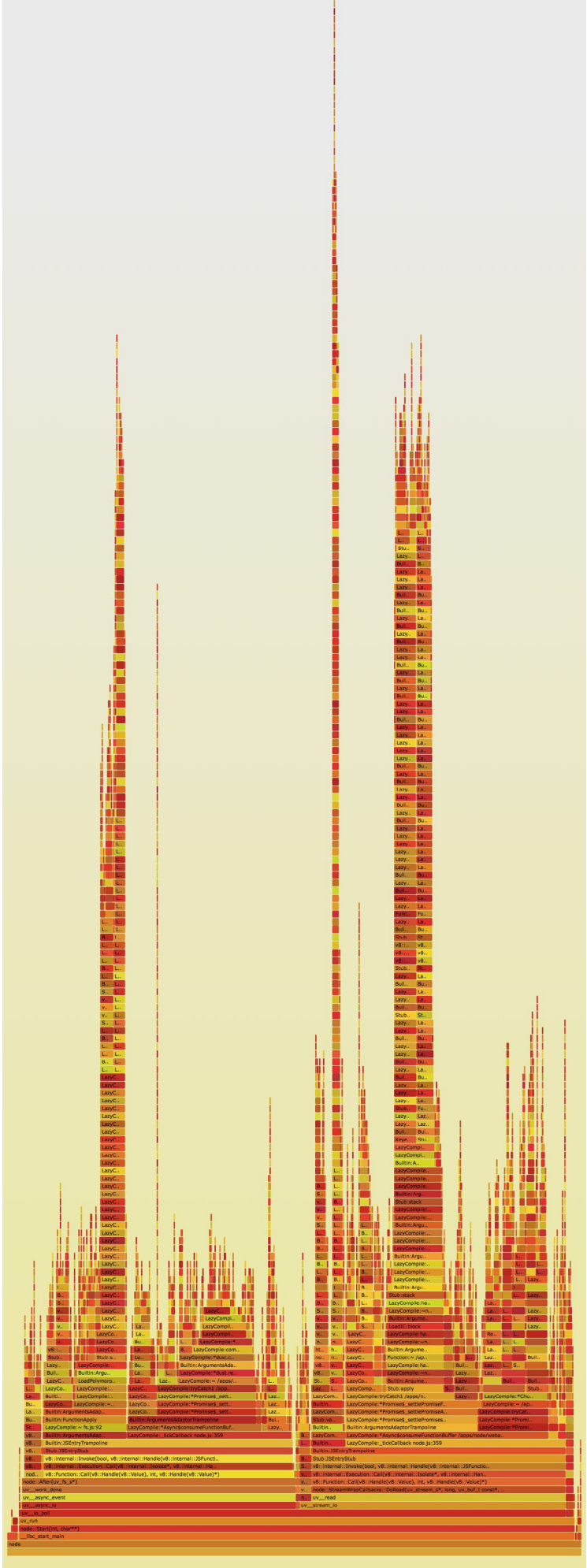
For those unfamiliar with flame graphs, it's best to read Brendan Gregg's excellent article explaining what they are [hn.my/flamegraph] — but here's a quick summary (straight from the article).

- Each box represents a function in the stack (a "stack frame").
- The y-axis shows stack depth (number of frames on the stack). The top box shows the function

that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent, just like the stack traces shown earlier.

- The x-axis spans the sample population. It does not show the passing of time from left to right, as most graphs do. The left to right ordering has no meaning (it's sorted alphabetically).
- The width of the box shows the total time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Wider box functions may be slower than narrow box functions, or, they may simply be called more often. The call count is not shown (or known via sampling).
- The sample count can exceed elapsed time if multiple threads were running and sampled concurrently.
- The colors aren't significant, and are picked at random to be warm colors. It's called "flame graph" as it's showing what is hot on-CPU. And, it's interactive: mouse over the SVGs to reveal details.

Previously Node.js flame graphs had only been used on systems with DTrace, using Dave Pacheco's Node.js jstack() support. However, the Google v8 team has more recently added perf_events support to v8, which allows similar stack profiling of JavaScript symbols on Linux. Brendan has written instructions for how to use this new support, which arrived in Node.js version 0.11.13, to create Node.js flame graphs on Linux.



Here's the original SVG [hn.my/200mins] of the flame graph. Immediately, we see incredibly high stacks in the application (y-axis). We also see we're spending quite a lot of time in those stacks (x-axis). On closer inspection, it seems the stack frames are full of references to Express.js's `router.handle` and `router.handle.next` functions. The Express.js source code reveals a couple of interesting tidbits.

- Route handlers for all endpoints are stored in one global array.
- Express.js recursively iterates through and invokes all handlers until it finds the right route handler.

A global array is not the ideal data structure for this use case. It's unclear why Express.js chose not to use a constant time data structure like a map to store its handlers. Each request requires an expensive O(n) look up in the route array in order to find its route handler. Compounding matters, the array is traversed recursively. This explains why we saw such tall stacks in the flame graphs. Interestingly, Express.js even allows you to set many identical route handlers for a route. You can unwittingly set a request chain like so.

```
[a, b, c, c, c, c, d, e, f, g, h]
```

Requests for route c would terminate at the first occurrence of the c handler (position 2 in the array). However, requests for d would only terminate at position 6 in the array, having needless spent time spinning through a, b and multiple instances of c. We verified this by running the following vanilla express app.

```
var express = require('express');
var app = express();
app.get('/foo', function (req, res) {
   res.send('hi');
});
// add a second foo route handler
app.get('/foo', function (req, res) {
   res.send('hi2');
});
console.log('stack', app._router.stack);
app.listen(3000);
```

Running this Express.js app returns these route handlers.

```
stack [ { keys: [], regexp: /^\/?(?=/|$)/i,
handle: [Function: query] },
 { keys: [],
   regexp: /^\/?(?=/|$)/i,
   handle: [Function: expressInit] },
 { keys: [],
   regexp: /^\/foo\/?$/i,
   handle: [Function],
   route: { path: '/foo', stack: [Object], meth-
ods: [Object] } },
 { keys: [],
   regexp: /^\/foo\/?$/i,
   handle: [Function],
   route: { path: '/foo', stack: [Object], meth-
ods: [Object] } } ]
```

Notice there are two identical route handlers for `/foo`. It would have been nice for Express.js to throw an error whenever there's more than one route handler chain for a route.

At this point the leading hypothesis was that the handler array was increasing in size with time, thus leading to the increase of latencies as each handler is invoked. Most likely we were leaking handlers somewhere in our code, possibly due to the duplicate handler issue. We added additional logging which periodically dumps out the route handler array, and noticed the array was growing by 10 elements every hour. These handlers happened to be identical to each other, mirroring the example from above.

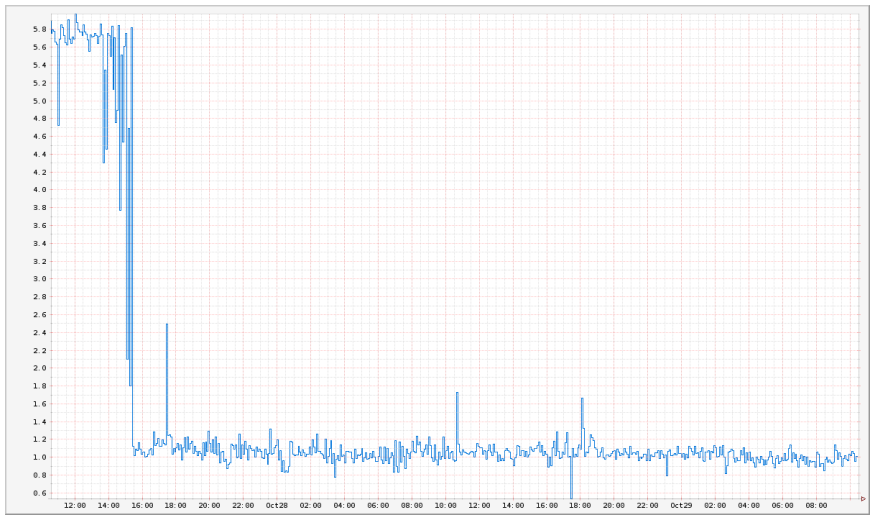
```
[...
{ handle: [Function: serveStatic],
   name: 'serveStatic',
   params: undefined,
   path: undefined,
   keys: [],
   regexp: { /^\/?(?=\/|$)/i fast_slash: true },
   route: undefined },
 { handle: [Function: serveStatic],
   name: 'serveStatic',
   params: undefined,
   path: undefined,
   keys: [],
   regexp: { /^\/?(?=\/|$)/i fast_slash: true },
   route: undefined },
 { handle: [Function: serveStatic],
```

```
    name: 'serveStatic',
    params: undefined,
    path: undefined,
    keys: [],
    regexp: { /^\/?(?=\/|$)/i
fast_slash: true },
    route: undefined },
...
]
```

Something was adding the same Express.js provided static route handler 10 times an hour. Further benchmarking revealed merely iterating through each of these handler instances cost about 1 ms of CPU time. This correlates to the latency problems we've seen, where our response latencies increase by 10 ms every hour.

This turned out be caused by a periodic (10/hour) function in our code. The main purpose of this was to refresh our route handlers from an external source. This was implemented by deleting old handlers and adding new ones to the array. Unfortunately, it was also inadvertently adding a static route handler with the same path each time it ran. Since Express.js allows for multiple route handlers given identical paths, these duplicate handlers were all added to the array. Making matter worse, they were added before the rest of the API handlers, which meant they all had to be invoked before we can service any requests to our service.

This fully explains why our request latencies were increasing by 10ms every hour. Indeed, when we fixed our code so that it stopped adding duplicate route handlers, our latency and CPU usage increases went away.



Here we see our latencies drop down to 1 ms and remain there after we deployed our fix.

When the Smoke Cleared

What did we learn from this harrowing experience? First, we need to fully understand our dependencies before putting them into production. We made incorrect assumptions about the Express.js API without digging further into its code base. As a result, our misuse of the Express.js API was the ultimate root cause of our performance issue.

Second, given a performance problem, observability is of the utmost importance. Flame graphs gave us tremendous insight into where our app was spending most of its time on CPU. I can't imagine how we would have solved this problem without being able to sample Node.js stacks and visualize them with flame graphs. ■

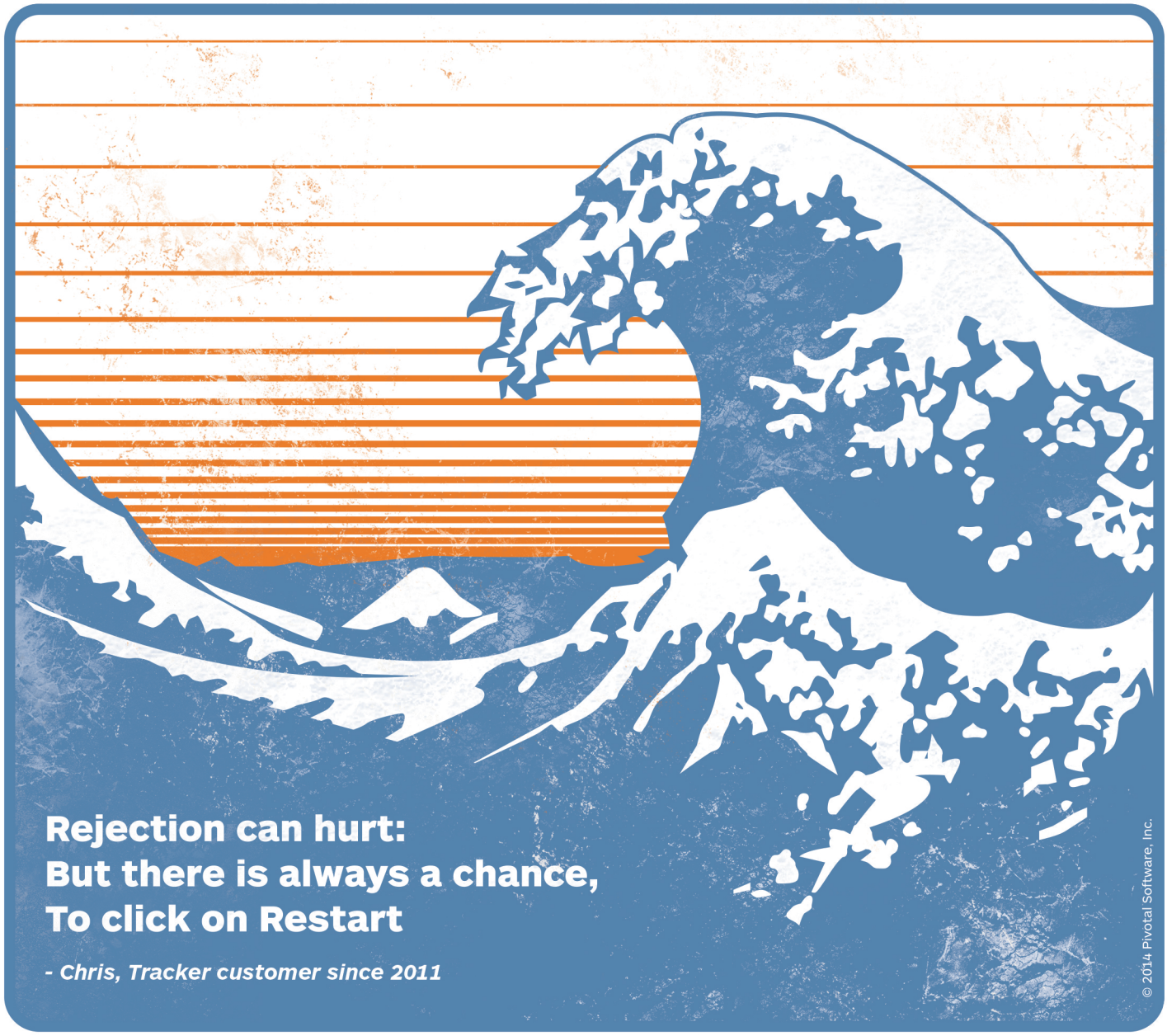
Yunong is currently a Senior Node.JS Software Engineer at Netflix — where he's leading the transition to Node.js there. He has spent his career scaling distributed systems; first at AWS, and more recently at Joyent — where he launched the Manta object store and compute service.

Reprinted with permission of the original author.
First appeared in *hn.my/flamenode* (netflix.com)

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com



Discover the newly redesigned PivotalTracker

As our customers know too well, building software is challenging. That's why we created PivotalTracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused and reflect the true status of all your software projects.

With a new UI, cross-project functionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com

PivotalTracker

Build better software, faster.