Founder/CEO Pulumi • Cloud, languages, and developer tools guy • Eat, sleep, code, repeat

April 10, 2016

Performance Culture

In this essay, I'll talk about "performance culture." Performance is one of the key pillars of software engineering, and is something that's hard to do right, and sometimes even difficult to recognize. As a famous judge once said, "I know it when I see it." I've spoken at length about performance and culture independently before, however the intersection of the two is where things get interesting. Teams who do this well have performance ingrained into nearly all aspects of how the team operates from the start, and are able to proactively deliver loveable customer experiences that crush the competition. There's no easy cookie-cutter recipe for achieving a good performance culture, however there are certainly some best practices you can follow to plant the requisite seeds into your team. So, let's go!

Introduction

Why the big focus on performance, anyway?

Partly it's my background. I've worked on systems, runtimes, compilers, … things that customers expect to be fast. It's always much easier to incorporate goals, metrics, and team processes at the outset of such a project, compared to attempting to recover it later on. I've also worked on many teams, some that have done amazing at this, some that have done terribly, and many in between. The one universal truth is that the differentiating factor is always culture.

Partly it's because, no matter the kind of software, performance is almost always worse than our customers would like it to be. This is a simple matter of physics: it's impossible to speed up all aspects of a program, given finite time, and the tradeoffs involved between size, speed, and functionality. But I firmly believe that on the average teams spend way less attention to developing a rigorous performance culture. I've heard the "performance isn't a top priority for us" statement many times only to later be canceled out by a painful realization that without it the product won't succeed.

And partly it's just been top of mind for all of us in DevDiv, as we focus on .NET core performance, ASP.NET scalability, integrating performance-motivated features into C# and the libraries, making Visual Studio faster, and more. It's particularly top of mind for me, as I've been comparing our experiences to my own in Midori (which heavily inspired this blog post).

Diagnosis and The Cure

How can you tell whether your performance culture is on track? Well, here are some signs that it's not:

- Answering the question, "how is the product doing on my key performance metrics," is difficult.
- Performance often regresses and team members either don't know, don't care, or find out too late to act.
- Blame is one of the most common responses to performance problems (either people, infrastructure, or both).
- Performance tests swing wildly, cannot be trusted, and are generally ignored by most of the team.
- Performance is something one, or a few, individuals are meant to keep an eye on, instead of the whole team.
- Performance issues in production are common, and require ugly scrambles to address (and/or cannot be reproduced).

These may sound like technical problems. It may come as a surprise, however, that they are primarily human problems.

The solution isn't easy, especially once your culture is in the weeds. It's always easier to not dig a hole in the first place than it is to climb out of one later on. But the first rule when you're in a hole is to stop digging! The cultural transformation must start from the top – management taking an active role in performance, asking questions, seeking insights, demanding rigor – while it simultaneously comes from the bottom – engineers actively seeking to understand performance of the code they are writing, ruthlessly taking a zero-tolerance stance on regressions, and being ever-self-critical and on the lookout for proactive improvements.

This essay will describe some ways to ensure this sort of a culture, in addition to some best practices I've found help to increase its effectiveness once you have one in place. A lot of it may seem obvious, but believe me, it's pretty rare to see everything in here working in harmony in practice. But when it is, wow, what a difference it can make.

A quick note on OSS software. I wrote this essay from the perspective of commercial software development. As such, you'll see the word "management" a lot. Many of the same principles work in OSS too. So, if you like, anytime you see "management," mentally transform it into "management or the project's committers."

It Starts, and Ends, with Culture

The key components of a healthy performance culture are:

1. Performance is part of the daily dialogue and "buzz" within the team. Everybody plays a role.
2. Management must care – truly, not superficially – about good performance, and understand what it takes.
3. Engineers take a scientific, data-driven, inquisitive approach to performance. (Measure, measure, measure!)
4. Robust engineering systems are in place to track goals, past and present performance, and to block regressions.

I'll spend a bit of time talking about each of these roughly in turn.

Dialogue, Buzz, and Communication

The entire team needs to be on the hook for performance.

In many teams where I've seen this go wrong, a single person is anointed the go-to performance guy or gal. Now, that's fine and can help the team scale, can be useful when someone needs to spearhead an investigation, and having a vocal advocate of performance is great, but it *must not* come at the expense of the rest of the team's involvement.

This can lead to problems similar to those Microsoft use to have with the "test" discipline; engineers learned bad habits by outsourcing the basic quality of their code, assuming that someone else would catch any problems that arise. The same risks are present when there's a central performance czar: engineers on the team won't write performance tests, won't proactively benchmark, won't profile, won't ask questions about the competitive standing of the product, and generally won't do all the things you need all of the engineers doing to build a healthy performance culture.

Magical things happen when the whole team is obsessed about performance. The hallways are abuzz with excitement, as news of challenges and improvements spreads organically. "Did you see Martin's hashtable rebalancing change that reduced process footprint by 30%?" "Jared just checked in a feature that lets you stack allocate arrays. I was thinking of hacking the networking stack this weekend to use it – care to join in?" Impromptu whiteboard sessions, off-the-cuff ideation, group learning. It's really awesome to see. The excitement and desire to win propels the team forward, naturally, and without relying on some heavyweight management "stick."

I hate blame and I hate defensiveness. My number one rule is "no jerks," so naturally all critiques must be delivered in the most constructive and respectful way. I've found a high occurrence of blame, defensiveness, and intellectual dishonesty in teams that do poorly on performance, however. Like jerks, these are toxic to team culture and must be weeded out aggressively. It can easily make or break your ability to develop the right performance culture. There's nothing wrong with saying we need to do better on some key metric, especially if you have some good ideas on how to do so!

In addition to the ad-hoc communication, there of course needs to be structured communication also. I'll describe some techniques later on. But getting a core group of people in a room regularly to discuss the past, present, and future of performance for a particular area of the product is essential. Although the organic conversations are powerful, everyone gets busy, and it's important to schedule time as a reminder to keep pushing ahead.

Management: More Carrots, Fewer Sticks

In every team with a poor performance culture, it's management's fault. Period. End of conversation.

Engineers can and must make a difference, of course, but if the person at the top and everybody in between aren't deeply involved, budgeting for the necessary time, and rewarding the work and the superstars, the right culture won't take hold. A single engineer alone can't possibly infuse enough of this culture into an entire team, and most certainly not if the entire effort is running upstream against the management team.

It's painful to watch managers who don't appreciate performance culture. They'll often get caught by surprise and won't realize why – or worse, think that this is just how engineering works. ("We can't predict where performance will matter in advance!") Customers will complain that the product doesn't perform as expected in key areas and, realizing it's too late for preventative measures, a manager whose team has a poor performance culture will start blaming things. Guess what? The blame game culture spreads like wildfire, the engineers start doing it too, and accountability goes out the window. Blame doesn't solve anything. Blaming is what jerks do.

Notice I said management must be "*deeply* involved": this isn't some superficial level of involvement. Sure, charts with greens, reds, and trendlines probably need to be floating around, and regular reviews are important. I suppose you could say that these are pointy-haired manager things. (Believe me, however they do help.) A manager must go deeper than this, however, proactively and regularly reviewing the state of performance across the product, alongside the other basic quality metrics and progress on features. It's a core tenet of the way the team does its work. It must be treated as such. A manager must wonder about the competitive landscape and ask the team good, insightful questions that get them thinking.

Performance doesn't come for free. It costs the team by forcing them to slow down at times, to spend energy on things other than cranking out features, and hence requires some amount of intelligent tradeoff. How much really depends on the scenario. Managers need to coach the team to spend the right ratio of time. Those who assume it will come for free usually end up spending 2-5x the amount it would have taken, just at an inopportune time later on (e.g., during the endgame of shipping the product, in production when trying to scale up from 1,000 customers to 100,000, etc).

A mentor of mine used to say "You get from your teams what you reward." It's especially true with performance and the engineering systems surrounding them. Consider two managers:

- *Manager A* gives lip service to performance culture. She, however, packs every sprint schedule with a steady stream of features – "we've got to crush competitor Z and *must* reach feature parity!" – with no time for breaks in-between. She spends all-hands team meetings praising new features, demos aplenty, and even gives out a reward to an engineer at each one for "the most groundbreaking feature." As a result, her team cranks out features at an impressive clip, delivers fresh demos to the board every single time, and gives the sales team plenty of ammo to pursue new leads. There aren't performance gates and engineers generally don't bother to think much about it.

- *Manager B* takes a more balanced approach. She believes that given the competitive landscape, and the need to impress customers and board members with whizbang demos, new features need to keep coming. But she is also wary of building up too much debt in areas like performance, reliability, and quality for areas she expects to stick. So she intentionally puts her foot on the brake and pushes the team just as hard on these areas as she does features. She demands good engineering systems and live flighting of new features with performance telemetry built-in, for example. This requires that she hold board members and product managers at bay, which is definitely unpopular and difficult. In addition to a reward for "the most groundbreaking feature" award at each all-hands, she shows charts of performance progress and delivers a "performance ninja" award too, to the engineer who delivered the most impactful performance improvement. Note that engineering systems improvements also qualify!

Which manager do you think is going to ship a quality product, on time, that customers are in love with? My money is on Manager B. Sometimes you've got to slow down to speed up.

Microsoft is undergoing two interesting transitions recently that are related to this point: on one hand, the elimination of "test" as a discipline mentioned earlier; and, on the other hand, a renewed focus on engineering systems. It's been a bumpy ride. Surprisingly, one of the biggest hurdles to get over wasn't with the individual engineers at all – it was the managers! "Development managers" in the old model got used to focusing on features, features, features, and left most of the engineering systems work to contractors, and most of the quality work to testers. As a result, they were ill-prepared to recognize and reward the kind of work that is essential to building a great performance culture. The result? You guessed it: a total lack of performance culture. But, more subtly, you also ended up with "leadership holes"; until recently, there were virtually no high-ranking engineers working on the mission-critical engineering systems that make the entire team more productive and capable. Who wants to make a career out of the mere grunt work assigned to contractors and underappreciated by management? Again, you get what you reward.

There's a catch-22 with early prototyping where you don't know if the code is going to survive at all, and so the temptation is to spend zero time on performance. If you're hacking away towards a minimum viable product (MVP), and you're a startup burning cash, it's understandable. But I strongly advise against this. Architecture matters, and some poorly made architectural decisions made at the outset can lay the foundation for an entire skyscraper of ill-performing code atop. It's better to make performance part of the feasibility study and early exploration.

Finally, to tie up everything above, as a manager of large teams, I think it's important to get together regularly – every other sprint or two – to review performance progress with the management team. This is in addition to the more fine-grained engineer, lead, and architect level pow-wows that happen continuously. There's a bit of a "stick" aspect of such a review, but it's more about celebrating the team's self-driven accomplishments, and keeping it on management's radar. Such reviews should be driven from the lab and manually generated numbers should be outlawed.

Which brings me to …

Process and Infrastructure

"Process and infrastructure" – how boring!

Good infrastructure is a must. A team lacking the above cultural traits won't even stop to invest in infrastructure; they will simply live with what should be an infuriating lack of rigor. And good process must ensure effective use of this infrastructure. Here is the bare minimum in my book:

- All commits must pass a set of gated performance tests beforehand.
- Any commits that slip past this and regress performance are reverted without question. I call this the zero tolerance rule.
- Continuous performance telemetry is reported from the lab, flighting, and live environments.
- This implies that performance tests and infrastructure have a few important characteristics:
  - They aren't noisy.
  - They measure the "right" thing.
  - They can be run in a "reasonable" amount of time.

I have this saying: "If it's not automated, it's dead to me."

This highlights the importance of good infrastructure and avoids the dreaded "it worked fine on my computer" that everybody, I'm sure, has encountered: a test run on some random machine – under who knows what circumstances – is quoted to declare success on a benchmark… only to find out some time later that the results didn't hold. Why is this?

There are countless possibilities. Perhaps a noisy process interfered, like AntiVirus, search indexing, or the application of operating system updates. Maybe the developer accidentally left music playing in the background on their multimedia player. Maybe the BIOS wasn't properly adjusted to disable dynamic clock scaling. Perhaps it was due to an innocent data-entry error when copy-and-pasting the numbers into a spreadhseet. Or maybe the numbers for two comparison benchmarks came from two, incomparable machine configurations. I've seen all of these happen in practice.

In any manual human activity, mistakes happen. These days, I literally refuse to look at or trust any number that didn't come from the lab. The solution is to automate everything and focus your energy on making the automation infrastructure as damned good as possible. Pay some of your best people to make this rock solid for the rest of the team. Encourage everybody on the team to fix broken windows, and take a proactive approach to improving the infrastructure. And reward it heartily. You might have to go a little slower, but it'll be worth it, trust me.

Test Rings

I glossed over a fair bit above when I said "all commits must pass a set of performance tests," and then went on to talk about how a checkin might "slip past" said tests. How is this possible?

The reality is that it's usually not possible to run all tests and find all problems before a commit goes in, at least not within a reasonable amount of time. A good performance engineering system should balance the productivity of speedy codeflow with the productivity and assurance of regression prevention.

A decent approach for this is to organize tests into so-called "rings":

- An inner ring containing tests that all developers on the team measure before each commit.
- An inner ring containing tests that developers on your particular sub-team measure before each commit.
- An inner ring containing tests that developers run at their discretion before each commit.
- Any number of successive rings outside of this:
  - Gates for each code-flow point between branches.
  - Post-commit testing – nightly, weekly, etc. – based on time/resource constraints.
  - Pre-release verification.
  - Post-release telemetry and monitoring.

As you can see, there's a bit of flexibility in how this gets structured in practice. I wish I could lie and say that it's a science, however it's an art that requires intelligently trading off many factors. This is a constant source of debate and one that the management team should be actively involved in.

A small team might settle on one standard set of benchmarks across the whole team. A larger team might need to split inner ring tests along branching lines. And no matter the size, we would expect the master/main branch to enforce the most important performance metrics for the whole team, ensuring no code ever flows in that damages a core scenario.

In some cases, we might leave running certain pre-commit tests to the developer's discretion. (Note, this does not mean running pre-commit tests altogether is optional – only a particular set of them!) This might be the case if, for example, the test covered a lesser-used component and we know the nightly tests would catch any post-commit regression. In general, when you have a strong performance culture, it's okay to trust judgement calls sometimes. Trust but verify.

Let's take a few concrete examples. Performance tests often range from micro to macro in size. These typically range from easier to harder to pinpoint the source of a regression, respectively. (Micro measures just one thing, and so fluctuations tend to be easier to understand, whereas macro measures an entire system, where fluctuations tend to take a bit of elbow grease to track down.) A web server team might include a range of micro and macro tests in the innermost pre-commit suite of tests: number of bytes allocated per requests (micro), request response time (micro), … perhaps a half dozen other micro-to-midpoint benchmarks …, and TechEmpower (macro), let's say. Thanks to lab resources, test parallelism, and the awesomeness of GitHub webhooks, let's say these all complete in 15 minutes, nicely integrated into your pull request and code review processes. Not too bad. But this clearly isn't perfect coverage. Maybe every night, TechEmpower is run for 4 hours, to measure performance over a longer period of time to identify leaks. It's possible a developer could pass the pre-commit tests, and then fail this longer test, of course. Hence, the team lets developers run this test on-demand, so a good doobie can avoid getting egg on his or her face. But alas, mistakes happen, and again there isn't a culture of blame or witchhunting; it is what it is.

This leads me to back to the zero tolerance rule.

Barring exceptional circumstances, regressions should be backed out immediately. In teams where I've seen this succeed, there were no questions asked, and no IOUs. As soon as you relax this stance, the culture begins to break down. Layers of regressions pile on top of one another and you end up ceding ground permanently, no matter the best intentions of the team. The commit should be undone, the developer should identify the root cause, remedy it, ideally write a new test if appropriate, and then go through all the hoops again to submit the checkin, this time ensuring good performance.

Measurement, Metrics, and Statistics

Decent engineers intuit. Good engineers measure. Great engineers do both.

Measure what, though?

I put metrics into two distinct categories:

- *Consumption metrics.* These directly measure the resources consumed by running a test.
- *Observational metrics.* These measure the outcome of running a test, *observationally*, using metrics "outside" of the system.

Examples of consumption metrics are hardware performance counters, such as instructions retired, data cache misses, instruction cache misses, TLB misses, and/or context switches.

Software performance counters are also good candidates, like number of I/Os, memory allocated (and collected), interrupts, and/or number of syscalls. Examples of observational metrics include elapsed time and cost of running the test as billed by your cloud provider. Both are clearly important for different reasons.

Seeing a team measure time and time alone literally brings me to tears. It's a good measure of what an end-user will see – and therefore makes a good high-level test – however it is seriously lacking in the insights it can give you. And if there's no visibility into variance, it can be borderline useless.

Consumption metrics are obviously much more helpful to an engineer trying to understand why something changed. In our above web server team example, imagine request response time regressed by 30%. All the test report tells us is the time. It's true, a developer can then try to reproduce the scenario locally, and manually narrow down the cause, however can be tedious, takes time, and is likely imperfect due to differences in lab versus local hardware. What if, instead, both instructions retired and memory allocated were reported alongside the regression in time? From this, it could be easy to see that suddenly 256KB of memory was being allocated per request that wasn't there before. Being aware of recent commits, this could make it easy for an engineer to quickly pinpoint and back out the culprit in a timely manner before additional commits pile on top, further obscuring the problem. It's like printf debugging.

Speaking of printf debugging, telemetry is essential for long-running tests. Even low-tech approaches like printfing the current set of metrics every so often (e.g., every 15 seconds), can help track down where something went into the weeds simply by inspecting a database or logfile. Imagine trying to figure out where the 4-hour web server test went off the rails at around the 3 1/2 hour mark. This can can be utterly maddening without continuous telemetry! Of course, it's also a good idea to go beyond this. The product should have a built-in way to collect this telemtry out in the wild, and correlate it back to key metrics. StatsD is a fantastic option.

Finally, it's important to measure these metrics as scientifically as possible. That includes tracking standard deviation, coefficient of variation (CV), and geomean, and using these to ensure tests don't vary wildly from one run to the next. (Hint: commits that tank CV should be blocked, just as those that tank the core metric itself.) Having a statistics wonk on your team is also a good idea!

Goals and Baselines

Little of the above matters if you lack goals and baselines. For each benchmark/metric pair, I recommend recognizing four distinct concepts in your infrastructure and processes:

- *Current*: the current performance (which can span multiple metrics).
- *Baseline*: the threshold the product must stay above/below, otherwise tests fail.
- *Sprint Goal*: where the team must get to before the current sprint ends.
- *Ship Goal*: where the team must get to in order to ship a competitive feature/scenario.

Assume a metric where higher is better (like throughput); then it's usually the case that Ship Goal >= Sprint Goal >= Current >= Baseline. As wins and losses happen, continual adjustments should be made.

For example, a "baseline ratcheting" process is necessary to lock in improvements. A reasonable approach is to ratchet the baseline automatically to within some percentage of the current performance, ideally based on standard deviation and/or CV. Another approach is to require that developers do it manually, so that all ratcheting is intentional and accounted for. And interestingly, you may find it helpful to ratchet in the other direction too. That is, block commits that *improve* performance dramatically and yet do not ratchet the baseline. This forces engineers to stop and think about whether performance changes were intentional – even the good ones! A.k.a., "confirm your kill."

It's of course common that sprint goals remain stable from one sprint to the next. All numbers can't be improving all the time. But this system also helps to ensure that the team doesn't backslide on prior achievements.

I've found it useful to organize sprint goals behind themes. Make this sprint about "server performance." Or "shake out excessive allocations." Or something else that gives the team a sense of cohesion, shared purpose, and adds a little fun into the mix. As managers, we often forget how important fun is. It turns out performance can be the greatest fun of all; it's hugely measurable – which engineers love – and, speaking for myself, it's a hell of a time to pull out the scythe and start whacking away! It can even be a time to learn as a team, and to even try out some fun, new algorithmic techniques, like bloom filters.

Not every performance test needs this level of rigor. Any that are important enough to automatically run pre-commit most certainly demand it. And probably those that are run daily or monthly. But managing all these goals and baselines and whatnot can get really cumbersome when there are too many of them. This is a real risk especially if you're tracking multiple metrics for each of your benchmarks.

This is where the idea of "key performance indicators" (KPIs) becomes very important. These are the performance metrics important enough to track at a management level, to the whole team how healthy the overall product is at any given time. In my past team who built an operating system and its components, this included things like process startup time, web server throughput, browser performance on standard industry benchmarks, and number of frames dropped in our realtime audio/video client, including multiple metrics apiece plus the abovementioned statistics metrics. These were of course in the regularly running pre- and post-commit test suites, but rolling them up in one place, and tracking against the goals, was a hugely focusing exercise.

In Summary

This post just scratches the surface of how to do good performance engineering, but I hope you walk away with at least one thing: doing performance well is all about having a great performance culture.

This culture needs to be infused throughout the entire organization, from management down to engineers, and everybody in between. It needs to be transparent, respectful, aggressive, data-driven, self-critical, and relentlessly ambitious. Great infrastructure and supporting processes are a must, and management needs to appreciate and reward these, just as they would feature work (and frequently even more). Only then will the self-reinforcing flywheel get going.

Setting goals, communicating regularly, and obsessively tracking goals and customer-facing metrics is paramount.

It's not easy to do everything I've written in this article. It's honestly very difficult to remember to slow down and be disciplined in these areas, and it's easy to trick yourself into thinking running as fast as possible and worrying about performance later is the right call. Well, I'm sorry to tell you, sometimes it is. You've got to use your intuition and your gut, however, in my experience, we tend to undervalue performance considerably compared to features.

If you're a manager, your team will thank you for instilling a culture like this, and you'll be rewarded by shipping better performing software on schedule. If you're an engineer, I guarantee you'll spend far less time scrambling, more time being proactive, and more time having fun, in a team obsessed over customer performance. I'd love to hear what you think and your own experiences establishing a performance culture.

Post