ECE459: Programming for Performance Winter 2024

Lecture 35 — DevOps: Operations

Jeff Zarnett *2024-07-14*

DevOps, but Operations

Let's imagine that we've got our services up and running and doing The Thing that they are supposed to do. A good start, but we need to keep it running. How do we know if there's a problem? Monitoring, of course. And if we see a problem? Alerting.

Monitoring and Alerting

Monitoring is surprisingly difficult. There are a lot of recommendations about what to monitor and what to do about it. Applications should have health checks, though these are often basic: is the app running and able to respond to incoming requests? A more realistic health check should probably see that the system can actually do useful work, such as reach its database. You can even expand on this by automated testing of basic workflows, like users can log in and see their data.

A simple health check that verifies that the app can respond or reach its database won't show us performance problems. Yes, the app might be able to reach the database and the automated test user may be able to log in, but what if these are ten times slower than usual? Here are some things we could look at:

- CPU Load
- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Queue lengths
- Application Response Times

We've already covered a number of ideas about profiling (and queueing theory!) that help us understand what and how to measure in many of these areas, so we won't repeat it here.

With multiple systems, you will want some sort of dashboard that gives an overview of all the systems in a summary. The summary needs to be sufficiently detailed that you can detect if anything is wrong, but not an overwhelming wall of data. Information dashboard design is a complicated topic and the subject of some systems design engineering courses. It's a little bit late for me to recommend you take one of those as we're at the end of the course, but maybe for graduate studies or continuing education?

Realistically, you do not want to pay someone to stare at the dashboard and press the "Red Alert!" button if anything goes out of some preset range of what is okay. No, for that we need some automatic monitoring. Monitoring tools allow us to set up alerting based on certain conditions or thresholds. Some immediate examples:

- CPU usage exceeding threshold for a certain period of time
- Increased rate of error logs over a period of time
- A service has restarted many times recently
- Queue length very long
- Taking too long to complete a workflow

There is a spectrum here of what these mean... CPU usage being high may not be a problem on its own. If the system is at 80% CPU usage consistently, it may be keeping up with the workload and it's just an indication that we're close to the maximum here. If it's at 100% then we should be worried that our performance is limited by CPU availability. If queue lengths are long, it might only be a problem if a certain response time is expected – merely having a big backlog of work to do isn't a problem unless it means missing deadlines or being unable to do other things. Other measures are more outcomes-focused: if logging in takes too long, users are going to struggle or be frustrated, so we should consider it a problem.

Remember also the lesson from Sherlock Holmes in "The Adventure of Silver Blaze" – the dog that did *not* bark was a clue as to who did the crime. Setting minimum thresholds for workflows may identify problems as well; if nobody has logged in to your application for an unusually long time, that might be a sign that something is wrong with the login process. On the other hand, it might be New Years Day on a Sunday, so maybe nobody's logging in because they're not working today.

The final option for detecting a problem is customer support: if users are complaining to the support team that something is broken or slow, well, that's one way to discover the problem. Automated monitoring maybe can't find everything every time, but we shouldn't be relying on this as the primary mechanism.

Suppose that we've detected a situation that we think needs some attention. Niall Murphy says there are three ways to respond:

- **Alerts**: a human must take action now;
- **Tickets**: a human must take action soon (hours or days);
- **Logging**: no need to look at this except for forensic/diagnostic purposes.

A common bad situation is logs-as-tickets: you should never be in the situation where you routinely have to look through logs to find errors. Write code to scan logs.

Alerting. Alerting is pretty much exactly what it sounds like: a person is notified about a problem. We usually talk about it in the metaphor of pagers, as in the small rectangular box that wakes up doctors when patients need them. Except now we're the "doctors" and the patients are software. So as the person who is on-call (or on-duty, or on-shift), when one of the conditions for alerting is fulfilled, you get notified, usually with some beeping.

The alert may contain some information about the event that has caused it, such as telling you that Service A CPU usage is high, but there's rarely enough in there to diagnose the problem. It's off to the dashboard, then.

It is very important to be judicious about the use of alerts. If your alerts are too common, they get ignored. When you hear the fire alarm in a building, chances are your thought is not "the building is on fire; I should leave it immediately in an orderly fashion.". More likely your reaction is "great, some jerk has pulled the fire alarm for a stupid prank or to get out of failing a midterm." This is because we have experienced too many false alarms, so we likely think that any alarm is a false one. It's a good heuristic; you'll be correct most of the time. But if there is an actual fire, you will not only be wrong, you might also be dead.

Still, alerts and tickets are a great way to make user pain into developer pain. Being woken up in the middle of the night (... day? A lot of programmers are nocturnal, now that I think of it) because of some SUPER CRITICAL notification that says OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson that production code needs to be written carefully, reviewed, QA'd, and perhaps run by a customer or two before it gets deployed to everyone. Developers, being human, will probably take steps to avoid their pain[1]. and they will take steps that keep these things from happening in the future: good processes and monitoring and all that goes with it.

[1] There is a great quotation to this effect by Frédéric Bastiat about how men will avoid pain and work is pain.

Incident Reports. Another important aspect of responding to an incident is an incident report, or post-mortem. This is an opportunity to try to identify the root causes of the issue and anything that we could learn from the situation. If we don't find the root cause, it could happen again. And if we don't learn anything, it's bad for the company (the same or similar problems will recur), and also bad for your own career growth. Therefore a report is worth doing, even if it's not fun.

The first thing that an incident report should contain is a breakdown of what happened, preferably with a clear timeline of events. The purpose of this is to understand when the problem started, when it was noticed, what actions were taken, and when the incident was resolved. Strictly the facts here, please.

The most important part is analysis of why the incident happened. There are two kinds of causes, root causes and proximate causes. Proximate causes are the things that happen just before the problem or are the trigger that makes it happen. If you remember talking about deadlock, a deadlock can occur because thread A acquires lock X then waits for lock Y, while thread B locks Y and then waits for X. The proximate cause is the lock acquisitions of threads A and B, such as because User 1 tried to update a record while User 2 was trying to run a report. That's not the root cause, though, the root cause is actually we don't always acquire locks in a consistent order.

It's key to focus on finding the root cause rather than stop at a superficial level. If we say the root cause is just "I wrote a bug" the solutions to that are just... don't do that, or write more tests. Those are part of the solution but they aren't going far enough, because we also need to think about the root causes like the test environment is unrepresentative of the production environment, so it's hard to assess the actual impact of a change before it goes live. Or the root cause could be a lack of knowledge about a particular subject (e.g., concurrency).

Toyota (yes, the car company) uses a model of "Five Whys", which is to say you should ask "why?" five times to get to the actual root cause. I don't think it's necessary to always use exactly five, but it's a guide to making sure we're looking at the right level. If we don't ask the question enough then our answers are too superficial, but if we ask it too much then we end up with a root cause of "writing code is hard" and we end up debating whether computers were a mistake.

Then there are action items: what will we do in the short and long term to solve the problem and keep it from happening again? Things might be as simple as fixing the bug, but maybe monitoring/alerting need to be improved? Ideally we can even think of something that would prevent this kind of problem from happening again in the first place, such as introducing a testbench deployment stage that would catch performance problems before they go to production.

Action items, obviously, only help if you do them, so they have to be realistic. While you could say that you could prevent the race condition problem by rewriting all the company code in Rust at once, and that might even be factually correct, but it's also unlikely to happen.

And the final things are about what lessons we learned. This can be relatively straightforward – not every incident report results in head-exploding revelations – but there should be at least a few takeaways. Things we learned don't have to be bad: perhaps we make a conclusion that the monitoring system is effective and catches the problem quickly.

Lastly, there are some things that should not be in a report. Irrelevant detail is the easiest to explain: long reports that contain things that distract from the point make it less likely that people read it or take action on it. Speculation should also be avoided because it's often wrong; now sometimes we may be forced to proceed without being totally certain about why or what happened. Educated guessing and theorizing are okay, but speculation is just guessing without any basis. Finally, the report should not contain blaming or shaming; these are counterproductive for learning and changes.

Why is blaming bad? If you know you're going to get in trouble for an incident, it incentivizes covering up incidents or denying the problem. It also incentivizes people to spend time shifting blame to others or arguing about who is really at fault. There are lots of articles out there about how blaming results in worse patient outcomes in the medical field as compared to a learning culture.

But performance problems or downtime are not the only things you need to be looking out for... There's also making sure that attackers aren't abusing your platform and systems...

Security, Report to the Bridge

Having an always-available service accessible over the internet makes security a very big concern. You can run some program with tons of security vulnerabilities offline and feel that the security problems can be managed, but when it's online the risk is enormous. All kinds of vulnerabilities are a problem, but I'll call out two of them as being especially bad: code execution/injection and data leakage (information exposure).

Code execution is exactly what it sounds like: attackers run their code using your platform. They can mess with your data, send spam or harrassing e-mails to your customers, take services without paying for them, mine bitcoin at your expense, and many other things. Sometimes these are company-ending events. Actually, let's go on a quick digression about monitoring, with a real life example about detecting the very bad behaviour of crypto mining.

Abusing Free Services for Fun and Profit. Our source here is an article about how some security researchers found some malicious actors using build and deployment and other systems to mine cryptocurrency [Mor22].

The attack in question is about abusing the free (as in, no charge) CPU time and other resources that cloud providers offer. This is nice of them, and most use these resources for non-nefarious purposes, such as how the course notes for this course are auto-compiled using Github actions. No longer offering free resources like this would make it a little harder for attackers to abuse, but we also need to worry about a scenario where the attacker gets inside your network and can use the resources you are paying for to mine cryptocurrency for them. This costs money, yes, but also may slow down or hurt your actual operations since resources are limited, after all.

Part of the attack also involves bypassing normal signup limitations, including defeating CAPTCHAs and scripting the use of a web browser to make it look as though a human is browsing the page and clicking on the buttons. This suggests that malicious actors could swamp your services in fake/bot signups. Typically, there are some resources related to creating a new account, and if the malicious actor is creating accounts at a very fast pace, we might not be able to keep up, or at the very least every database will be full of extra data, so each insertion or retrieval becomes more expensive.

Once accounts are in hand, mining cryptocurrency in the way that's outlined in [Mor22] is not necessarily the most efficient route: given the number of accounts needed and the limitations of free tier resources, it's estimated that the attackers would need a few thousand free accounts to mine one coin. The return on investment is miserable from one point of view: using about $103 000 of resources produces one coin worth about $137, which is like a 0.13% return. Except, of course, it's Github who is paying the $103 000, so the return on paying nothing to get $137 is basically infinite. It's free money.

Of course, if someone looked at running jobs and saw a container called `mine-crypto-lol` deployed then it would certainly look suspicious, so the mining images and apps are given names to make them look much less conspicuous, e.g., `linux88884474` and `linuxapp84744474447444744474`. They also generate random GitHub action names to reduce the chance of detection there also. This makes it hard to detect, at a glance, rogue containers or processes.

Something like crypto mining may be easier to spot than other malicious activity, because it is CPU-intensive, so if we see some containers using a lot of CPU without an explanation or without knowing what the service does, we will likely feel compelled to investigate. But a container that's just sitting there quietly, observing things and sending your company's secrets to the competition or extortion rings? That's harder to find and could also be very costly.

Information Exposure. Information exposure is not only terrible for your company's reputation, but also against privacy laws and data protection regulations. A breach of the EU GDPR can get very expensive. See `https://www.enforcementtracker.com/` to find out which companies have recently gotten their wrists slapped or a huge fine. At the time of writing, the record holder is Amazon Europe (based in Luxembourg) with a fine of 746,000,000 EUR which is just over a billion CAD (using exchange rates from October 2022). Not exactly pocket change. Other big offenders are whom you might expect: Facebook (Meta), Google, but H&M – yes, the clothing store – is a top 10 in this category.

You will notice that a lot of companies get fined for things like insufficient technical measures. That is to say, they don't do enough to avoid the data getting into the hands of the wrong people.

One situation I've seen that is potential nightmare in the making is when personally-identifiable-information (PII) is put into the logs. As long as the logs themselves remain secure, then there's no problem... but if an attacker can view the logs, they've got a one-stop-shop for everything they should not be able to see.

Why log these things in the first place? The security policies forbade the developers from accessing – even in read-only mode – the databases, so to debug and trace the code, excessive data was being put into the logs. So developers could see what they needed then. That's actually a fun example of what happens when security policies backfire, like when password policies are so restrictive that it just leads to users writing the password on sticky notes.

Defending against Vulnerabilities There are some companies out there that will check your code for libraries with versions having known security vulnerabilities. This is transitive, so if your app depends on some library that depends on a parsing library with a vulnerability, the vulnerability is noticed and reported. Each vulnerability is usually assigned a severity and the service may even suggest that updating from version X to Y will solve the problem. They can say this because they observe that a vulnerability reported in version X of a library is reported as fixed in version Y.

Sadly, when there is an updated version of a library, there may be breaking changes in it, so an upgrade might be more painful than just changing a version number and rebuilding. It may also take time for libraries to correct and release a fixed version of the library. In the meantime, you may need to implement some mitigation of your own, or just keep an eye open for when the patch is released. To support your wait for upstream, the vulnerability checks give you the ability to ignore or snooze the alert. Tempting as it is to do that and get on with other work, it's leaving the vulnerability in your code. Good practice would be for reviewers to discourage ignoring if it can be avoided.

As with the other parts of managing your application, like build and deployment, checking for vulnerabilities should be an automatic process as part of your build and release procedures.

Near-Miss? `xz` and `ssh`. Early in 2024, a vulnerability was discovered in the `xz` library and Andres Freund posted about it on the mailing list [Fre24]. The library itself is used for data compression but some versions of `openssh` rely on it. This is a strong example of what's termed a "supply chain attack".

The term comes from the logistical supply chain. Supply chains for physical products are often complicated; if you want to build a gaming PC you need a large number of different components and you only get the end product when you've got all the pieces at hand. The problem is recursive, though: how many different components and pieces did, for example, nvidia require to make your graphics card? If one of the components – however small or seemingly-insignificant – is unavailable for whatever reason, the device (or computer) can't be built.

Similarly, almost any modern piece of software that isn't a toy or academic application is built in a compositional way (app A uses libraries B, C, D each of which has dependencies $E, F, G, H...$), compromising any one of the dependencies F could result in a vulnerability in A. But if A is `openssh`, you know, the *secure* shell daemon, being able to exploit that means being able to have root access on the vulnerable server and execute arbitrary code.

Part of what makes this example so relevant for this course is that the problem was discovered via performance profiling! The backdoor makes the `openssh` server run much slower due to the injected code that is running before the authentication actually takes place; see the example from [Fre24]:

```
before:
nonexistant@...alhost: Permission denied (publickey).

before:
real 0m0.299s
user 0m0.202s
sys 0m0.006s

after:
nonexistant@...alhost: Permission denied (publickey).

real 0m0.807s
user 0m0.202s
sys 0m0.006s
```

There's a lot of obfuscation in the implementation and the conditions needed to trigger the vulnerability are very specific, to reduce the likelihood of discovery. The details of the vulnerability and the social engineering needed to land it in a library are better suited for a security course than this one – but this is an important lesson in how it's not enough to just defend against vulnerabilities in the libraries or tools we know we use.

References

[Fre24] Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise, 2024. Online; accessed 2024-07-14. URL: https://www.openwall.com/lists/oss-security/2024/03/29/4.

[Mor22] Crystal Morin. Sysdig trt uncovers massive cryptomining operation leveraging github actions, October 2022. Online; accessed 2022-10-29. URL: https://sysdig.com/blog/massive-cryptomining-operation-github-actions/.