AppMap

The DevOps Hangover

After years of consuming countless tools and services to maintain high levels of performance and security, with little oversight for design or costs, the DevOps hangover is real.

The greatest irony is that DevOps aimed to help developers talk with operations, and vice versa, yet developers are even further away from understanding how their code operates at runtime.

Engineering and development teams, from startups to large corporations, have faced hard economic events over the last 12 months. Large-scale layoffs and company restructuring have forced teams to "do more with less" while building, scaling, and operating large-scale systems. Yet progress still needs to be made, and value still needs to be shown higher up the chain.

After years of consuming countless tools and services to maintain high levels of performance and security, with little oversight for design or costs, the DevOps hangover is real. The average company continues to use well over 100+ SaaS tools, and many companies are still investing in new tools and technologies to help them weather the current economic storm (Battery Report 2023).

The teams who can take this time to assess and streamline their arsenal will be the ones who emerge on the other side, cutting costs and improving efficiency, while gaining a competitive advantage.

So, what's the best medicine for our DevOps hangover? First, let's understand how we got here.

DevOps then: Party time, excellent!

DevOps emerged during a time of unbridled economic expansion with financial resources and the easy-to-use, pay-as-you-go SaaS products to support it. The problem has been two-fold:

Developers continue to use solutions that aren't perfect, as long as they are useful enough for their needs. However, when budgets become tight (like today), these tools or services may cause unnecessary financial stress.

And, developer toil was largely ignored as we yelled over our shoulders that it's ok to "move fast and break things!" But today, developers are spending more than 17 hours per week dealing with maintenance issues, such as debugging and refactoring, according to the Developer Coefficient report by Stripe.

DevOps now: Waking up to the hangover

Imagine you are now waking up to a request to do more with less, whether that is a reduced team size or a request to lower cloud and tools bills. With the loss of team members, you may be losing institutional knowledge that enables key applications to run. This loss can result in hundreds of thousands of dollars in outages, lost revenue, and regulatory fines. Despite these challenges, you may still be expected to maintain a five nines SLA to achieve aggressive growth targets.

It's not just engineers and operators who are waking up to the DevOps hangover. Executives, including CEOs, COOs, CFOs, and engineering leaders, are also taking a closer look at operational spending in relation to sales and growth. Many are now hiring consultants to help them map their cloud spending with the rest of their operational spending (like Observability and Data Platforms) in order to find ways to make necessary cost reductions.

What does the hangover cost us?

It depends on the conditions under which your development team is operating. Here are a few common environments:

- Complex legacy codebases
- A microservices platform that is growing exponentially
- A hybrid, stalled-out digital transformation project that is a Frankenstein's monster of monolith and microservices

Let's break down each one.

Cost of a legacy codebase

When a company's codebase becomes a monolith that nobody can easily understand, new features take substantially longer to release because it's much harder to add features and make changes when there are too many unknown upstream and downstream effects. This delay can be frustrating for both the development team and the end-users who are eagerly waiting for new features to be implemented.

Cost of a microservices platform

On the other end of the spectrum, Kubernetes and Lambda-native applications proliferated during the growth-at-all-costs era when cash was being burned at an astonishing rate. When running multiple products across shared underlying infrastructure, it becomes significantly more difficult to attribute per-product COGS (Cost of Goods Sold). This shift to a microservices architecture was made to improve the speed at which developers could deliver new applications. However, with shrinking budgets, already constrained Ops/DevOps/Platform teams are now pushing the ongoing management of these platforms onto the overburdened dev teams.

Cost of a hybrid monolith + microservices platform

And finally, in the middle of these two extremes are the teams stuck with both a legacy monolith, a constellation of microservices, and maintaining their five nines SLA with every code change. Teams are now faced with the challenges of breaking code changes, scaling issues, and the growing number and complexity of dependent services. And they are throwing all the observability tools at the problem to better understand how the code operates at runtime.

Autoscaling up is easy, Autoscaling down isn't

While understanding how to scale up infrastructure as your usage grows is important, it can be challenging to determine how to scale down when usage decreases. And many teams never spent the time to plan for a scenario where they would need to reduce the infrastructure as usage decreased.

If the cost of delivering your application becomes detached from your sales and revenue, your company may end up spending more money than necessary. This can happen if your cloud and observability bills remain constant, even as your business slows down.

Curing the DevOps Hangover

Every software company is asking its development team to cut infrastructure costs. Yet, the same problems remain: bad code creates blockages, outages, and inefficiencies that cost hundreds of thousands, sometimes millions, to the business, annually.

First, let's tackle operational costs.

Look beyond cloud spending as our only barometer for operational costs. The costs of delivering a product feature can exist over many services and across teams.

- Start with a spreadsheet or list and account for all of the other applications you use to deliver the entire application.
- Identify the value each feature or product provides, not only what sales and marketing say they do.
- Find areas of overlap between applications and teams. This is a great opportunity to identify and break down technical and organizational barriers built up.

Some costs for delivering your application are hidden in third-party SaaS services that follow pay-per-use or subscription models but do not register as your typical cloud expense. In particular, observability tooling and data infrastructure lead the pack for non-cloud infrastructure spend.

- Services like DataDog and NewRelic have extremely high costs related to custom metrics, and once a metric is added, it is unlikely to be removed later. Identify and attribute these underlying costs back towards the product or team supporting it.
- Review and understand which team owns which data sources. Take note of where data is moving within your applications and services. Many costs overruns stem from developers not understanding the cost impacts of data transfer between services.
- Identify where these tools are currently operating. In expensive production environments where it is costly to maintain these tools and more expensive to cure the issues? Consider migrating these tools earlier in the development process where mistakes are cheap to fix and the cost of running developer tools is lower.

Debugging in production may seem like the best way to solve the problem, but this will be after your users are already impacted, or after a critical code change balloons your AWS costs. It's simply too late to be effective.

Next, let's address the importance of code analysis and application design.

Code quality and design influence costs. Your senior developers know it, but it may be much less obvious to your CFO. Consider this:

1. A developer adds new code to your application, like a new feature or a bug fix.
2. That code change makes a repeated call to a database, increasing database queries.
3. Those increased queries cause application and database performance loss.
4. An SRE, siloed from a developer, is alerted to the poor performance and increases the underlying infrastructure to support the workload.
5. The cost to deliver this application has now doubled as the SRE team threw cloud at the problem and increased the database to a larger instance.

It is quick and easy to make a database larger, but in reality we simply masked a code quality issue AND increased our overall spend with no additional value being provided to the user. Over time, the code becomes riddled with performance issues that, in reality, never get fixed. Code quality suffers, as well as your bottom line.

For example, this type of N+1 SQL query problem isn't ONLY a problem for databases and slow-performing applications. Today's services are pay-as-you-go, where each API request has a defined cost, and inadvertently calling that same API repeatedly will rack up the bills.

Finally, let's acknowledge this will take continual improvement.

Remember, all hangovers will eventually pass. The ideal scenario is that you and your team can begin taking stock and evaluating your tools and services, and then you start seeing an improvement in the performance and costs of delivering your application. What can you then do to ensure things stay this way?

Remember that this isn't a static process. That's what got us into this mess. Make taking stock and evaluating your tools and services a quarterly step. Even better, include this work in your ongoing product development workflows. A little bit of low-friction software governance can go a long way toward reigning in wasteful and costly code changes and incorporating into your CI/CD means catching problems earlier in the process.

Even if you are unable to make a large investment into optimization today, understanding how your software is changing when it changes will be critical to stem the tide of growing operational expenses. Use tools to help you analyze your software during the development and testing phase in order to identify performance issues early and put those tools into the hands of the developers so they can better understand the operational impact of the code changes they are making.