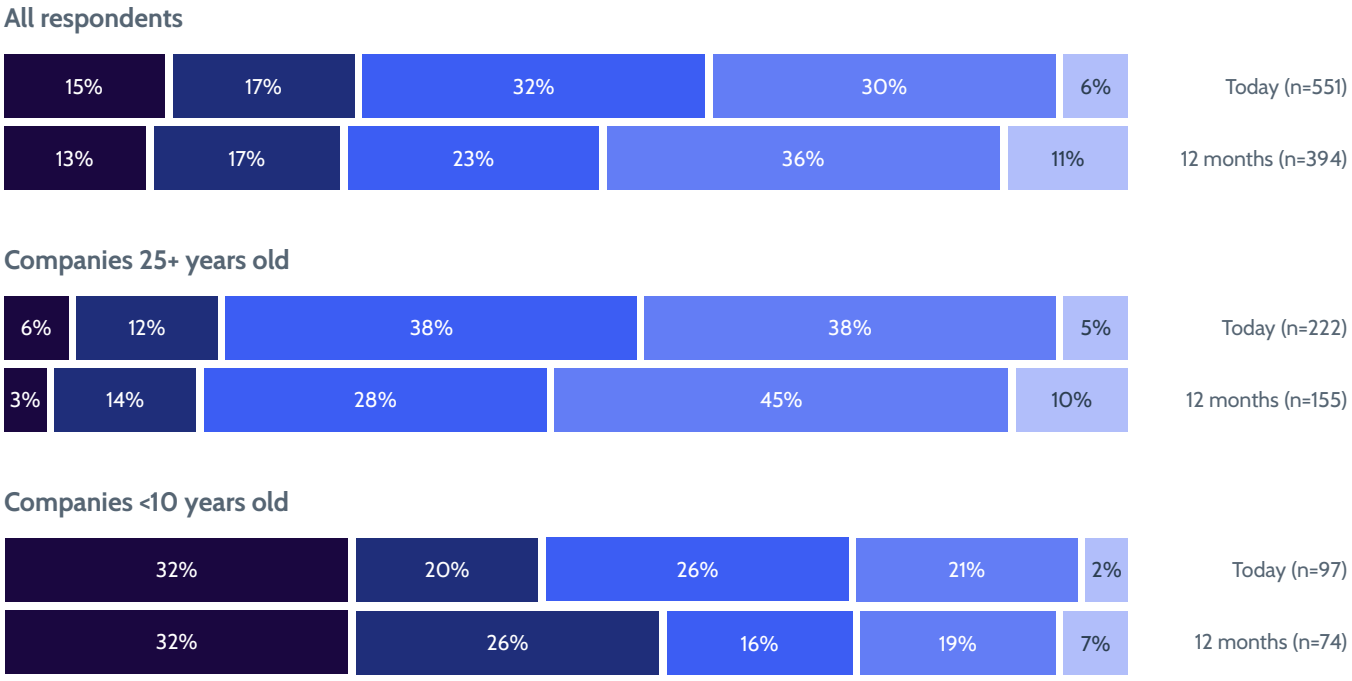
rewind

Automation in Cloud Operations

Whether you're a DevOps lead, Site Reliability Engineer, IT Manager, or CTO, odds are *automation* is pretty much ingrained in what you do. In many ways the terms automation and DevOps are synonymous. And in recent years it feels like the term has taken over, as there's almost no end to solutions that aim to *automate* something.

A recent report from Red Hat found that on average, companies have automated 36% of their cloud operations. The longer a business has been established, the more their manual processes have been replaced with automated workflows:



There are a variety of reasons for the rise in automation popularity:

- Companies want development operations focused on mission-critical tasks
- A desire to speed up timelines for releases
- Leaders want more agility and collaboration from teams
- Manual systems are notoriously prone to human error
- More demand for IT, SRE, and DevOps roles than there are people to fill them

More DevOps roles than people to fill them

According to a study from Bains & Company, DevOps talent was the highest in demand between 2015–2019, increasing 443% over that four-year stretch. In addition, the demand for software engineers during that same time grew by 69%.

The bigger question is, can those roles be filled? For the most part, the answer is likely no. Gartner predicted a few years back that 75% of DevOps initiatives will fail to meet expectations in 2022. Even at half the number, it's a staggering forecast. Although automation may not be the *only* answer to these challenges, in many ways it's being presented as a holy grail. Some even think we could get to 100% automation of DevOps tasks in this decade.

This gold rush mentality can blur lines between what's fact and fantasy. The concept of automation is associated with a range of definitions, philosophies, and an endless list of jargon. This short guide aims to separate what's helpful and what's harmful.

We'll cover:

- What automation means in a DevOps, SRE, and IT environment
- How to approach an automation project
- Build vs. Buy
- Shifting priorities and challenges in a growing company

There is no one right way to approach automation

To help us sort through the madness, we spoke to four software development leaders about their automation experiences, how automation fits into their team's goals, and the frameworks they use for making decisions:

Lena Feygin, DevOps Team Leader at OwnBackup

Lena is an experienced DevOps leader, continuous integration (CI) and continuous delivery (CD) architect with over fifteen years of experience in the software industry. She began her career as a full-stack developer but quickly built CI/CD processes and automation tasks. In 2011 when DevOps was in its infancy, Lena received a green light to form the first DevOps team and to lead the DevOps initiative in the R&D group in HPE. Since then she has enjoyed focusing on and leading DevOps initiatives. Her current role is leading a DevOps team in Ownbackup, based in Israel. Lena has a B.Sc. in computer science and an MBA in management.

Nigel Kersten, Field CTO at Puppet

Nigel Kersten is Field CTO at Puppet and is responsible for bringing product knowledge and a senior technical operations perspective to Puppet field teams and customers, working on services strategy, and representing the customer back into the product organization. He works with many of Puppet's largest customers on the cultural and organizational changes necessary for large-scale DevOps implementations. He has been deeply involved in Puppet's DevOps initiatives, and has served in a range of executive roles at Puppet over the last 10 years.

Scott Sturgeon, CTO at Tugboat Logic by One Trust

Scott Sturgeon is the CTO of Tugboat Logic by One Trust, with over two decades of software development and web application expertise. He is skilled in all aspects of the SDLC including requirements, design, development and deployment of complex applications.

James Ciesielski, CTO & Co-Founder of Rewind

James is the co-founder and CTO of Rewind, the leading data backup and recovery provider for cloud and SaaS data. After completing a Bachelor of Math, Computer Science/Software Engineering at the University of Waterloo, James has over 20 years of experience building highly scalable software and services in the fields of telecommunications, media, and financial technology in both enterprise and start-up environments. An experienced technical leader, James has successfully overseen the development and launch of a variety of software products, including Rewind's inaugural backup-as-a-service (BaaS) app, Rewind Backups for Shopify. In 2019, James was honoured as a member of the Ottawa 40 Under 40. When he isn't in front of his computer, James can typically be found running after his kids, cooking with his wife, and volunteering to be in net for every pick-up hockey game he can find.

Defining automation development operations

Like most things, the definition of automation is expansive and can vary based on who you talk to. To build a consensus on what automation is and isn't, let's revisit RedHat and their definition.

Automation is the use of technology to perform tasks with reduced human assistance. Any industry that encounters repetitive tasks can use automation, but automation is more prevalent in the industries of manufacturing, robotics, and automotive, as well as in the world of technology—in IT systems and business decision software.

> **Insight from Scott:**
>
> "From my perspective, automation has become a buzzword. Not everything can be automated in a sane way. Not everything should be. Things that are repetitive and time-consuming, are the things that I tend to focus on."

Already we are starting to see how broadly the term automation can be applied. This guide focuses more on development operations and IT systems. So when looking at building software, are there distinctions that we can make? Here is what Google says in their SRE Handbook:

In the industry, automation is the term generally used for writing code to solve a wide variety of problems, although the motivations for writing this code, and the solutions themselves, are often quite different. More broadly, automation is "meta-software"—software to act on software. Automation is a force multiplier, not a panacea. Of course, just multiplying force does not naturally change the accuracy of where that force is applied: doing automation thoughtlessly can create as many problems as it solves.

According to this definition, automation is not just about removing any and every manual task, it's about thoughtfully selecting what tasks you should automate and why. The objectives of a DevOps team is to speed up and create more efficiencies in the process of building, testing, deploying and maintaining code. Most people agree that automation can help achieve this goal. The term CI/CD (continuous integration, continuous delivery/deployment) is now commonly used to express this.

> **Insight from Nigel:**
>
> "I think automation is a very generic term and you want to be more specific about what you're actually going for. It's a technique and method, rather than an outcome. You don't automate for automation's sake, you automate because you're trying to achieve a result. I also think it's a good marketing buzzword because it resonates with the folks on the ground who think, 'I can make robots do it.' So you have to be cautious about how you're using that word."

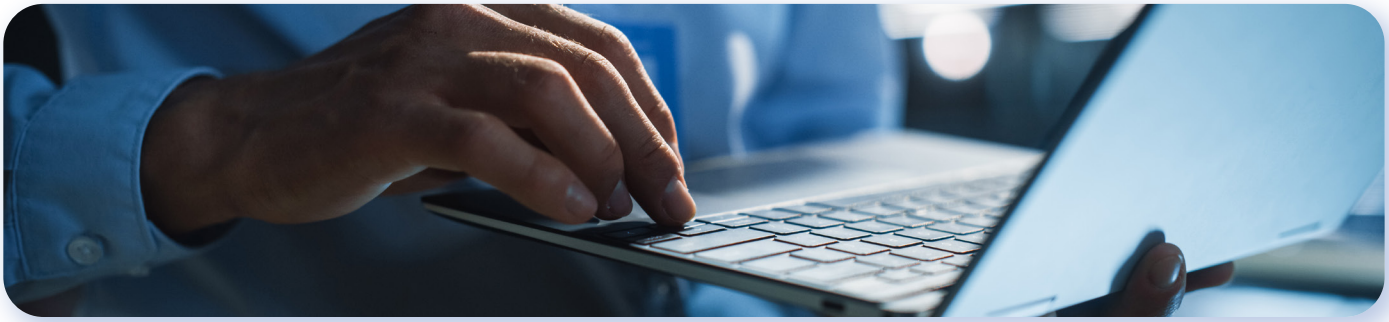
Automation shouldn't be limited to simply serving the needs of development operations. Here's an excerpt from Puppet's 2021 State of Dev Ops Report:

One of the most unfortunate incarnations of DevOps we see, particularly in large companies, is treating it purely as "Developer Operations," focused entirely on the care and feeding of CI/CD pipelines. Build engineers bridge the gap between development and production, and so it's understandable how we ended up here—yet this is not DevOps.

Software development is not the same as an assembly line in a factory. In many ways, software development is a social endeavour, a team sport. People and processes are not automatically interchangeable. Yet sometimes teams approach automation as if the tasks people can do can easily be automated and sustained. Or things automated in one environment or business culture can be copied and pasted into another one. The reality is that automation is context-specific. The things you should automate and how you should approach them are all relative. In other words, everything can be automated but not everything *should* be automated.

> **Insight from Lena:**
>
> "Every company has a different culture. For automation, you cannot copy, paste and implement a template from one thing to the next. When I start a new job, some processes may seem a bit crazy, but after a couple of months of understanding and learning the culture, it usually makes sense.
>
> I worked in a large company that had lots of production environments and no staging environment. They deployed directly to production—cautiously, of course, monitoring the process—and surprisingly enough, it worked well. 'Best practices' would suggest you need to go through staging environments first. So an outsider may want to fix that, however, an insider would know better. In the end, we only need to automate issues that really cause problems and not fix problems that don't exist.
>
> Of course, there's always a place for improvement. That's why it's important to remember that nothing stands still and we can always do better: we want to improve continuously."

If what you automate is situational dependent, how do you decide what to automate? Are there standard tasks which are always beneficial to automate? In the next section, we will cover some of the commonly held beliefs about what parts of development operations should always be automated.



What to automate

There are dozens (if not hundreds) of articles talking about what to automate in DevOps. The majority of the advice revolves around software delivery, lifecycle deployment, change management, monitoring, alerting service, and recovery.

For most companies there's a high cost to getting these things wrong. There's also significant value in making these processes as efficient as possible. When looking at the consensus about what to automate, here is a rundown on things you may want to consider:

- **Testing:** The goal here is to ensure your tests are consistent and repeatable, and to reduce the reliance on people. It also helps monitor unintended changes to existing functionality or when new things are introduced. Teams can achieve this with scripts or third-party tools.
- **Provisioning:** Often synonymous with Infrastructure as Code, automating this replaces manual work around delivering computing resources when needed. It also builds reliability and consistency in provisioning. You're guaranteed that the infrastructure you provision today will function exactly the same tomorrow. Some examples are configuring encryption on a database or some data, backups & recovery for a database, or specific configuration flags that need to be enabled for performance reasons.
- **Deployment:** Often considered of the main things to automate. It improves development lead time by enabling teams to release features more quickly and frequently, reduces errors due to limited human involvement, eliminates the need for a "deployment gatekeeper" and ensures nothing breaks your core functionality.
- **CI/CD:** We've already mentioned but it's worth mentioning again. Continuous integration (CI) and continuous deployment (CD) processes can be automated to support agile monitoring, integrations, and testing to deliver and deploy changes to the application faster.
- **Infrastructure Management:** Essentially automating the tasks which control the hardware, software, network, and operating systems that deliver IT solutions and services.
- **Observability:** The continuous oversight of an application's performance, which increases in difficulty as operations grow and new features are added. Can also include log management. Since every piece of software creates log events, automation can help with collecting, storing, searching, and even deleting logs.
- **Security (DevSecOps):** The need for more secure releases is growing and can be a headache if manual tasks pile up.

> **Insight from Scott:**
>
> "Most of the automation you hear about these days comes down to CI/CD –continuous integration, continuous deployment. You can build out very complex pipelines that can automatically update your production instance. Which is great, if you're set up to do that properly."

Insight from James:

"When it comes to deciding what to automate, we assess the risk of having people involved in maintaining a process. I've seen development environments where people were granted God-like privileges to the development team. Often this is done in the name of serving customers but people make errors, intentional and unintentional."

Yet this doesn't mean *everything* should be automated. Google coined the term toil, which is a way of discerning between work that is essential and tied to the productions and processes of running a service or product. Non-toil (or "grungy" work) is any task not directly tied to production. This could be team meetings, filling out timesheets, setting personal goals, etc.

Insight from Lena:

"Take creating a new feature as an example. The amount of time spent writing code is usually ten, maybe twenty percent of the whole time until the feature goes live. Maybe another fifteen percent checking for bugs and fixing them. The majority of the time is spent waiting for something - for builds, for tests, for an environment that maybe doesn't work.

I call this *the dev experience* and many companies don't invest much in it. This causes developers to have a lot of waiting time, instead of spending time writing code. As the company grows, it should invest in automating the dev experience: make it easy to start coding a new feature, facilitate fast and reliable tests, and ensure environments that are up to date.

You also need automated processes for maintaining production and this is something you'll likely want to automate from the beginning. Even just a small human error in production can cause companies to waste a lot of resources in the form of time and money."

As the SRE team at Google sees it, toil isn't inherently bad, in small doses it can be manageable. However, left without addressing it, it can start taking over more and more of an engineer's time.

Insight from Nigel:

"I'm a big fan of eliminating toil. This is work that's manual and repetitive, doesn't create systemic improvements in your service or application, and generally doesn't provide enduring value. I think a lot of teams struggle with building up trust across the organization, as they're rolling out more automation. The more you can start solving big problems, that everyone agrees is a big problem, the faster other people trust you to automate more stuff."

The reality is that many businesses will need to automate things that directly or indirectly impact their product or service. Unique use cases can arise in any environment, especially as markets become more competitive, more SaaS companies come online, and road maps get more ambitious. But as any development team on the planet will tell you, their To Do-list is impossibly long. As a team, how do you prioritize and agree on what to tackle next?

What *not* to automate

Although most tasks can be automated, it doesn't mean they should be. Here are some quick tips on what can be removed from the *to be automated* list:

- Tasks with a low return on investment, where the amount of effort required doesn't produce any significant returns
- User experience testing or anything where you have subjective results
- Tasks where there's a very high degree of unpredictability
- Tasks with a low amount of repeatability
- Tasks with no well-defined procedure/pattern

> **Insight from Scott:**
>
> "Things that a computer can't do well should not be automated. For example, a lot of companies try and automate code reviews. This is fine for things following syntax or weird circular dependencies. A computer can figure that stuff out. However, looking at the logic of a piece of code and whether or not that's done properly, isn't something a computer can do well. A code review in my mind still requires a person to look at it."

> **Insight from Lena:**
>
> "I know there are a lot of tools right now, trying to analyze and come up with reasons why something failed in production. But after a decade of working in DevOps, I find people are still the best resource for figuring out what happened. A person who knows the system in and out, often has a gut feeling about what's happened and can address the issue and find a solution quickly."

Signs to automate tasks

The Google guidebook, in the section about toil, includes these suggestions for when a manual task needs to be addressed:

1. **Manual:** Like manually running a script that automates a task. Although the script may be quicker than actually executing each manual step, the time it takes to run a script is still time that could be used elsewhere. In other words, if I have to keep pressing a button, I may eventually spend all my time pressing that button.
2. **Repetitive:** After about the second time of performing the same task, it may need to be automated. Solving new problems doesn't count.
3. **Automatable:** If a machine can do the same job, just as well as a human it can be considered toil. If the task needs a human to really think about things, it may not be a good candidate.
4. **Tactical:** If something keeps interrupting your team and pits them in "reaction "mode, minimizing these distractions could be something to address.
5. **No enduring value:** If you perform a task, and your product or service remains the same, it probably toils.
6. **Tasks grow as operations grow:** If the magnitude of a task increases as service size, traffic volume or user count—it's likely an issue.

In the end, often a good starting point is finding processes (which add value) that you are doing frequently and relying on a human. When people are involved, the chances of error go up exponentially.

Insight from James:

"When something is consuming time, and preventing us from making progress elsewhere, that's a good sign it should be automated. For instance, if we ever find ourselves with a plan to deliver customer value and within that plan, we have to allocate some percentage of time to a manual task or process that is not otherwise automated, then we are delaying the time to delivery of that customer value which is not good for our customers nor our business."

Insight from Scott:

"As your system gets more complex, there are more moving parts. There's more opportunity for human error. If you make a mistake during deployment, now you're bringing your system down, you're affecting customers and everyone's in a panic to get it fixed. So get that stuff all automated and test it properly. For us, it's pretty much just a one-button click for this."

Insight from Nigel:

"Any manual work being done on behalf of a large set of users is a good place to start. For example, say you work on an infrastructure team and you regularly need to open a port on a firewall. If there are ten people on your team but 5,000 developers in your organization, creating a ticket to open a port, may not feel like a lot of work for you, but it creates waiting time for many others. Using an automated self-service offering removes the waiting time. This also saves people from having to switch contexts while they wait for it to be done. So they are more productive and it reduces their extraneous cognitive load."

Insight from Lena:

"One lens we use is assessing anything which can cause a huge problem with production. Once we had an issue that caused a system to go down for a day. That's a huge risk. So I would want to explore automation solutions to reduce our risk. Another one is determining how much time a developer wasted to get to the point that their feature is done."

Build vs. Buy

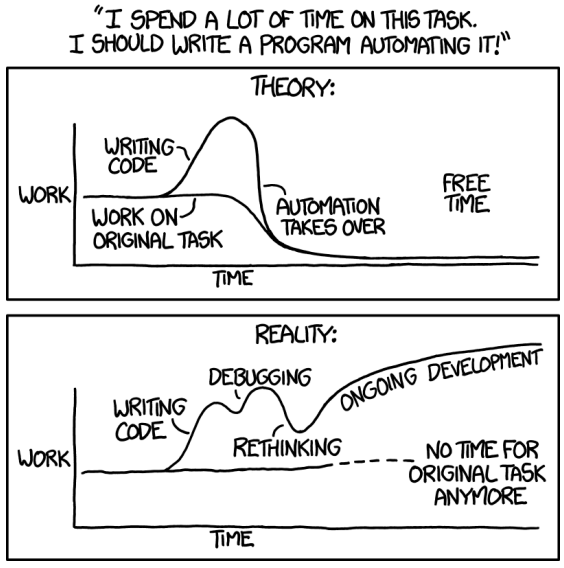
There is an incredible temptation for teams to want to build their own automation and tools.

Although it may save on overheard or budgetary line items, it may cost more in terms of time and people resources. When evaluating whether to build or buy, the answer usually comes down to...it depends. Here are some things to consider:

- Any **security and associated risks**. Once you've written something, often you can find vulnerabilities that need fixing.
- Consider the skills and **core competencies** of your company. Does what you want to build align with the main offerings of your company or your developer's skill set? If not, building with internal resources may cause challenges.
- Determine if you have the luxury of **diverting resources** from other projects.
- **Budgets** are finite. You don't need a formula one car when an electric bicycle will do just fine—try building your own.
- Can you **maintain** it? Software is not a *one and done* exercise. For example, languages and libraries you are using become deprecated or unsupported, problems occur with processing data patterns you didn't expect/hadn't seen before, or the infrastructure you are using becomes de-supported or worse, discontinued. If you can't keep something running internally, maybe externally outsourcing or finding a managed solution is the way to go.
- Is this an **urgent requirement** or something you need down the line? If it's urgent, off the shelf may be the way to go.
- What's the total expected **lifecycle cost** for either option? Teams often focus on the initial build or buy costs, without factoring in the additional costs and resources over time.
- Will building give you a **competitive advantage**? If not, maybe it's worth outsourcing.

> **Insight from Lena:**
>
> "I don't want to really manage a database if AWS knows how to do it better than me."

These are just some of the things to think about but in the end, it's a judgement call you need to make at the time, factoring in your resources and goals. But whatever you decide, always revisit and re-evaluate, as a company's needs always change over time.



Insight from Scott:

“Think about whether you're going to be better off in the end. Think about its maintainability over the long term. If you do all this work to automate something and then spend a ton of time just maintaining it, then you're probably doing the wrong thing.

You have a finite number of resources and I have a huge roadmap of application features I need to deliver. I can’t have all my devs sitting back and building all these tools. They need to be working on the product.

I've seen automation done in a highly complex way that nobody really understands how it works. One little thing can break everything and then you're spending months trying to figure it out. The sane way of automating things to me means easy to maintain and easy to understand."

Insight from Lena:

"There is no black and white answer here. It depends a lot on a company and its goals. We have actually developed a lot of custom processes but it takes time and manpower. If you build something in-house, you will have to maintain it. There is no such thing as “it works forever”. So before building a customized process, one needs to consider all the aspects and see if it’s really worth it."

Insight from James:

"Think of something like Lego. I could actually 3D print my own Lego but it doesn't make sense to do that if a tool like Lego already exists. In development, the same logic applies, just because we can build something in-house, doesn’t always mean you should."

Automation tips from growing SaaS companies

The reality is what got you there today, likely won't get you there tomorrow.

As your business grows, the needs and priorities will shift as well. So your automation strategies will need to account for this as well. Here are some things to keep in mind:

- **Build consensus early:** Development is a team sport so you need everyone's help and buy-in. Explain and educate how a certain project will help move the needle.
- **Document processes and standardize:** Silos will hurt you so create guides and procedures that everyone can follow.
- **Implement changes gradually:** If you try to tackle everything at once you will be overrun. Taking incremental approaches is best.
- **Prepare for scaling-up:** This is not always easy but do your best to prepare for the future. Building in the ability to scale up or down will help the business as its' needs evolve.
- **Evaluate tools that integrate with other systems:** Tools that solve specific issues are great but sometimes ones that allow for multiple integrations provide more flexibility over the long term.
- **Consider managed solutions over open-source tools:** Again this comes down to "Who's going to run it?". Just because you have a problem, doesn't mean you need a shiny new toy or can afford to divert resources to run it. Picking your battles early on can save future headaches.
- **Avoid proprietary tools:** tools that are tied to one code base or operating system reduce your agility and flexibility.

> **Insight from Lena:**
>
> "As you grow, one of the main issues is scaling up: either in development processes or production environments. Some processes work when you're small, but when you need to scale up, everything changes.
>
> For example, as soon as an application grows and gets more complicated, people are just not capable of manually testing everything. When developers finish a feature, they need a clean environment to test their code before they deploy it to production. In the beginning, when there's a small number of developers, that shouldn't be an issue. When you have over 100 developers, who need to test their code, you don't want to create 100 environments manually or run the tests manually to make sure nothing else got broken. It just doesn't make any sense.
>
> One important thing I've learned is not to try and build a perfect process at the beginning, but rather break it into small milestones. Early in my career, I always wanted to create the perfect process. The reality is sometimes it's enough to spend two weeks doing something to solve 80% of a problem. You can improve it later. This frees up everyone's time to focus on other stuff."

Insight from James:

"Here's an example of *critical* automation that was tied to our growth. Rewind 1.0 was a traditional web application: there was a UI, and some back-end logic, and it all sat on top of a single database. The back-end expected the database to always be in a certain state—if it wasn't we could have major issues.

There were points in those early days where fixing a problem or getting specific data required manually connecting to the database to make changes or fetch data, so that's what we became accustomed to doing. The risk was small in the beginning because the number of customers was low and if we changed something that made the back-end unhappy, it wasn't very critical that we fix it in a hurry.

As we iterated on the service and became more successful, the risk of manually changing the database grew. To mitigate that risk, we created a simple admin app which became the back office gateway to making service changes.

In the early days, we took a more traditional approach to software delivery. We built a bunch of features, aggregated them together into a release, tested the release, and manually pushed that release into production. When we were a mono repo, this was relatively easy to manage. As the product became more complicated and the team grew, so too did the release process—to the point where a release was a "stop the world" process that literally halted development and our production systems while we pushed code

To speed up our processes, we switched to trunk-based development and invested heavily in our CI/CD pipeline, test automation, and feature flagging. Every time a feature was merged, it went into production. This change gave developers the freedom and confidence to constantly make changes. Our lead time on features went down by a factor of 10x and the quality of our deployments went up by the same margin. I kick myself for not making this a day 1 priority."

Insight from Scott:

"In the beginning, our team was me and two other people. So automation wasn't really a big deal. Where we did do automation immediately was testing. Writing tests that have code coverage across your entire application, they run on commits, you can run the tests ad hoc, that way you know stuff works. You can refactor things easily and be confident that you didn't break anything because the tests still pass.

When you first start out, you're iterating super quickly on your product, trying to find market fit. So you're blowing stuff up all the time. Being able to do that with confidence, that's mission critical for rapid iteration. So that was the first automation that we focused on.

Here's another way to look at things. When you first launch your application you're kind of like a car from the 1960s. It's not super high-tech and anybody can take it apart in their backyard and then put it back together. And then as you grow and get more sophisticated, well, now you're a modern car. It's actually now more computer than car and you can't really mess with it too much yourself."

Insight from Nigel:

"When going from small to large, some of the practices that are valuable at the beginning, become counterproductive when you're larger. An example is teams picking their own tooling and deciding what to automate for themselves.

If you're a small company and you've got two dev teams, one's using GitLab and one's using GitHub, it's probably not the worst thing in the world. But as you grow, if you kept fragmenting like that, it's just expensive for the business. I think there's a real balance there between autonomy and optimizing for the larger organization. And things like compliance and governance start to be a problem.

To get ahead of this you want to build communities of practice. Some people are passionate about the tools they actually use and some people will just use whatever—whatever wrench works to get the job done. It's a good idea to collect these people in a loose format, like an architectural review or create a cross-functional practitioner group with input into these things.

You should also be building all of your applications so they deploy through an empty CI/CD pipeline. It's called the walking skeleton model, which is the minimal implementation of application architecture. This makes it easy and cheap to put a test in. So if you're running a retrospective on an incident report and someone says, 'Hey, we could fix that by just having a simple test that did this,' the cycle time of being able to actually implement it is really short."

Author: Dave North, VP of Cloud Operations at Rewind

Dave North has been a versatile member of the Ottawa technology sector for more than 25 years. Dave is currently working at Rewind, leading the technical operations group. Prior to Rewind, Dave was a long time member of Signiant, holding many roles in the organization including sales engineer, pro services, technical support manager, product owner, and devops director. A proven leader and innovator, Dave holds 5 US patents and helped drive Signiant's move to a cloud SaaS business model with the award-winning Media Shuttle project. Prior to Signiant, Dave held several roles at Nortel, Bay Networks, and ISOTRO Network Management working on the NetID product suite.

Our vision

In the next 10 years, most businesses will be SaaS or SaaS-enabled, which means hundreds of petabytes of vital business data would be left at risk without our help.

Not everything you do online gets saved in the cloud. All of the data that you rely on daily to run your business is your responsibility to protect in case something goes wrong.

That's where we step in to help. With the simple click of a button, we help you restore your SaaS data back to a time when everything worked perfectly so you can focus on the things that matter.

rewind