2021

DevOps Setups

A Benchmarking Study

humanitec

Table of Content

Executive summary

The gap between top and low-performing engineering teams is dramatic, whatever angle you look at it from. Whether you analyze their tech stacks and architecture choices, performance metrics, or cultural and team elements, the delta tends to be quite impressive. Some teams use public clouds, some teams use private clouds. In some situations, microservice applications are the right choice, in some cases monoliths. Despite the wide range of approaches, a few indicators paint a clear picture of the typical characteristics of a well-oiled setup.

We're getting to the root cause when we zoom into the cultural aspects. How do teams manage DevOps? How is ownership distributed and thought about? This is where the data is most definitive and where teams that want to drive change can actually start off with. 100 of 100 developers in high-performing teams report they are running a true "you build it, you run it" setup. A setup so streamlined and optimized that it doesn't get in their way through overly complex design or restrictive abstractions. In comparison, only 3.1% of low-performing teams report this. In 52% of cases, low performers are still stuck in a model of "throw over the fence" operations. In 44.9% of cases, developers are left alone and develop a "shadow operations" model, where senior developers are taking over the role of operations to help less experienced developers, creating all sorts of inefficiencies.

It is vital for teams to hone in on the optimal tradeoff between the complexity of their delivery setup and the cognitive load it forces upon developers. Understanding this is the key to determining the optimal "hand-over" point between developers and operations. Beyond this point, opaque abstractions risk only limiting developers needlessly. Such abstractions should be developed "as a product" by an internal platform team that constantly automates and streamlines developer experience in conversation with their internal customers - the developers. This is the only way teams can reach true "you build it, you run it", which, from all data points we looked at, is the main driver of high performance. The leanest way to nail such tradeoffs is by deploying Internal Developer Platforms.

Introduction

Throughout 2021 we used the reach of our brand to offer teams a framework to benchmark their DevOps setups against other teams' in our databases. Our rationale for it: most studies tend to stay fairly blurry when it comes to the details of teams' everyday workflows and practices. Research suggests that the majority of engineering organizations are seeing their DevOps ambition stale, but how this actually looks like on the ground is not clear.

The response to our study was much better than we expected. We ended up with 1,856 organizations providing deep insights into their DevOps setup. In the survey, we covered three key aspects:

- Tech and Infrastructure
- "Classic" DevOps key metrics such as Deployment Frequency, Lead Time, Mean Time to Recovery and Change Failure Rate
- Team Setup & Culture

The dataset includes teams of all sizes, ranging from a dozen to hundreds of developers, with a focus on the regions "US" and "EU". Respondents primarily use the following titles to describe themselves on Linkedin: DevOps, CloudOps, Infrastructure, SRE, Platform Engineer. For a more thorough view of the data and a discussion on statistical significance, bias, etc. please refer to the appendix "Underlying Data".

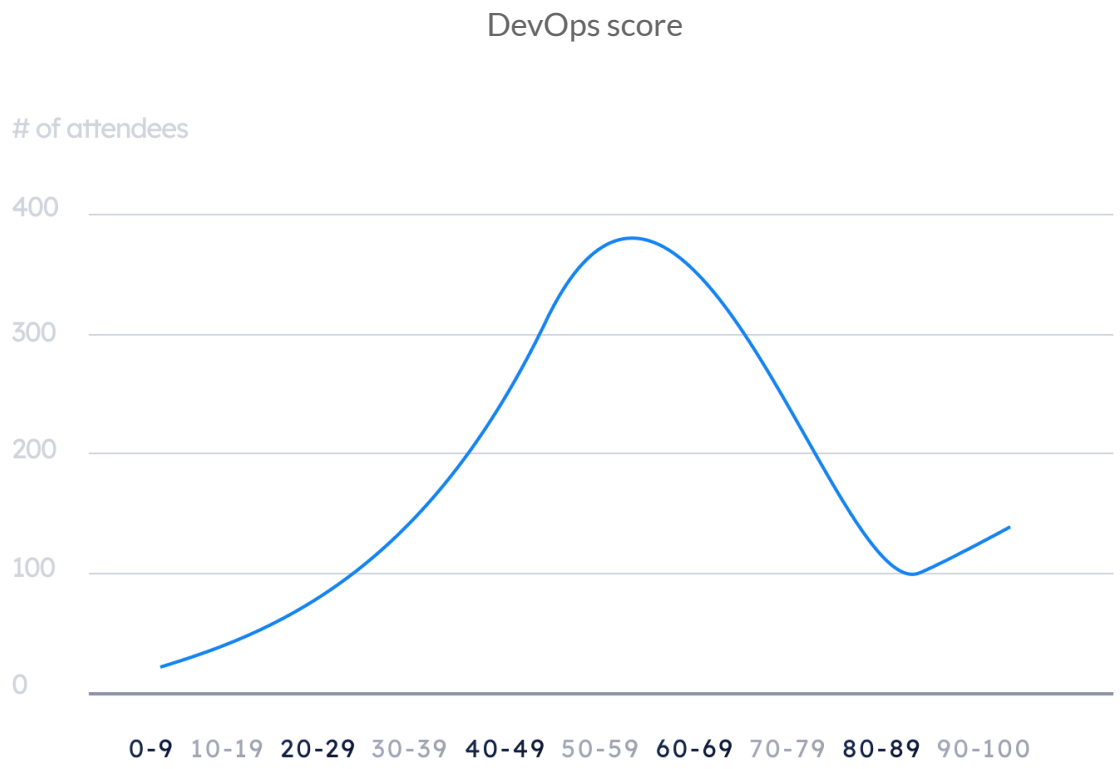
In this report we will try to answer two questions:

1. If we talk about a company with "good" DevOps practices, what does this mean in terms of their KPIs, team setup, technical setup, and cultural approach? What are the key differences between low and top performers?
2. What can low performers do to cross the "DevOps mountain of tears" and become top performers?

The DevOps mountain of tears

In order to benchmark teams' performance against one another, we used a DevOps Maturity Score (DMS) computed from all data we collected. Depending on the answers of the survey, a DMS between 0 - for indicating a low-performing DevOps setup - and 100 - for excellent DevOps performance - was calculated. The assumptions that went into the scoring are based on metrics and concepts that are accepted industry-wide. To give a simple example: we probably all agree that a team that enables true "you build it, you run it" across the org should receive a higher score than a team with a dedicated sysadmin team that receives config files "thrown over the fence". Or that a deployment frequency of a month should be tagged with a lower score than a team that deploys on-demand. So although a simplification, the DMS should be a fairly accurate representation of the state of a setup. Below, you can see all 1,856 setup scores plotted. It's a powerful graphical representation of the "DevOps Mountain of Tears" that so many teams have problems crossing.

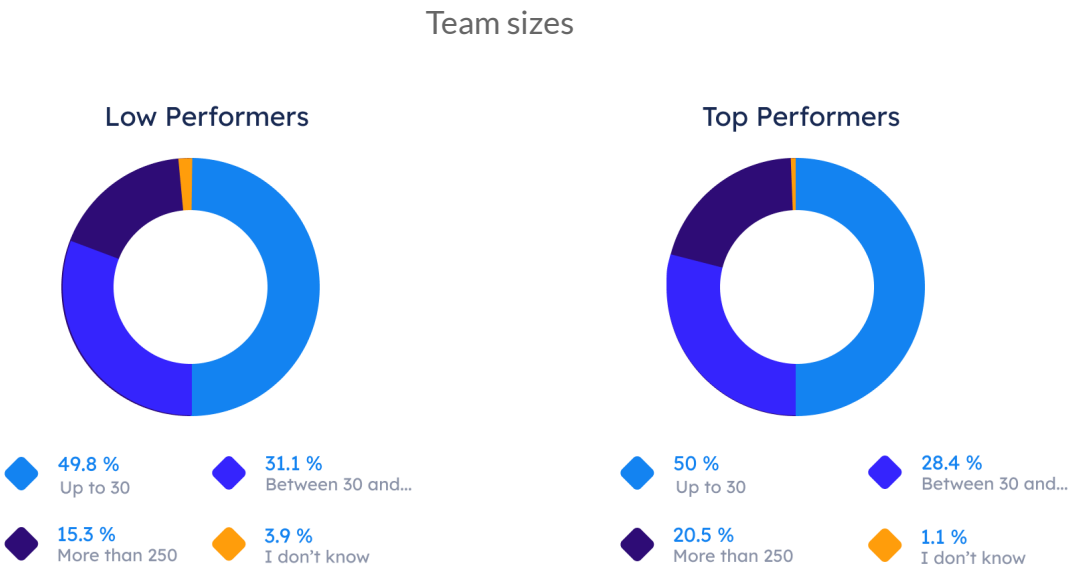
Median and average scores are both around 53. DMS distribution:



To gain a better understanding of what differentiates low performing from high performing teams, we clustered the data into four cohorts that we used for our analysis:

- Top performers with a score ≧ 90
- High performers with a score between 70 and 89
- Mid performers with a score between 40 and 69
- Low performers with a score between 0 and 39

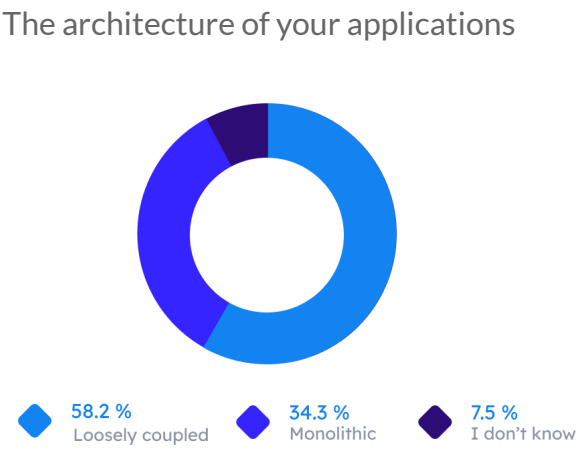
We mapped all scores against these cohorts and will analyze them in more detail. We also made sure there weren't any other key drivers and variables that heavily influenced results (such as the number of developers in a team). We tested this thoroughly and could not identify any particular bias that would be intrinsic to using the cohorts for the analysis. Mapping the number of developers against the score provides a first glance at relative consistency across - for example - team sizes:

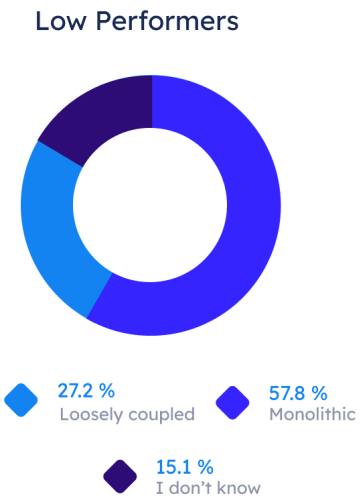


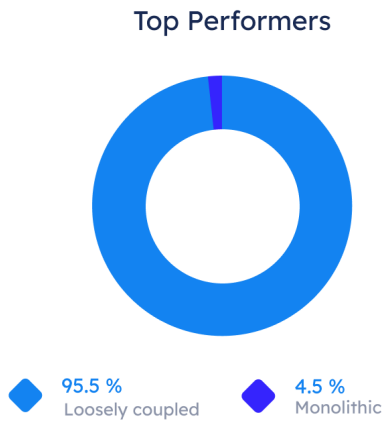
Overall results

Tech stack & architecture

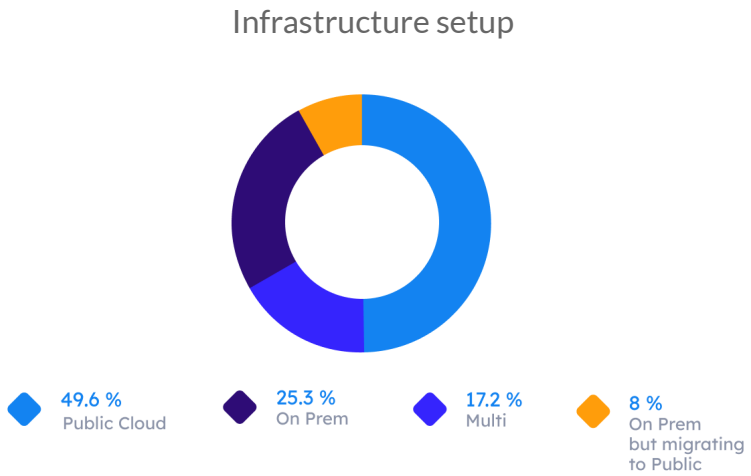
Let's start with the trends in application architecture. If we look at the entire data set, the monolith is still surprisingly dominant, present in 34.3% of all setups. The picture is a lot more differentiated if you cluster based on the four cohorts. Low performers run almost twice as many applications that are monoliths (58%). Top performers run all loosely coupled architectures in 95.5% of their applications. There's no need to get into the old microservices vs. monolith debate, as it is always going to be very specific to the use case. But the difference is really quite large. It's also not super surprising, as you need quite an advanced system to run microservices at scale effectively.



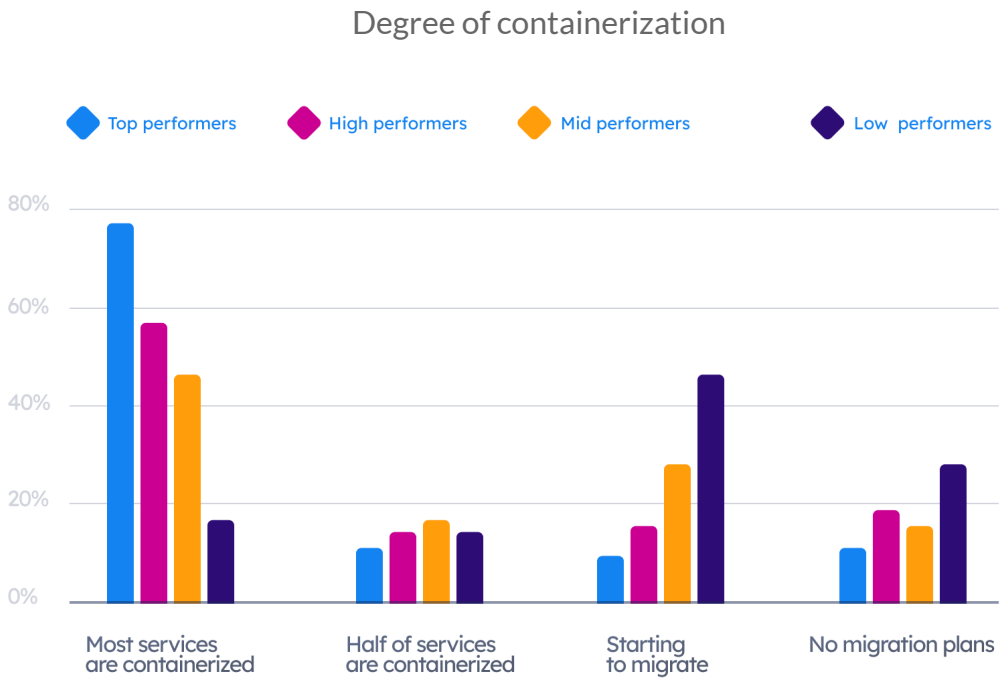




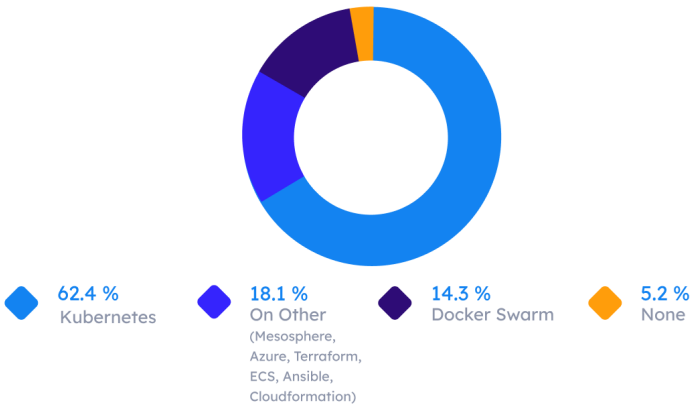
On the infrastructure side, results are in line with expectations and with previous findings. Multi-cloud is the reality of only 17.2% of teams. Public cloud is the dominant approach, especially with top performers. And there are more low performers running on-prem than the other way around. It is important to note here that on-prem setups tend to exist more often than not in regulated industries with higher security requirements. This would influence the DMS without necessarily telling us too much about the maturity of a setup.



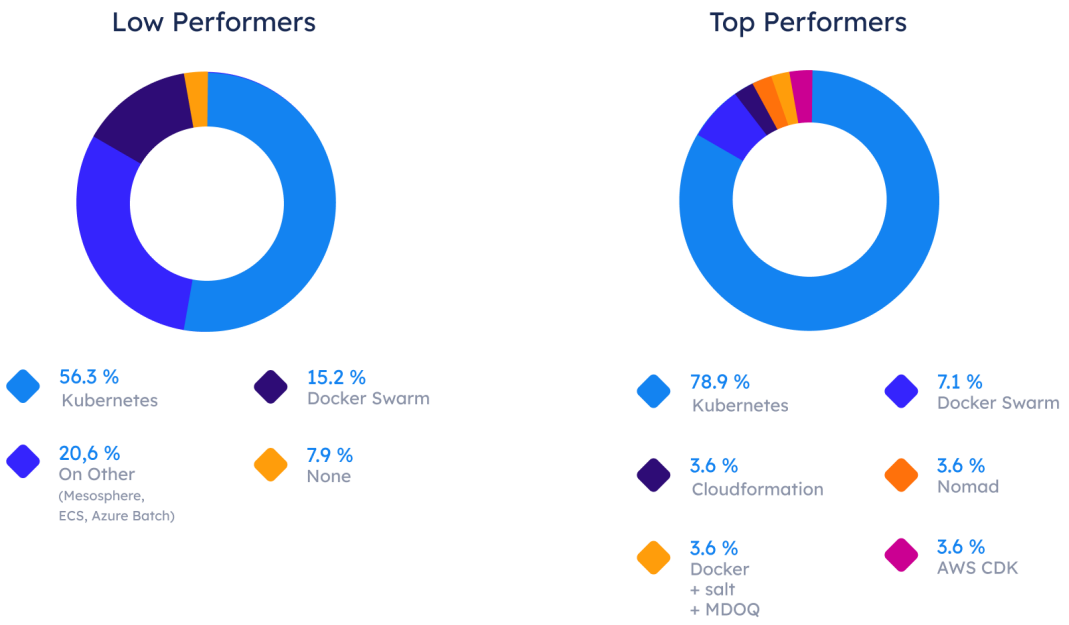
What does vary considerably is the degree of containerization. Across all teams, 82% are either already fully on containers, currently migrating, or currently starting a migration. 18% are not planning to ever start a migration. Interestingly enough, this again varies across scores. 93% of top performers are adopting containers and most of them are already fully migrated, while this is only true for less than 20% of the low performers. More than twice as many low performers report they are “never going to migrate”. Aging ecosystems and serverless are the remaining options.



If you're looking at how all of these containers are being orchestrated, the picture is pretty consistent and in line with expectations. Kubernetes is the new legacy and solution of choice for more than 62% of dall teams. Docker Swarm is slowly disappearing. Every time we run the survey, the number of users on Docker Swarm keeps shrinking. The same applies to Mesosphere. The tooling landscape is fully behind Kubernetes at this point in time. You get a feeling of where the future is going if you look at top performers. Almost 80% are now all-in on Kubernetes vs. 56% for low performers. That said, it is important to point out Kubernetes is not the solution to all problems. If you have a mediocre setup and move it to Kubernetes, chances are you'll be worse off in the end.



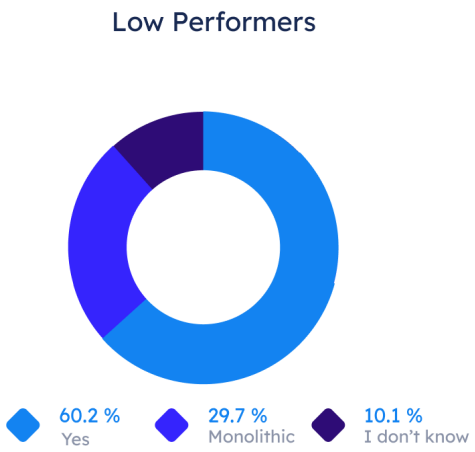
Example: What is your orchestration tool of choice?

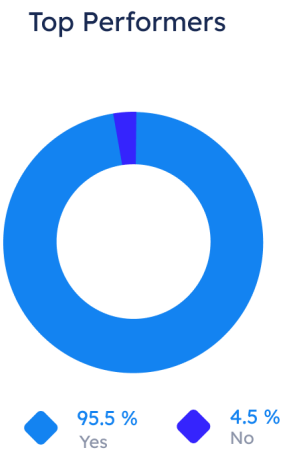


Before we make our way into performance metrics let's look at the methodology of storing the configuration of infrastructure as well as application configuration in code.

Let's start by looking at application configuration. Answering with "yes" means teams describe their configuration using something like YAML as a format. "No" means that configs are hard-coded into the code or applied on a change by change basis. This approach makes it hard to audit, rollback, maintain and govern.

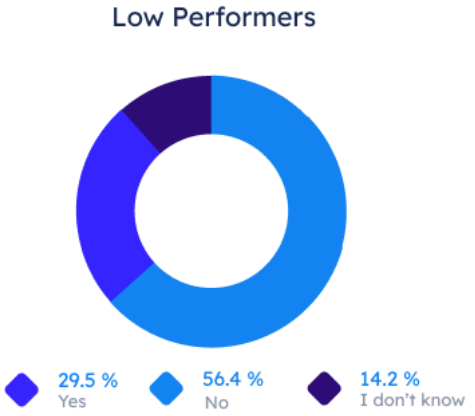
For top performers, configuration as code is completely normal (95.5%), while low performers only store their application configuration as code in repositories in 60.2% of cases (and don't actually know it in 10.1% vs 0% for top performers). This is significant because that 40% that don't store config as code will see skyrocketing amounts of work trying to roll back, follow good governance, meet audit requirements, etc.

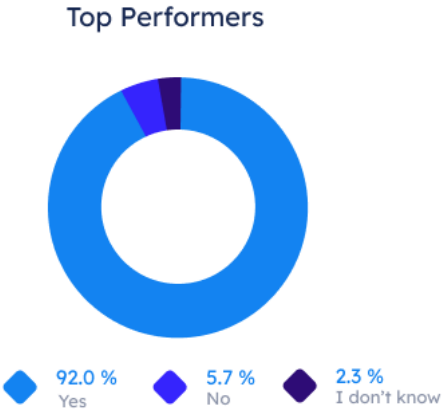




The picture is worse if you ask whether teams store infrastructure configurations as code in a version control system. Around 60% do so across cohorts, 34% don't. If we cluster we find that 92% of top performing teams do this vs 56.4% of low performers. Introducing an approach like Infrastructure as Code to manage your setup can be a large initial investment but tends to yield results quickly: time savings for new infrastructure components, maintenance on existing infrastructure, and improvement in security best practices, just to name a few.

Example: Do you store your infrastructure configurations in a version control system?



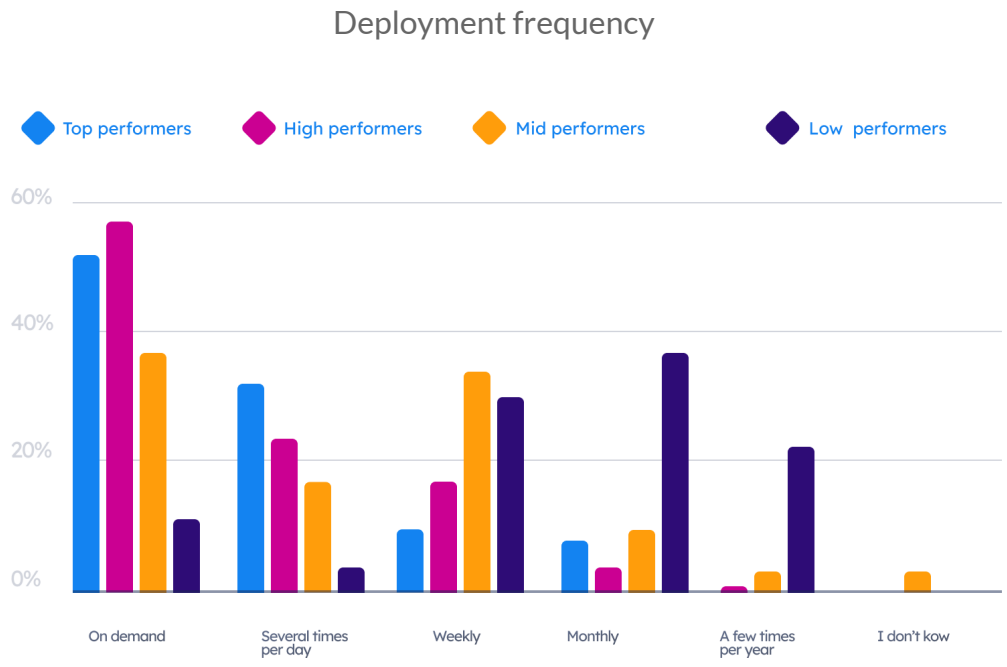


Performance metrics

We looked at the impact that all the above choices can have on "classic" DevOps metrics: Deployment Frequency, Lead Time, Mean Time to Recovery (MTTR), and Change Failure Rate (popularized by the book Accelerate, by Nicole Forsgren, Jez Humble, and Gene Kim).

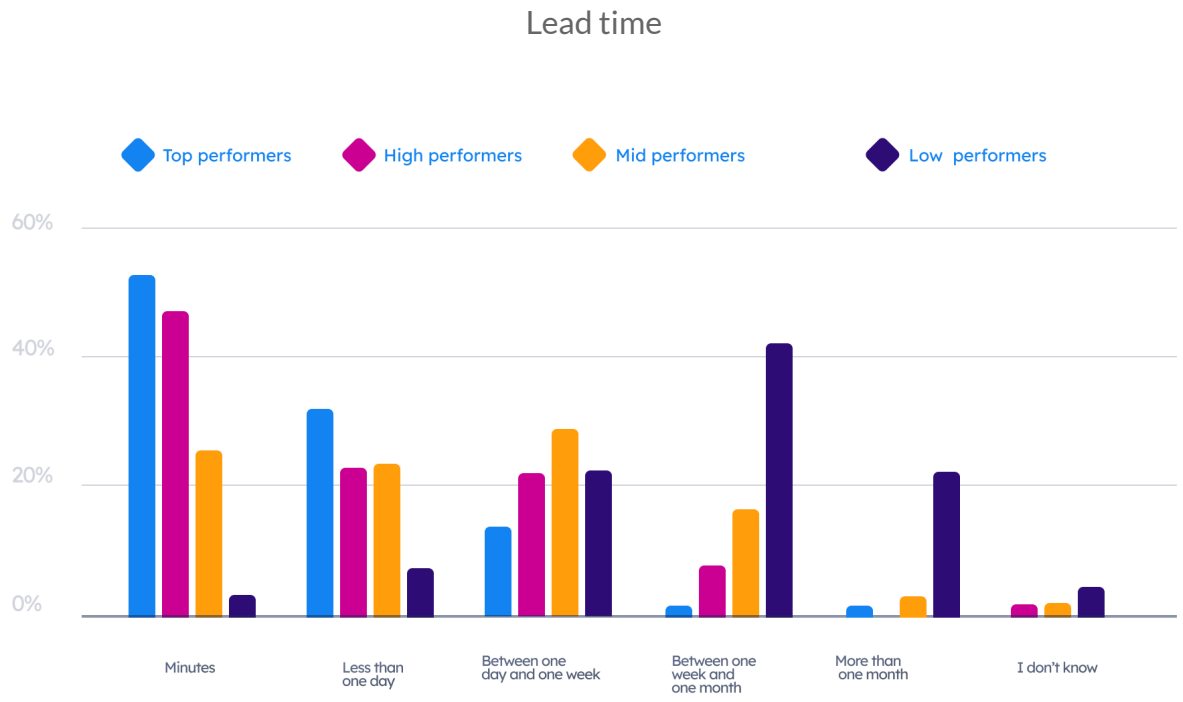
Starting with **Deployment Frequency**, or how often a team is deploying: this metric can tell us a lot about the state of the deployment setup. If it is too low, deploying to production on demand is basically impossible. If your environment management isn't dynamic, you're on a monolith, or your team composition and process flow is buggy, deploying and shipping software fast gets very hard. Deployment frequency also tells a tale of how comfortable developers really are to deploy to production. If your QA setup is not good, you are less likely to hit deploy because this might impact your Change Failure Rate. So teams usually have a tendency to hold back and not deploy frequently.

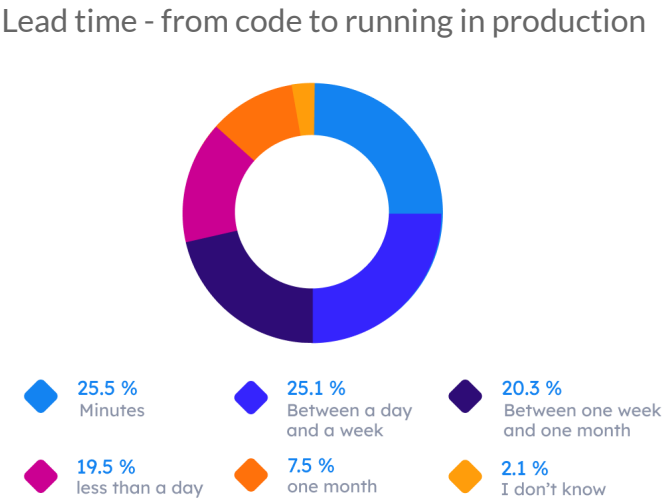
This is well documented in the results. High and top performers enable their developers to deploy to production in more than 50% of cases. Over 80% of top performers deploy at least several times per day and less than 10% of them deploy less than weekly. A stunning 22% of low performers say they deploy only "a few times per year" and less than 10% of them can deploy on-demand.



The picture is quite similar when looking at **Lead Time**, i.e. the time it takes to implement, test, and deliver code. For more than 20% of low performers, it takes longer than one month to deliver their code through all stages. It takes minutes for over 50% of top performers and there are almost no high or top performers that take more than a week. Think about the compounding effect if you are 100

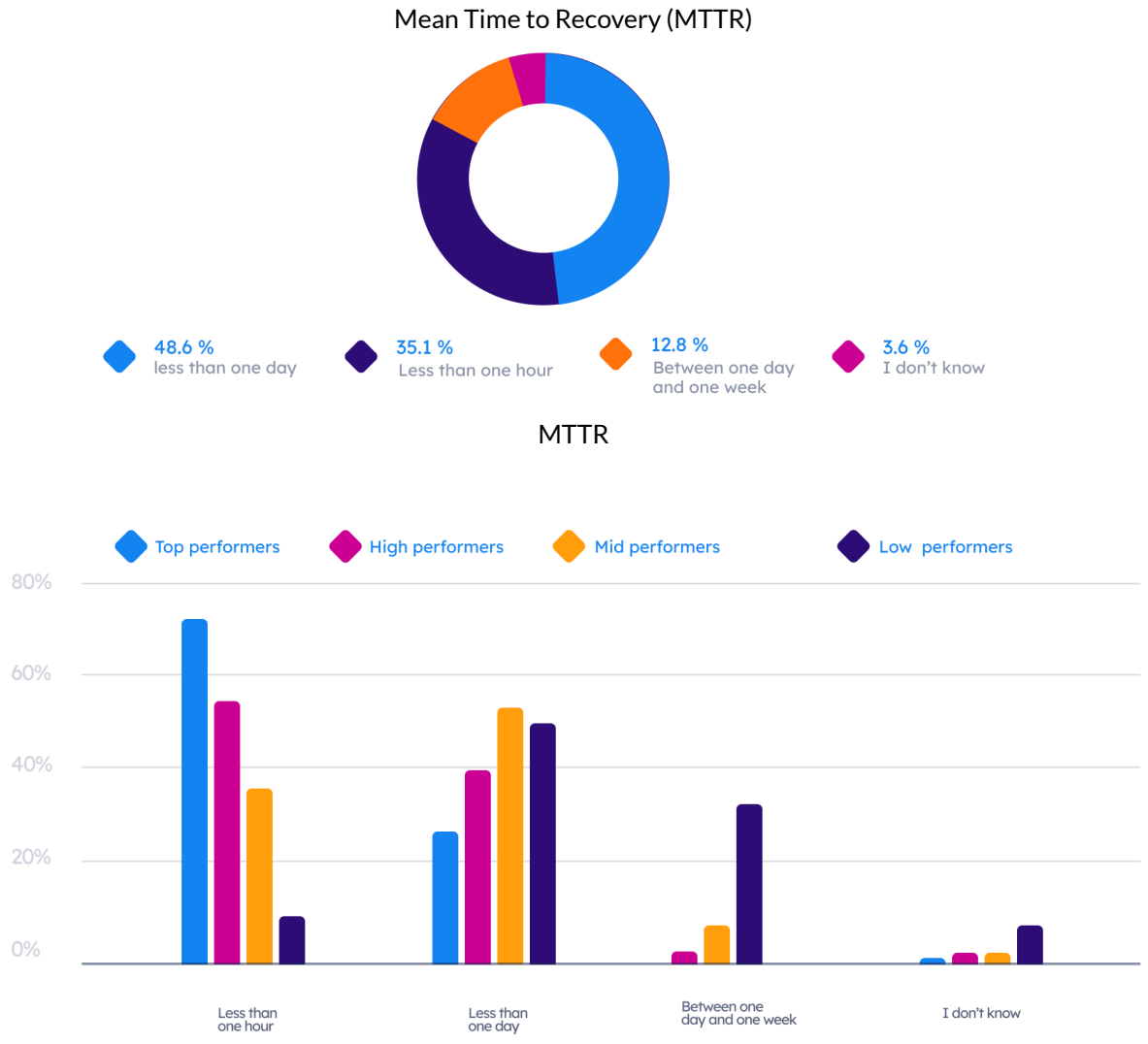
times faster in every single delivery. For top performers, individual code changes applied with any given deployment are likely to be significantly smaller, which makes them easier to review for colleagues and in turn lowers Change Failure Rate.





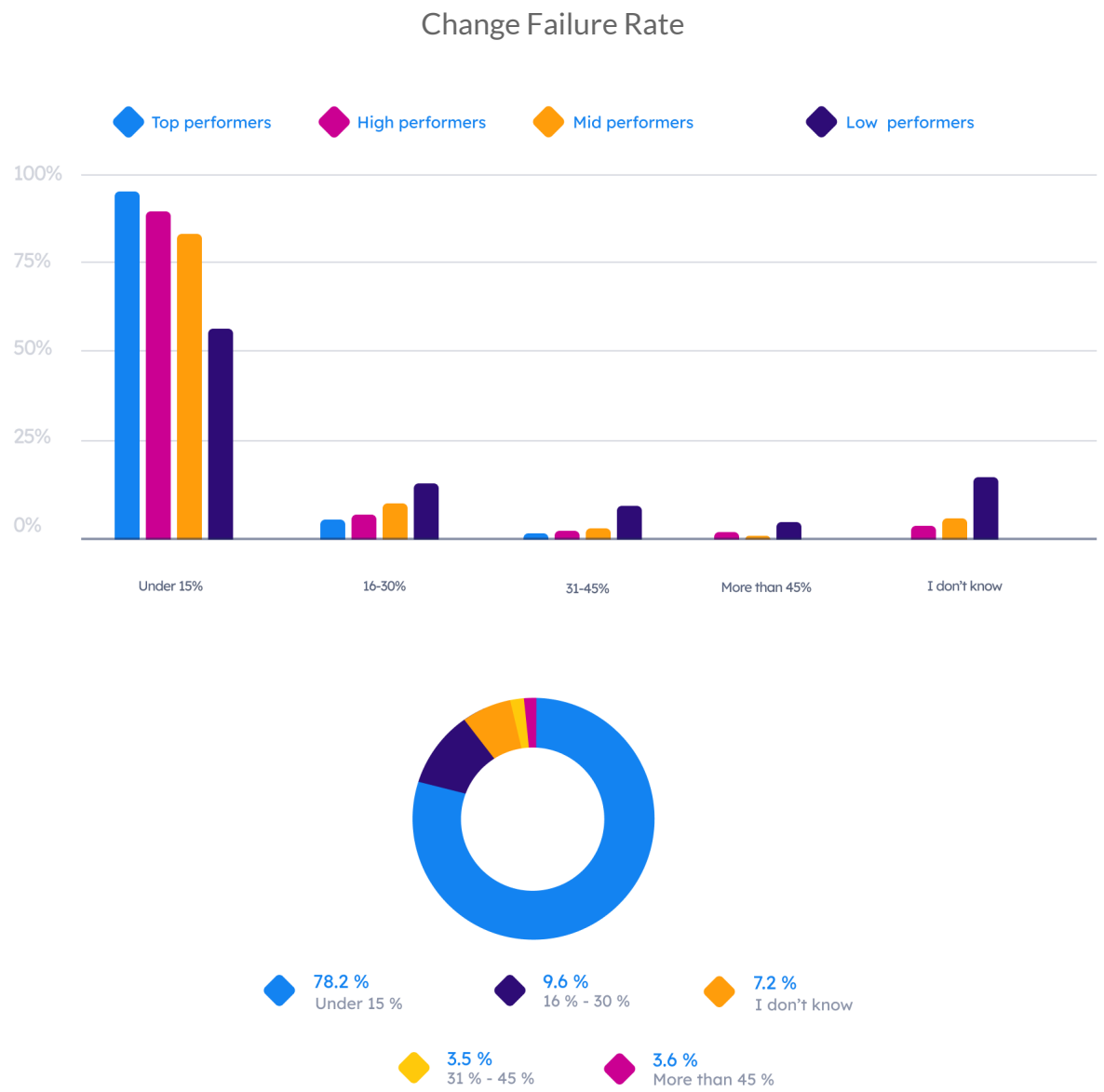
Next up is Mean Time to Recovery, or MTTR. Imagine an outage from a product or system failure. How long does it take you to get everything fully operational again? This metric is a great example of how investments in a good setup immediately impact KPIs. If a team doesn't have IaC or Config as Code (the case for 30% of low performers), it just takes a lot of time fixing things if they go sideways. Even the average across cohorts looks brutal: 12.8% need between one day and one week to restore a system. For

low performers, this number increases to above 30%. Quite instructive is the recent case of a Humanitec's client that was clearly in the low-performing cohort. During the global Covid Pandemic, they settled 1 billion Euros in transactions on their online store. That's a whopping 2.7 million euros a day. After the OVH disaster in a data center in which they ran their core applications, it took them 4.5 days to restore normal operations. That's the damage of 12.32 million euros. For that, you can hire Thoughtworks for 7,252 days in a row. That's enough time for them to write an entire Terraform library from the ground up, ten times.



The picture is similarly striking if you look at the Change Failure Rate. This rate signals how much "faulty" code makes its way to production with any given deployment. Again, the picture for low performers is not great. 21% have to roll back in more than 15% of all deployments. If you think about the associated cost that is involved in revisiting these things over and over again, you can easily see the

impact on the bottom line.



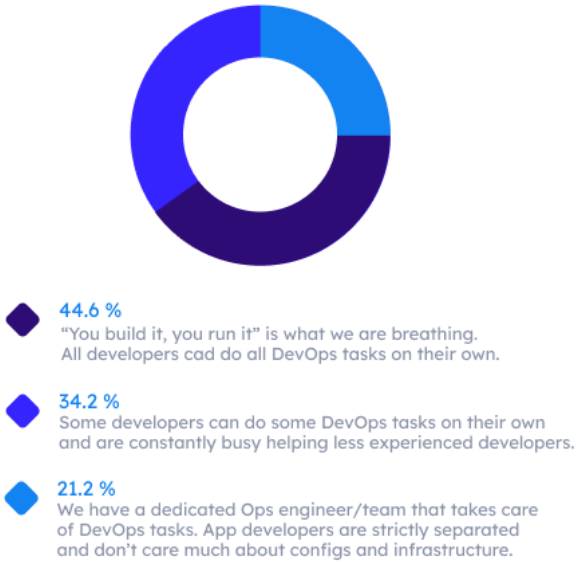
The metrics analyzed across the four cohorts shed further light on the differences in performance. Low-performing teams ship slower, ship more faulty code, take significantly longer to recover from incidents. Yet these numbers in isolation tell us nothing that one could use to improve their setup. They are a pure measure of how good or bad the situation is. As discussed previously, you cannot simply "tell" people to increase deployment frequency. This is a function of the setup and, probably more importantly, of culture.

Team setup and culture

We have now taken a good look at infrastructures, tooling, and overall metrics. But as Peter Drucker said so well: “culture eats strategy for breakfast”. So let’s look at the cultural approaches the different teams took and benchmark them across cohorts.

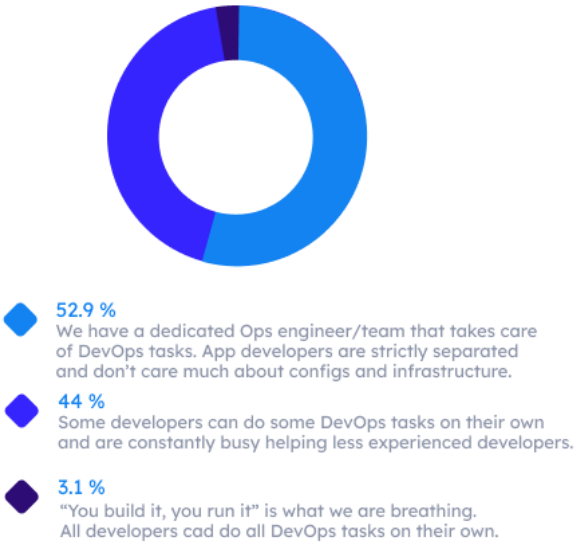
We will start by analyzing the way engineers report how DevOps tasks are managed in their organizations. A brief look at the average results reveals that only 21.2% of teams report that they can do all DevOps tasks on their own. In 44.6% of cases, they are supposed to, but the setup is so complicated that in reality only experienced developers can work on DevOps tasks and become a bottleneck for the team. In a stunning 34.2% of cases, the reality is a “throw over the fence” split like 20 years ago.

Which answer describes best how DevOps tasks are managed in your organization?



Things are more dramatic if you cluster the results by cohort. Almost 100% of all teams in the top performance bucket report that their developers are completely self-serving and operating a “you build it, you run it” approach.

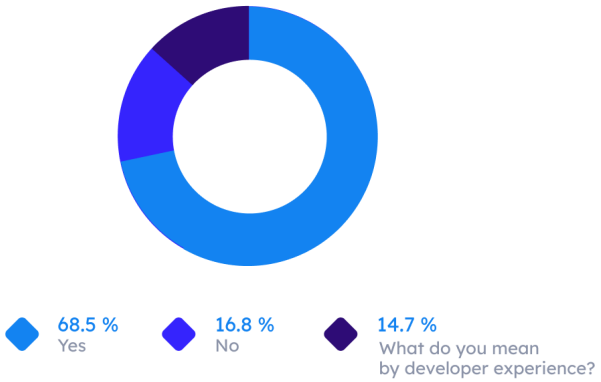
Looking at low performers, the difference is quite dramatic. 52.9% of teams report they throw things over the fence to operations. 3.1% report that their developers run their apps themselves.



"You build it, you run it" is often not possible because of the cognitive load you add to engineering teams if you just provide complex delivery setups and ask them to self-serve deployments. The hybrid response is a great signal of this. The business thinks they are following DevOps, but because the setup is so complex, a "shadow operations" team of senior developers is actually doing the hard tasks for less experienced developers. That's the case for 44.9% of low-performing teams.

If your delivery setup is too complex, you have to invest in developer experience and self-service capabilities, something we've covered in detail in a recent article on self-service. This means in turn, one should see a higher rate of willingness to improve and invest in developer experience for high and top performers, as they are aware of the impact this can have on their overall delivery capabilities. And that's exactly what we found. While high performers are doing so in merely 43% of cases, 96.6% of top performers report to heavily invest to improve their developer experience and consider this a top priority.

We as an organization are constantly striving to improve the developer experience



Crossing the DevOps mountain of tears

So what to make of all this data? How can teams actually use this to get better? Should we start splitting monoliths into microservices and the mountain is conquered? Definitely not. In fact, changing your setup significantly might even make things worse and there is definitely no single right answer to help your team cross the mountain. An argument can be made that the vendor landscape of DevOps tools is more differentiated for public cloud and containerized workloads, but that's probably about it.

Should you increase Deployment Frequency? Metrics are a result of something, not the root cause. Increasing Deployment Frequency artificially will simply increase your Change Failure Rate. Internal penalties for high Change Failure Rates are usually so feared that the Deployment Frequency will drop with some lag and you're back to where you started.

Should you proclaim "you build it, you run it"? You can literally see in the data what happens if you do this. Developers that are overwhelmed and don't have an operations team to bother, will bother their peers. That's the case in 44.6% of teams, on average. So that is not the solution either.

Optimize for cognitive load

The bad news is that it depends. The good news is that what it depends on is cognitive load. In cognitive psychology, cognitive load refers to the used amount of working memory resources.

Let me explain: it's probably fair to assume that developers are somewhere on the right of the bell curve of the IQ distribution. In plain words: they are smart. And of course, they are able to figure out how to run Terraform at scale, if they have to. But while they're becoming IaC wizards, they fall back on their area of specialization. That is not why their managers hired them or what the market pays them for.

Humans tend to underestimate the impact of cognitive load on productivity. In a recent interview, the CTO of Github explained how extremely simplified the delivery setup at the tech giant is.

"We understand these concepts
better than anybody on the planet.
In the end we invented 90%
of all concepts we use today".

And he goes on to explain:

> "You cannot scale an efficient engineering organization on bash-scripts, you'll die."

> "If you run a setup where Dev and Ops talk to each other, you don't have a great setup."

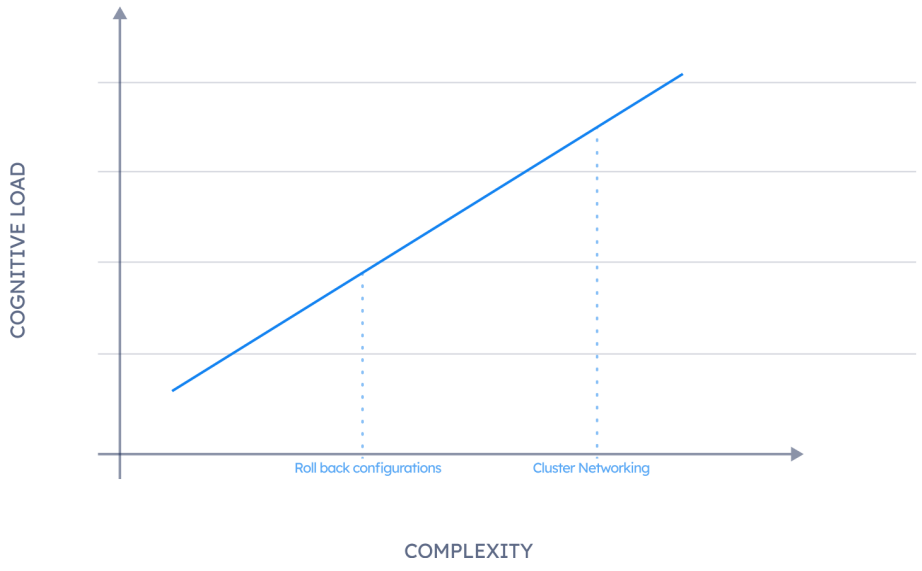
His peers complement this. Aaron Erickson, who build the self-service setup at Salesforce, explains:

> "Service ownership is a good idea in theory, but in practice people get confused. If developers have to run all the ops for their services, you do not have any economies of scale. To run 1,000 different services around Kubernetes, it shouldn't need 1,000 Kubernetes experts to do that."

What all these high-performers acknowledge is that limiting cognitive load matters.

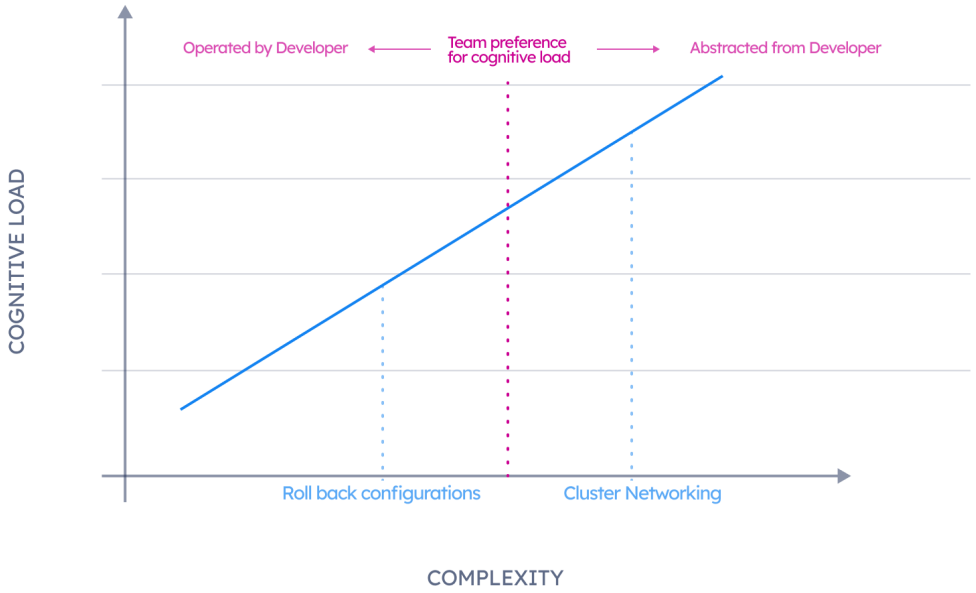
The Cognitive Load tradeoff

And they are correctly determining what amount of cognitive load they can throw at their individual developers. If we look at the graphic below it becomes clear how to approach these things. There is a positive relationship between complexity and cognitive load. More complexity, more cognitive load.



Your developers are usually tasked with delivering business logic. The question is how much cognitive load you can reasonably expect to allocate to them on top of what they have to do already. Should they have to deal with applying changes to application configurations? Should they need to worry about

cluster networking? In short, it's about determining the teams' preference for cognitive load and then limiting load without restricting their daily work.



Everything left of this hand-over point is on them to handle and master. Everything to the right "abstracted" from them. Abstracted in this sense either means providing automated self-service capabilities such as Internal Developer Platforms or literally having other teams do it for you.

Striking the balance

Striking the balance between self-service and abstraction isn't trivial. If you're not abstracting enough you end up with too much cognitive load. If you're abstracting too much, developers feel restricted and - worst-case - spend more time trying to circumvent your abstraction.

Over-Communicate

The only way to gain an understanding of the optimal cognitive load for your team is through constant communication. We've tried to get a proxy for optimal setups or optimal cognitive load, but it's literally impossible to isolate a variable that can tell us which approach to choose. It varies hugely between teams and the context of what the team is building. If a platform team doesn't talk to developers to test, iterate and surface this you have no idea where you stand.

Golden Paths over Golden Cages

Once you have a feeling for the right level of cognitive load, you can start designing your self-service setup around this hand-over point. We end up with the next challenge: preventing too much abstraction. This is where the mantra “building Golden Paths over Golden Cages” comes into play.

A golden path is about abstracting without abstracting. Developers should be able to understand what’s under the hood if they want. They should be able to go low-level. You must not keep them from doing so if needed. All you do is to give a guarantee that things are easier if you stick to the Golden Path. What we see in the data is that 97% of developers stick to a Golden Path once established, following the “social contract” that staying on the Golden Path makes the setup more scalable and easier to operate for everybody.

The most commonly used description for Golden Path-style self-service setups are Internal Developer Platforms. An Internal Developer Platform, or IDP, is a self-service layer that allows developers to interact independently with their organization’s delivery setup, enabling them to self-serve environments, deployments, databases, logs, and anything else they need to run their applications.

Internal Developer Platforms are usually structured and governed by the internal platform team. Which is another critical function we want to propose here.

From operations to platform teams

Top performers built in most cases a platform team to set up and maintain their self-service workflows. This team is not tasked with doing transactional tasks for somebody. In fact, a platform team shouldn’t have any transactional conversation with developers, but strictly qualitative conversations, so they can strike the right abstraction balance.

Winning DevOps with a Self-Service setup build by platform teams

The mission

As usual, from the why. When building an Internal Platform team, it is key that you clearly define its purpose and mission. “To build the tools (IDPs) that enable developers to ship scalable applications with high speed, quality and performance” is a good example. Whatever makes the most sense for your organization. Make sure you set this in stone. Along with the mission statement, it is important that you establish early on that the Platform team is not to be seen as some sort of extension of the SRE or Ops teams, but rather as its own product team, serving customers (app developers) within your organization.

This differentiation is crucial not only from an engineering and product point of view but also from a management perspective. In order to get buy-in on different levels, Platform teams cannot afford to be speaking only the technical language. They also need to master the business lingo too. It is critical for the long-term success of any Internal Platform team that it gets seen within the organization as not yet another cost center we are adding to our already expensive engineering balance sheet. Instead, it is a value center for all other app development teams and - eventually - for the end consumer.

Internal balance

Once the mission is clear, you need to strike the right balance. Successful Internal Platform teams manage to put in place strong guardrails and standards for their development teams. Without taking away too much of their autonomy. To have a meaningful impact, Platform teams depend on having standards in their organization. Trying to support every possible programming language, framework, DB, and whatever exotic new tech only results in Platform teams spreading themselves too thin. On the flip side, you don’t want to come across as a patronizing ruler of infrastructure by imposing your standards on every other team. Standards vs. freedom is a complex topic we have covered before. There’s no one-size-fits-all solution here, but you have to be mindful of the challenges of introducing a centralized set of standards in your organization.

Finally, ensure you select the right SREs and DevOps engineers to build out your team. This sounds obvious, but there’s quite some debate around what makes for a good Platform engineer. It is paramount he or she fully appreciates internal tooling as a real product to iterate on, based on the feedback of the end customer: then application developers. Deep technical capabilities like networking skills are key for a Platform engineer. But make sure you don’t only consider technical sophistication when hiring for this position. Also, look for candidates with a multidisciplinary understanding of their role within the

organization. Alongside these core Ops competencies, you'll also need frontend and design roles to build a complete Platform team. Remember, they are building a full-fledged product for your organization.

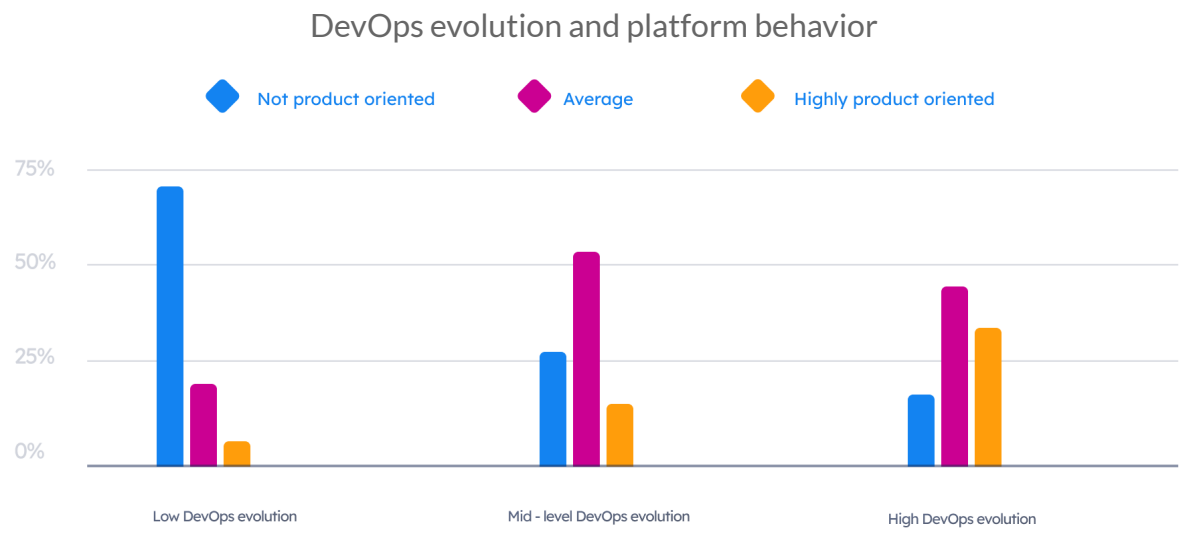
The value(s) of Platform teams

Congratulations, you convinced management to give you the green light and went off to build a world-class Internal Platform team. So how do you make sure it was all worth it? At the end of the day, they are not shipping anything that's end customer-facing. So how do you know they are working on the right things to maximize value for the rest of the engineering org?

We compiled the key areas that we have seen Internal Platform teams focus on to deliver the right internal tooling, standards, and workflows to their application developers. The cornerstone is always the reduction of complexity and cognitive load for the end-user of the Internal Developer Platform. If that is unstable, your engineering productivity will drop dramatically. But let's get into the details.

Treat your platform as a product

We mentioned this already, but it is probably the most important takeaway: your Internal Platform team needs to be driven by a product mindset. It needs to focus on what provides real value (as opposed to what is "cool" to work on) for its internal customers - the app developers - based on the feedback they gave them. This philosophy needs to be engrained into every aspect of your Platform team's activities. Iterate on your Internal Developer Platform, ship new features. At the same time, don't forget you are also responsible for maintaining a reliable and scalable Ops setup. If something goes wrong in your team, all other teams will suffer from it.



Source: 2020 State of DevOps report by Puppet

Optimize iteration speed

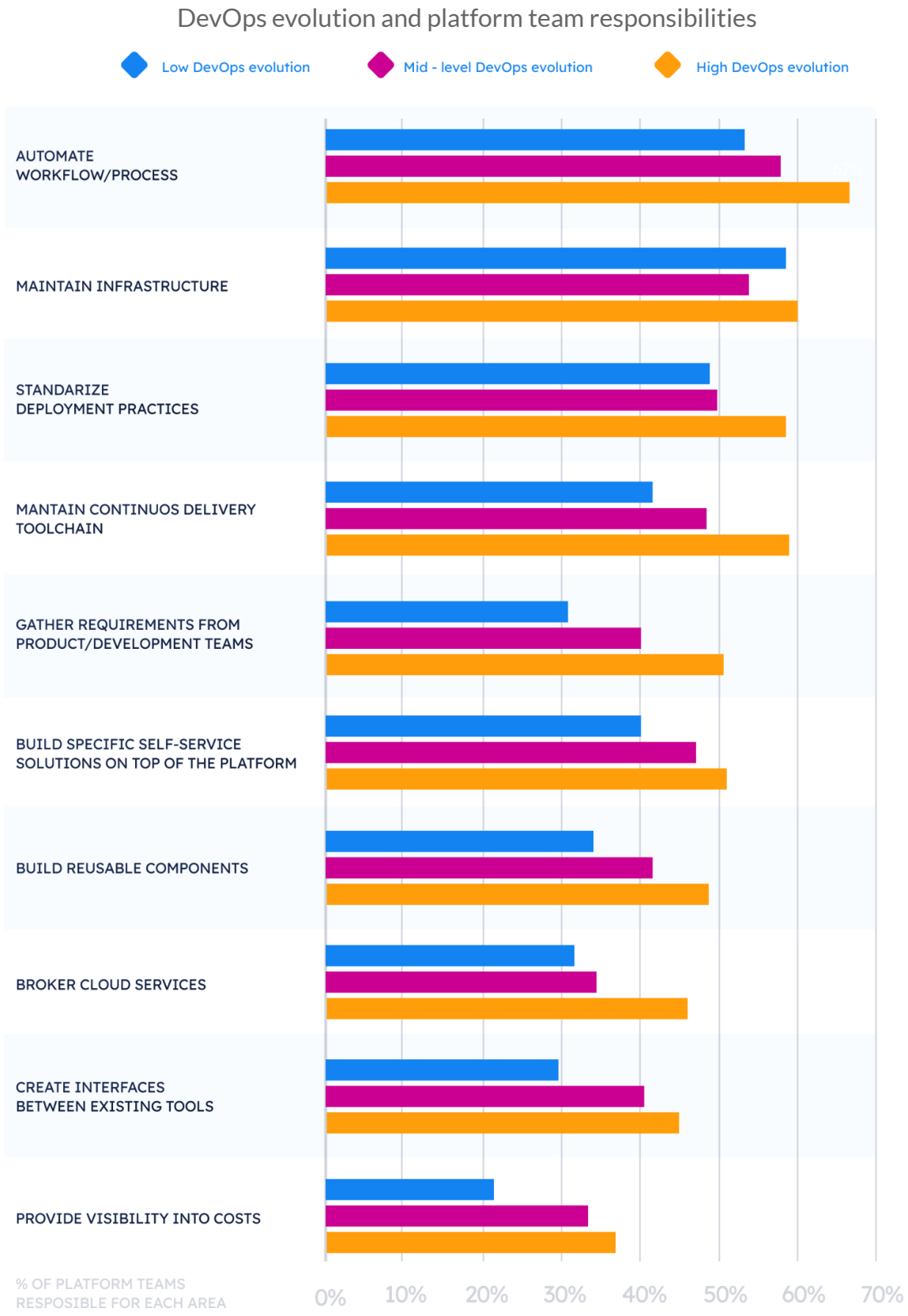
When you think about it, the speed at which your organization innovates is directly correlated to (and constrained by) your iteration speed. Increase that and your app developers will be able to consistently ship more features and products to your customers while being confident that things won't break.

In order to do so, Internal Platform teams need to focus on optimizing every step of the software delivery process. In particular, they should:

- Make local development and testing as fast and painless as possible.
- Critically simplify the way developers interact with their infrastructure (and remove the zoo of scripts from their day-to-day operations).
- Lower the barrier to entry by building the right tools and documentation that enable engineers to onboard faster. You should move away from tribal knowledge as much as you can.

Solve common problems

A good Platform team prevents other teams from reinventing the wheel by solving common problems once. It's key to figure out what these common problems are: start by understanding developer pain points and friction areas that cause slowdowns in development. You can gather this information both, qualitatively through developers' feedback and quantitatively via engineering KPIs. This intel, combined with an understanding of the future direction of the product, can help the Internal Platform team to shape a good roadmap and pick the right battles to fight. The Puppet report gives us some insights into what the core responsibilities of Platform teams are.



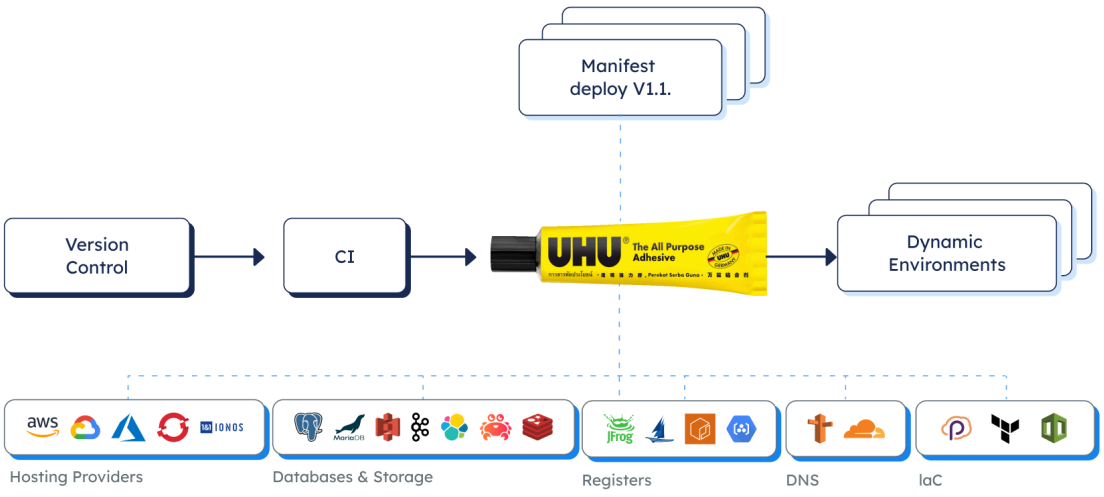
Source: [2020 State of DevOps report by Puppet]

Be glue, my friend

Platform teams need to define a golden path for their developers: a reduced set of sane, proven choices of tools that get the job done and allow you to build, deploy, and operate your services. Once you have defined this path, the main value you create as an Internal Platform team is to be the sticky glue that brings all the tools together and ensures a smooth development and deployment experience for your engineers.

It's important you don't start to fight commercial vendors. It doesn't matter if your homegrown CI/CD solution is superior today. AWS, GCP, Humanitec, etc. will catch up faster than expected and make your tool and team redundant. Every Platform team should be asking themselves: what is our differentiator? Instead of building in-house alternatives to an existing CI system, a CD tool, or a metrics platform and compete against businesses that have 20 or 50 times your capacity, focus on the specific needs of your teams and tailor off-the-shelf solutions to your requirements. Commercial competitors are more likely to optimize for more generic needs of the industry anyway.

Glue for Gold



Educate and empower your teams

Finally, a good Platform team is a central source of education and best practices for the rest of the company. Some of the most impactful activities we saw elite teams routinely perform in this area include:

- Fostering regular architectural design reviews for new functional projects and proposing common ways of development across dev teams.
- Sharing knowledge, experiences and collectively defining best practices and architectural guidelines.
- Ensuring engineers have the right tools in place to validate and check for common pitfalls like code smells, bugs, and performance degradations.
- Organizing internal hackathons so dev teams can surface their requirements for internal tooling needs. Nigel Simons (Director Enterprise Tech at a Fortune 100) explained in a conversation with us that 50% of their teams' hackathon ideas actually make it to production.

Closing statement

Let's give the data a final say on whether developer self-service and experience matter. 96.6% of top performers report that developer experience and self-service are their number 1 priority in improving DevOps. Less than 45% of low performers report this. What this clearly reveals is that the starting point for getting better is acknowledging the cognitive load trade-off and investing heavily in developer self-service, platforms, and experience. The data is crystal clear.

About the Data

Participants in total: 1,856

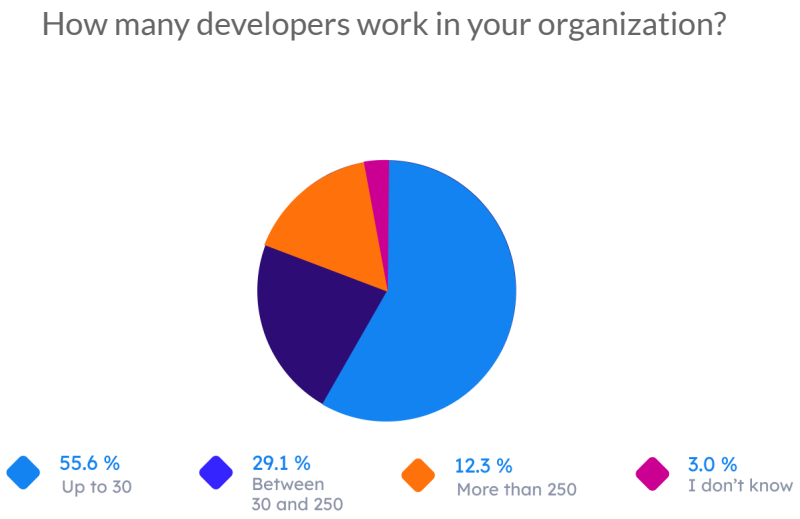
Trying to be fair statisticians, we always ask the question of significance and bias. Of course, this data is not a random set of engineering teams selected without any bias. These are teams that signed up on Humanitec.com, interested in building an Internal Developer Platform. This intention is already an indication of a certain sophistication and openness to innovate.

We decided not to do N/A replacements to approximate results. Instead, we deleted data with vital entries missing, such as the number of developers. We correlated other attributes of N/A data points to

ensure deleting these data points wouldn't skew the final analysis. We ended up with a total of 1,856 engineering organizations.

So yes, the data is biased to a certain extent. But it's still a good representation of the roughly 20,000 teams that match our above-mentioned criteria. We are happy to share the data set (anonymized) upon request.

This data reflects the usual distribution of teams that start thinking about Internal Developer Platforms. Jason Warner explains when to look at Internal Developer Platforms in his interview with us.



Literature

Results of Humanitec's DevOps study show a high coherence with the results of other studies.

Example: Puppet's State of DevOps Report 2020 and 2021.

Puppet State of DevOps Report 2021

Puppet State of DevOps Report 2020

You build it, you run it = developer self-service