RED BADGER

TechLab_Report19

Our view on what matters in tech

Get set for what's next
Here's what you need to know

1
Sh
Stu Harris

Stu Harris
Chief Scientist
and Founder

Once a year our software and test engineers come together to share what matters in tech and how we apply it for our clients. The result is this Red Badger TechLab_Report19 which aims to help you confidently get set for what's next.

This report is inspired by our commitment to making things better. Everything you'll read here comes directly from our people and was written based on first-hand experience. We hope our words will point you towards something useful that will change your (work) life and help you drive meaningful change in your organisation.

As we all get stuck into 2019, it's worth taking a moment to reflect on last year. History will mark 2018 as the beginning of the mass adoption of Kubernetes container orchestration. Why is this significant? Kubernetes is a game changer. It's the product of an immense collaboration between developers and operations (DevOps). It embodies all the principles we talk about in this report, enabling us to move faster than we ever did. As a result, we can now measure the time it takes for a software update to get from a developer's hands to a user's hands in seconds – even at web-scale. This is True Agile.

So, what will 2019 be known for? Service mesh. When we add service mesh (e.g. Istio) into the equation we have a new standard platform for distributed microservice applications – one that's taking over the world. Imagine this platform as a homogeneous substrate, onto which you can declaratively deploy any containerised workload. It looks exactly the same everywhere: in any cloud, on-premise, or hybrid. It will take care of all your cross-cutting concerns and non-functional requirements. Think scaling, healing, security, resilience, reliability, and observability, to name a few. Most importantly, it allows you to concentrate on the value you want to add. It takes care of your tricky, sticky tech challenges, freeing you up to focus on your core business logic and, most importantly, your customers.

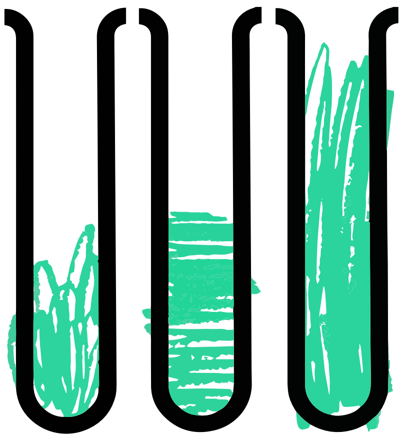
Change is a constant, and its pace is accelerating. But when you're equipped with the right knowledge, insights, and solutions, you're set up for success and can enjoy the ride.

Make change happen
Want to talk about exactly what this thinking means for you? Get in touch.

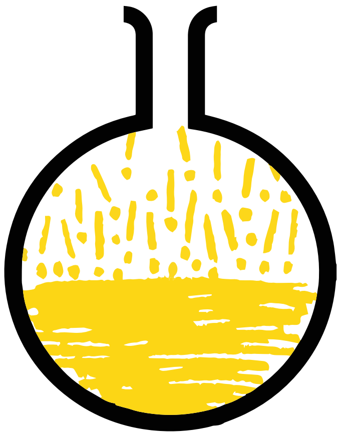
hello@red-badger.com

A quick overview
The report explores these themes

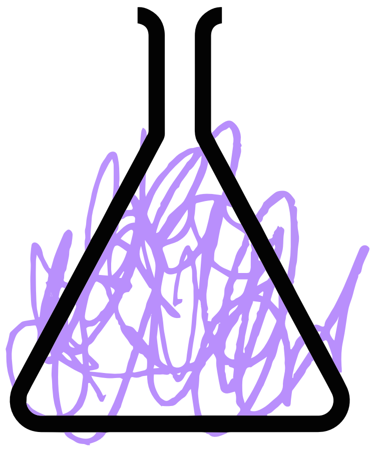
Common problem areas
Read about problems that many of our clients come to us with, along with our insights on how to set things right.



Tools and technology
We're always trying out new tools and technology, so we can help our clients make the right choices for their businesses. Explore a few that we are particularly excited about, see what makes them interesting, and why we think they might be useful in solving client problems.



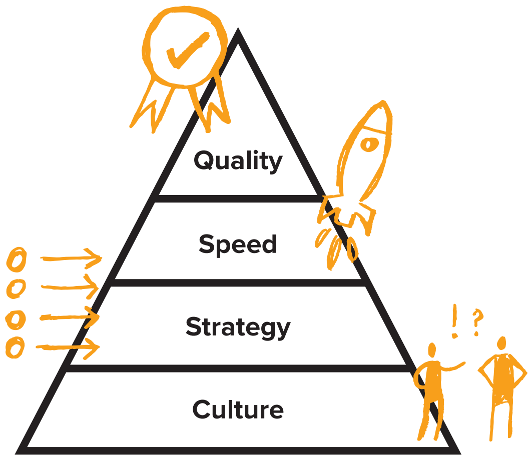
Our specialisms
Our core specialisms have evolved out of the needs we've seen in countless businesses over the years. Dive in to find out how we can use our core strengths to devise solutions tailored to your organisation.



Our engineering principles
Explore the guiding principles that underpin all our engineering decisions and discover how these help us deliver insight, direction and support to our clients and their customers.

Common problem areas
Challenges you may be experiencing

Time and again, we witness similar issues cropping up in clients' businesses, from building software right through to testing, deployment, infrastructure, scaling and reliability. Here are some typical problems our clients come to us with, along with our insights on how to set things right.



Delivery speed and agility

Effective, timely software delivery requires close collaboration between product, delivery, and technology. In order to remain competitive, reduce waste, and respond to customers' needs promptly, short lead times – the duration a piece of work takes from ideation to being live – are essential.

In software delivery, fast iteration is the most important enabler and, by using a combination of techniques, lead times can be improved from several months to just a few hours – without sacrificing reliability or quality. These things will help:

Implement data-driven product management
Data-driven product management forms a product backlog based on user research and analysis of the ways users interact with software. Rapid prototyping and mechanisms for the small-scale release of features and A/B testing allow you to validate ideas quickly and in a low-risk, cost-efficient way.

Use agile delivery processes
Robust, agile delivery process and practices enable the creation of autonomous, empowered cross-functional teams. These teams have all the tools necessary to succeed. They're also data-driven, striving to continuously iterate and improve their ability to deliver high-quality software.

Make continuous delivery happen

Technology enables the continuous delivery of value to customers. Techniques such as feature flags, canary deployments, A/B testing and close monitoring of the systems let you gradually release new features with minimal risk. Building analytics into products from the beginning allows for powerful insight generation, to help inform the future roadmap of the product.

High velocity in software delivery requires high levels of automation. Human gates should be replaced with automated checks wherever possible. Code deployments and health-checking should be fully automated, including the ability to roll back if things aren't looking good. Approaches such as this can help make continuous deployment to production a reality, providing value to customers as quickly and robustly as possible.

Good news – speed of delivery has an added benefit. You'll soon notice everyone involved in building the software relishes the fast feedback and the autonomy required to perform at that level. Continuous delivery has been shown to correlate with more open collaborative cultures. And – even more good news – scaling an organisation composed from autonomous self-sufficient teams is much easier.

Reliability and scalability of service

There's nothing more disappointing than an outage of an application serving all your customers. These systems take significant time and investment to deliver and they typically fail at the time they're most needed – when most people try to use them. Outages result in lost revenue and, often, bad press too.

Scale – elastically

A web or mobile service's demands on computing resources grow with the customer demand of the service. Sometimes the capacity needed at the peak of demand is thousands of times higher than in the middle of a quiet night. In order to fulfil it, services need to elastically scale, by adding resources as they're needed. Otherwise, your spare capacity will likely go to waste. And you'll still end up paying for it.

This requires infrastructure automation commonly supported by cloud computing services, but also a system design that can use the added resources effectively – a property known as horizontal scalability. Applications need to be packaged in a portable way and able to start within seconds, when a new, independent instance is needed to serve more customers. These assumptions are the foundation of the current infrastructure tools, such as Docker and Kubernetes.

Look to smart integration layers

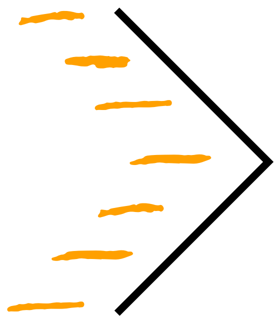
Horizontal scalability is especially challenging for applications which integrate with legacy systems running on physical hardware. These systems can't scale easily. Smart integration layers are often a good way to isolate the effects of them reaching their capacity limits from the customer-facing application.

Unlock the power of continuous delivery
An added benefit of having an application and infrastructure design which scales easily is the ability to create development and testing environments in minutes, enabling the continuous delivery of new features. The risk posed by changes can be reduced by using methods whereby they're released gradually to a small subset of customers first, before dialling the number up. And if a change does cause an outage, full automation makes it much easier to quickly restore the service.

Manage risk
Special consideration is needed for monitoring web scale production systems. Real-time performance metrics, observability, and well-tuned alarms are a must. Why? Because every minute of outage can cost thousands of pounds. Think e-commerce systems going down in the weeks coming up to Christmas, or flight operations systems outages causing chaos.

Empower your teams
Scalable and reliable systems are far more complex than many people first assume. Teams should be given the time and autonomy to learn to operate them safely. Some incidents in the early stages are inevitable, but they should be used as lessons to improve the system.

It's also important to point out that while cloud computing is a great enabler of scalable and reliable systems, many expect it will also reduce their like-for-like infrastructure costs. Be warned – this is rarely the case. But the spend is well worth it to prevent expensive disasters.



Strategic direction

At the end of the day, technology is a means to an end. The goal is to deliver value to customers. The tech should be subservient to this goal. And it should be invisible. Too many organisations let tech choices drive a solution when it ought to be the other way around.

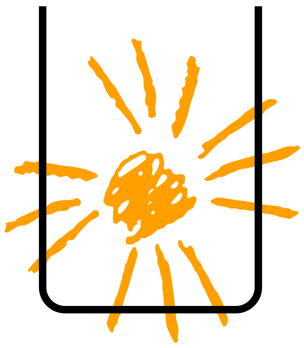
Forgetting the customer comes at a cost
Technology departments spend years – and millions – on strategic shared solutions for infrastructure, deployment and monitoring tools. These efforts are so complex they often never deliver and product delivery teams are left with perpetually tactical solutions which barely work, slow down product delivery, and cause outages. To make matters worse, technology initiatives are run as projects and often completely ignore even internal customers, let alone the end customers, treating them as subjects of governance.

Make it about products, not projects
Instead of being project focussed and building from the back to the front, we should be product focussed and build from the front to the back. Projects start and end, then the money goes somewhere else. But products live on, especially if it's all about the product and we work backwards from a true understanding of customers' needs. With this way of working, we only build what we need to support those needs, keeping things as simple and agile as possible.

Stay curious
To win at the product-first approach, we have to keep up to date with the technology choices available to us. We have a problem to solve and we want the widest choice we can get, so we can select the most appropriate tech to serve us. This means understanding and navigating the open source revolution that continues to change our world. It means trying out new tech, new ideas, and new patterns, and being bold. It's ok to fail early, cheaply, and to abandon experiments that aren't going to work. That's where the learning happens.

Like taking a thousand photos in the hope of getting one good one, innovation is born from experiments bravely conducted by autonomous teams with a deep sense of where they're going. And where the industry is going. These kinds of teams pick up trends early, steal a march on competition, and buy themselves precious time to spend on evolving value. With awareness of open source trends, making the right strategic bet when technologies mature (think React Native or Kubernetes this year) is easy. Just rely on the evidence from your in-house experiments.

An added benefit? A culture of experimentation makes it significantly easier to hire brilliant, ever-curious engineers.



Engineering culture

Engineering culture problems aren't always easy to identify, but there are some telltale signs we've noticed over the years of supporting our clients with all-things tech. If you spot these problems in your teams, it's time to take action.

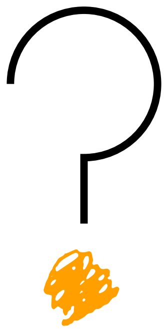
The rest of the business keeps away
If other departments avoid engaging with the technology department, a storm may be brewing. You may notice that, if other departments do engage, the technology department is just perceived as a cost centre, not an opportunity to add value. In these kinds of scenarios, the technology department is rarely invited to help with product decisions.

Silos are stifling projects
When it comes to projects, siloed teams separately work on code, testing, infrastructure, and operations. Projects feel like they're thrown over the fence and they take a long time to deliver – if they even get delivered at all. When a project is delivered, end users only see value when the whole project is complete.

New talent isn't interested
It's a constant struggle to hire and retain developers, plain and simple. When you ask candidates why they're not keen to join, they're not forthcoming. There seem to be so many more exciting opportunities for them in the market.

There's no autonomy, or there's full-on anarchy
All too often, developers are seen as code writers to be strictly managed. In these instances, the same technology and approaches are applied to every problem. Deviations from this are quickly stopped and new technologies and tools are rarely introduced. At the other end of the scale lies anarchy. In these kinds of scenarios, nothing gets done. The systems built are complex and riddled with defects. Developers prioritise fixing problems that users will never see, often jeopardising the customer experience in the final product. When something goes wrong, nobody takes ownership or steps up to fix it.

Of course, the sweet spot is somewhere between autonomy and anarchy. Here, you can reap all sorts of benefits and stay immune from extremes.

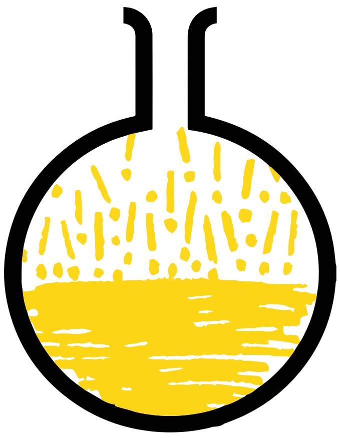


Here's how to solve a culture problem
The most potent remedy we know of is striving for continuous delivery. This enables organisations to reduce the amount of planning, and react to business and customer needs faster. It requires a lot of changes in the organisation, chief among them the move from functional silos to autonomous, cross-functional teams. People on such teams feel more empowered, they have more knowledge of the domain, take more ownership, and ultimately enjoy the work more. Continuous delivery also needs to be supported by flexible infrastructure and good testing strategy with high levels of automation.

A final word – it's worth it. Engineering culture takes a long time to foster. Trusting people and giving them time to learn is essential. But the return on this investment can be immense, enabling you to respond to changing customer and business needs within hours, and outmaneuvre your competitors. All with happy, thriving tech teams.

Our specialisms
Solving your business's biggest problems

Our core specialisms have evolved out of needs we've seen in countless businesses over the years. We know what to focus on, which tools will help, and which principles to be guided by. We harness our deep and wide expertise in these capability areas and devise solutions tailored to your organisation.



Continuous delivery

In the midst of Digital Transformation, speed is the most important metric technology departments should focus on. With the right speed, you can deliver features and fix problems in production systems at a moment's notice, several times a day.

To do that safely, you need to invest in establishing a set of systems and practices that enable the continuous delivery of value, without compromising quality and reliability.

Aim for continuous deployment to production
Continuous integration and continuous deployment (CI/CD) are well-known automation practices which have become widely used to deliver software quickly and safely. Quite often, however, these automaton pipelines only go as far as a testing or 'staging' environment, where code changes and infrastructure changes build until such a time as a decision maker says it's time to push the button to go live.

While this is great for the development process, you'll then most likely hit the same issues you're trying to avoid during development in your deployment into production. Continuous deployment to production essentially takes these practices one step further – all the way to production.

Make small changes, often
Continuous deployment to production means every feature is "done" when it's running in production. The process generally starts when a feature or a part of a feature has been written by (a pair of) engineers, reviewed, and merged into the master branch of a repository. The automation then kicks in to run all the tests and any other automated quality checks, eventually ending with a deployment to the production environment, in front of customers, with no further manual gates.

Making small changes often has been proven to help de-risk software delivery. Small changes made in the morning and deployed in the afternoon allow for any small issues after deployment to be easily tracked down, debugged, and fixed quickly – even if your automated checks didn't catch them (which, over time, becomes pretty rare).

Experience the benefits

The benefits of constantly shipping small changes to production aren't limited to quality and reliability. Product, user experience, visual design and ultimately your customers also benefit from these changes as the time through the deployment loop is the lower bound on how quickly things can change and adapt. With continuous delivery, it can be significantly reduced. This allows a full change in how you go about product delivery, enabling a shift to a continuous build, measure and learn cycle.

Quite simply, continuous delivery (and infrastructure to support it) is the number one strategic capability you need to have.

Infrastructure

Application infrastructure – servers, load balancers, name servers, and more – can be one of the biggest barriers to fast, agile delivery. But it also has the potential to be your biggest enabler, and it's one of the most important areas to focus on getting right.

Infrastructure's come a long way

Setting up infrastructure used to involve screwdrivers and typing commands using a keyboard attached to a serial port in a noisy, cold data centre. The processes were slow, expensive, and very prone to human error. Updates and patches had to be applied manually, to each individual machine, and this effort required dedicated teams. The manual aspect to all this also made it incredibly easy to end up with 'snowflake servers' – servers that are unique, irreplaceable, and a liability.

Innovation has shaken things up

Over the past decade the industry has made incredible progress. Virtual Machines (VMs) and Virtual Networks brought the dawn of cloud computing, which grew to a full offering of Infrastructure as a Service (IaaS). Think VMs, Virtual Networks, block storage, firewalls, load balancers, gateways, and more. The key innovation was that IaaS had an API. We could suddenly write programs to build infrastructure, resulting in the practice known as Infrastructure as Code (IaC).

Tools like Terraform make the process of setting things up fully repeatable and allow us to have development and production environments that are nearly identical. In turn, this helps catch issues earlier in the process and enables various kinds of testing (performance, penetration, etc.) to be done outside the production environment.

Containers and cluster orchestration are changing the game

The next leap in infrastructure came with containers – an Operating System level virtualisation. Container technology (the most popular being Docker) made it possible to package software written in various languages in a uniform, portable way. It also made it possible to deploy multiple containers onto a single VM. This meant better utilisation, and improvements in the time taken to deploy or scale up an application.

The next big leap forwards in the infrastructure space is happening right now. Cluster orchestration software, such as Kubernetes, solves the problems that emerge when you run lots of containers on a cluster of virtual machines – from orchestrating zero downtime deployments to handling machine failures. This creates a reliable, self-healing, easy-to-scale runtime surface. With this in place, application teams can focus more energy on delivering application features and customer value.

Smart infrastructure will serve you
Infrastructure is a key area of strategic investment for organisations. The focus should be on reusable solutions that enable continuous delivery and increase autonomy of teams as much as possible.

The latest generation of infrastructure tools and technologies makes this easier than ever.

Testing and quality assurance

Testing is heading into a new paradigm – one shaped by self-healing systems, observable platforms, and everything as code. Gone are the days when working software was enough of a barometer of quality. Recent years have seen the testing function move from the diagnosis of issues to their prevention. This trend is set to gather pace.

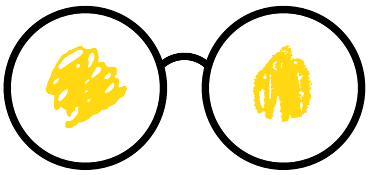
Embrace observability
The rise in applications built with observability in mind allows for a broader perspective on the overall state of an application. In embracing observability, we accept that failures will always occur and direct our efforts towards making unknowns manageable when they arise. The modern tester will play a large part in the incremental improvements of such systems. This will require a shift of focus from more traditional independent components, such as user interfaces or APIs, towards incorporating the monitoring of overall platform health into test strategies.

Think 'everything as code'
The advent of the 'everything as code' approach and the ever-increasing reliance on automation of process, application builds, type checking and test suites have drastically changed the test function. In a setting where a tester can spin up a production-like local test environment in seconds with tools such as Docker and Kubernetes, front-loading test-effort has never been easier.

Dare to test in production
Testing in production was once seen as a risky endeavour, but the increased adoption of feature flags has mitigated that risk. As a final quality gate, there's no substitute for testing in the production environment, under real-world conditions. Feature flags are also an excellent enabler for A/B testing with real users who can provide invaluable insights you won't always get from traditional testing.

Try out new tools
Finally, new tools such as Puppeteer and Cypress have the potential to overhaul UI testing for the better. Focussing on speed and reliability, automation can act as a conduit to continuous deployment and continuous delivery – like never before. For too long, UI automation has been a pain point to so many, as teams struggle with reliability and execution times.

If these tools live up to their promise, these struggles may become a thing of the past.



Observability

We all want reliable production systems that can recover from problems fast. Observability is key to making it happen. When your systems are instrumented with monitoring, logging, and alerting tools, humans can see what's going on, get notified if things go wrong, and take action.

Know the three pillars of observability
Monitoring is the tracking of key metrics and data. Logging is collecting the details of the occurrence of events that are happening in your system. And DevOps teams set up alerting so they're notified when something dodgy is happening and can jump in and investigate the potential issue. Observability may seem subtle, but it impacts how your instrumentation is designed and can help guide your choice of tools.

Speed up the diagnosis of new problems
Observability is about focusing the design of your instrumentation on speeding up the diagnosis of new problems, instead of monitoring for things you've seen before. DevOps teams often fall into the trap of creating multiple dashboards that only show things that have already been instrumented.

You've probably seen a dashboard that shows memory usage and lists of errors, as well as performance data. This helps as a general overview of the health of your service, but during a call out or investigation, engineers spend time browsing these dashboards for anomalies and then potentially go on to investigate by diving into logs. The result? Valuable minutes are lost. And, in order to create such a dashboard, specific instrumentation has been built into your code and infrastructure.

Choose tools that bring you built-in visibility
A design more focussed on observability would instead choose tools that provide more built-in visibility. The incident response workflow should go from receiving a callout to the triggering event and then immediately to searching and correlating event logs from related systems relevant to the source of the call-out. The workflow focuses more on asking questions relevant to the situation than browsing questions you asked before.

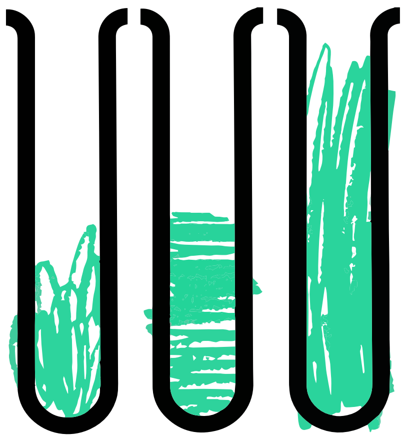
To quickly diagnose issues, you need to be able to see interactions between processes and services easily. Service meshes are your friend here. And you need to store logs so interactive searches can be super-fast with tools like Prometheus.

The takeaway? Designing for observability, rather than monitoring, helps reduce time to recovery during an incident. And it focuses instrumentation on finding the cause of a problem and fixing it.

Tools and technology

Supercharging business transformation

We're always trying out new tools and technology, so we can help our clients make the right choices for their businesses. Through this process, we identify the best solutions out there and gain a deep understanding of where and how to use them. Here are some of our favourites, along with what makes them interesting, and why we like them.



Kubernetes

Jan Kuehle, Software Engineer

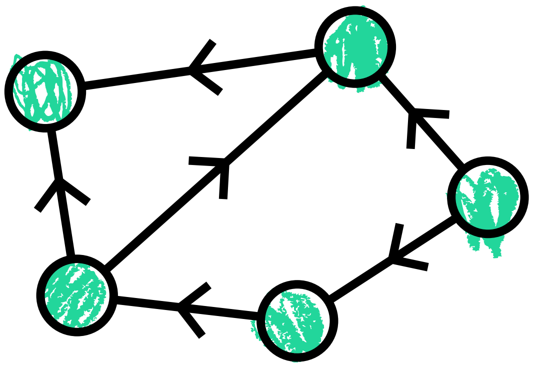
Traditionally, highly available, resilient applications would have to be deployed to a large number of servers or virtual machines. Having one application per virtual machine makes it hard to make the most of your resources. And creating and deleting virtual machines can take a long time. Containers (such as Docker) are designed to solve these issues. They start and stop quickly and improve your resource utilisation because you can run many containers on a single machine.

Kubernetes was created to make it easy to manage containerised applications across many – potentially distributed – machines. Some of the high-level features you get with Kubernetes are: automated, zero-downtime deployments; horizontal and vertical autoscaling; load balancing across multiple containers on multiple servers; and container-aware service discovery through DNS.

All of these work with a declarative API. Users define the desired state of the application (say, two instances with load balancing) in their preferred configuration language (JSON or YAML). Kubernetes performs the necessary steps to bring the application into the desired state. This enables a self-healing system. For example, when one instance of the application dies, Kubernetes will automatically start a new instance to maintain the desired state.

The configuration can be stored in files and added to source control repositories next to the application's source code. This neatly ties in with the infrastructure as code and immutable infrastructure patterns.

Kubernetes is clearly winning as the new standard of cluster orchestration. Several different providers offer cloud-hosted or packaged Kubernetes distributions, making it an obvious strategic choice of an application platform. That said, Kubernetes is a complex toolset and, if you're looking to adopt it within your organisation, you'll need a strategy. In our opinion, Kubernetes makes it possible for small teams to fully own their infrastructure, for the first time. We've given that strategy a name – microplatforms.



Service mesh

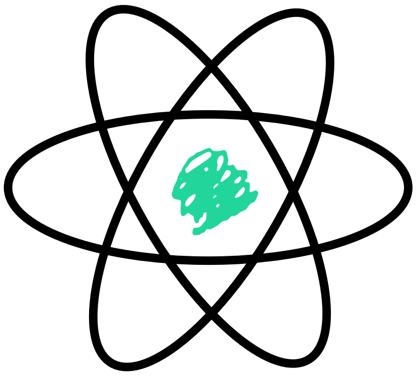
Tornike Keburia, Software Engineer
The move to cloud-native applications and microservices architecture brings many well known benefits, but also creates some new challenges. One of the trickier problems is managing the communication between multiple services in a distributed system.

However small and cohesive the microservices are, there's always a set of common problems to solve in a distributed world, such as routing traffic to the right service, handling timeouts, balancing load across instances, enforcing security measures (authentication, authorisation, encryption), and making all the communication observable. These quickly become very complex to manage as more services are added to your application, especially if the services are written in different languages, forcing you to repeatedly implement the same solution for different technology stacks.

This is where service mesh lends a helping hand. In a nutshell, the service mesh is another layer of infrastructure handling all the common networking functionality. Service meshes don't really introduce any new functionality – they just move the logic that would have to be repeatedly built as part of every microservice to a separate, centrally-managed logical layer. Physically, service mesh is a set of side-car proxies attached to each service. Traffic to and from services is forced through its respective proxy which applies various policies as decided by the global configuration.

Introducing a service mesh into your infrastructure will reduce complexity and time spent on managing system-wide policies and complex deployments. And your day-to-day operations become easier thanks to the centralised way of viewing, troubleshooting and maintaining your service communication layer.

While service mesh is a relatively new concept, it's built on a foundation of tried and tested, reliable technologies. The leading one – Istio – is backed by industry leaders such as Google and easily integrates with commonly used technologies like Kubernetes. It's actively supported and new functionality and integrations are constantly being added. At Red Badger, we believe service mesh will soon become a standard part of any infrastructure solution.



React Native

Matt Paul, Software Engineer
2018 was the year React Native first became a viable option for many organisations – the year when the question changed from ‘Why?’ to ‘Why not?’.

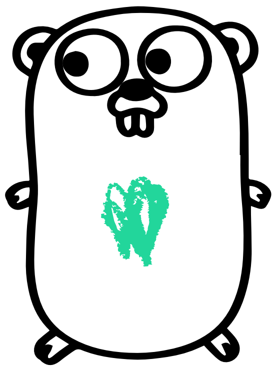
At first, React Native was ‘simply’ a platform for building native-feeling applications with native UI widgets, but in JavaScript and using React – a widely adopted, declarative UI library that many web developers have become big fans of in recent years. Over time, the number of target platforms has grown. In addition to iOS and Android, we can now target Apple TV, VR, and even Windows applications, Xbox, and – interestingly – web. The web version of React Native is an entirely different beast from the original web version of React and is much closer to React Native in the building blocks it uses, which makes components portable across the React Native target platforms.

For many organisations, apart from managing increased expectations from consumers, increasing the number of platforms can result in increased adoption due to the wider audience now able to access the application. The ability to target multiple platforms has made React Native an ever-more compelling strategic choice for user interface.

Combine this with the sharing of component libraries across teams utilising other tools in the React ecosystem, such as Storybook, and the era of platform agnostic UI development and design systems is here. On a recent client project, we saw around 80% code reuse whilst targeting three platforms – iOS, Android, and web with React Native Web. After an initial setup period, speed of delivery increased significantly.

Fundamental to its success, React Native has a vibrant community of contributors and companies moving it forward, including tech giants such as Facebook and Microsoft. Given this plus the fact that the underlying technology is React – a battle-hardened and widely-used library – stakeholders can feel warm inside knowing that there’s a considerable ongoing investment, reducing the risk of a technological dead end.

While React Native is a strong proposition, there are a number of considerations you must take into account, especially if you have existing native development teams in place. Given the unfortunate tribal nature of software development, it’s important to take care to integrate these groups without friction. You can be sure that a few toes will be trodden on until the dynamic is established. The experience of Airbnb, which stopped using React Native, is an interesting case study showing it’s not necessarily a solution for everybody.



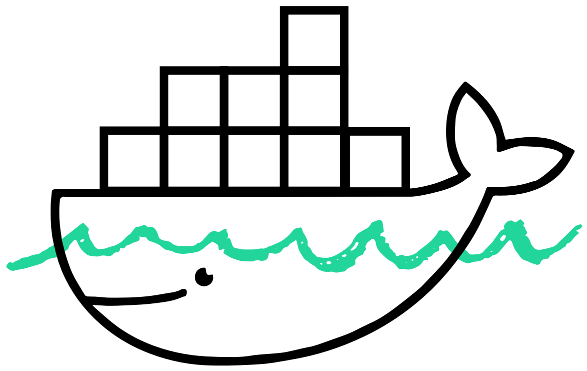
Golang (Go)

Sam Taylor, Tech Director
We're big fans of functional languages but we can appreciate a very well designed imperative language when we see one. Golang (Go) is a great example. It's statically typed. The huge benefit? The compiler and static analysis tools can do a lot of checking that your test suite or users would otherwise end up doing.

Go's design philosophy is instructive and opinionated, making plenty of useful decisions for you so your engineers don't fall into a choice paradox. For example, Gofmt defines how your code will be formatted so you don't have to. And unit test tooling is built-in, making tests look familiar in every project.

Sometimes it feels like hard work to code in Go, especially if you're used to dynamically typed languages like Ruby or Python, but that extra work is coding type information the compiler needs, or explicitly handling errors. These are both examples of Go's design being more sympathetic to the machine and the realities your program will face at runtime. It has concurrency features built in as language primitives and a preference to pass by value, both of which make it easier to design robust concurrent systems.

Using Go feels slightly laborious and intellectually frustrating if you love FP but, in return, you end up having to make fewer decisions and getting more robust services. The result? A boost to your productivity. It's great for infrastructural services, APIs, and command line tools. Though when it comes to data transformation, prototyping, or view-rendering, Go is good, but by no means excellent.



Docker

Andy Cumine, Software Engineer
Billed as the solution to the 'it works on my machine' problem for building your project infrastructure, Docker is now the go-to tool for containerised infrastructure. Docker's concept of storing build configuration in the project codebase (Dockerfiles) gives you more understanding of your project infrastructure, as this configuration can now be included in the project source control.

The declarative nature of Dockerfiles, and the high quality, up-to-date documentation of the Docker ecosystem makes for an easy-to-learn, simple, and reliable build tool, as well as runtime which scales well with an increasing load. With the Docker for Desktop application bundling the base Docker engine plus docker-machine, swarm, and Kubernetes support, the setup speed from nothing to a full containerised infrastructure is much lower than during Docker's initial release a few years ago.

As a result, it's almost trivial to set up a small-scale replica of your production infrastructure on a developer's laptop. Although your mileage may vary depending on your application's resource needs – some simply don't fit on a single laptop. This brings you the previously hard-to-achieve ability to test your fully integrated system locally. Here, it can be much more easily poked and prodded to reproduce the issues and complex behaviours distributed systems exhibit.

Containers are also being used to run jobs that finish in a finite time – batch processes, build jobs and other similar tasks – as people are realising the full potential of having a portable runtime format that can be run on a platform, such as Kubernetes. GitHub Actions uses this capability, along with community-managed base image ecosystem, to deliver a highly customisable continuous integration service where you don't need to worry about a tool's availability. You can just build them into your build step's base image.

Even enterprise engineering solutions can leverage Docker now. A full enterprise level version of Docker is available with all the support you'd expect to come with an enterprise-level product.

Simply put, if you haven't yet used or at least tried Docker on a project, there are very few reasons not to. The lower level building blocks of Docker are also being open sourced and ownership is being transferred to the community. The container standard is here to stay, regardless of what the future may hold for Docker as a brand and company.

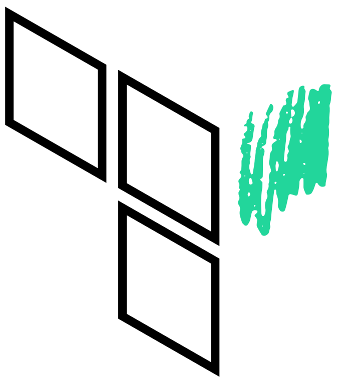
Terraform

Tom Grimley, Software Engineer

Terraform is now a well-established tool for managing your infrastructure. And for good reason. The ability to tear down and reliably recreate your infrastructure is extremely useful. Gone are the days of flicking through a myriad of complex GUIs, or manually typing commands into a remote console. Now everything can be managed straight from the CLI, in your own codebase – with just a few keystrokes.

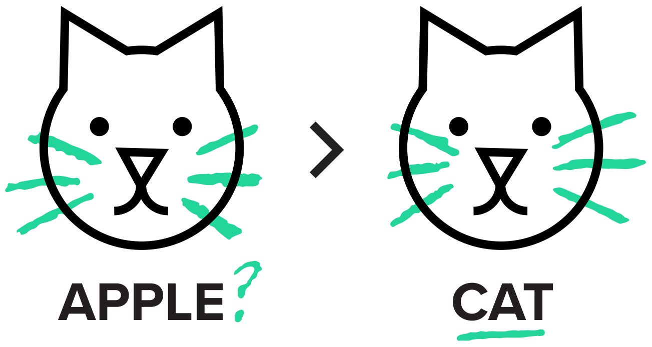
Terraform is a popular open source project with wide support for many different infrastructure providers, so chances are it'll cater for your infrastructure needs. A thriving community means updates and features are regularly provided and, even if not, pull requests can be made or custom providers can be written.

The language itself is easy to pick up, with the declarative HCL providing a simple and easily human-readable syntax for the configuration files Terraform relies on. Knowledge of only a handful of commands is enough to get started using Terraform effectively. For new projects, it's a no-brainer. For the more established projects out there, there'll be a period of slight pain as you collate and write down all your infrastructure into Terraform.



Having all your infrastructure changes committed alongside your application's code simplifies your CI/CD processes and helps document your infrastructure. Much handier than a plethora of dusty old readme files. The greater transparency these committed changes offer during the review process helps makes sure the right architectural decisions are made.

Terraform certainly has a bright future and is well worth the initial investment. Whether you're creating additional environments, performing migrations, responding to the changing demands of your system or recovering from outages, it all becomes far more manageable with Terraform. Infrastructure becomes part of the work of an autonomous delivery team, rather than a separate part of your organisation.



Statically typed languages

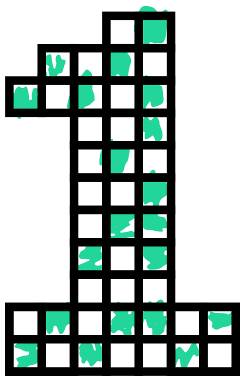
Joe Paice, Tech Director
Static type checking is the process of verifying the safety and reliability of a program based on analysis of the program's source code. While we've had statically typed languages for a long time, in recent years more advanced statically typed languages have become a common choice for web and mobile front-end development. The dynamic, runtime type checking of traditional JavaScript is being replaced with static type checking at compile time in languages like ReasonML, Flow, Typescript, or Elm.

So, why the change? Statically typed languages can bring a number of benefits. Types take what used to be implicit in a dynamic language and make it explicit in a statically typed one. They help document intent, and in many cases can replace entire classes of manually written tests that may have previously been needed to ensure the same level of quality in a dynamic language.

Certain statically typed languages are able to analyse the program and make sure it's 'correct' and cannot fail at runtime in an unexpected way. Some statically typed languages also enable tools to fill in gaps and suggest how to complete the jigsaw of a program, increasing programmer efficiency while maintaining correctness of code. Some tools can even optimise code performance based on the types, resulting in a better performing program.

With these benefits, why would you ever use dynamic languages? Depending on the use case, you might be willing to trade off correctness for being able to rapidly prototype an idea without thinking of every single eventuality. In these instances, a dynamic language may actually be a better fit to avoid jumping through unnecessary hoops to solve a problem. Switching to statically typed languages will also likely demand a different approach to the code itself. Patterns that the programmer may not be familiar with are used to ensure the code will pass static analysis and subsequently compile.

For code that you want to deploy to production, which needs to be reliable, the explicit statement of intent that's verified at build time is super-beneficial. This, paired with the guarantee that what is written won't fail at runtime, makes statically typed languages a great choice for production applications.



Monorepos

Viktor Charypar, Tech Director

The traditional way of managing your organisation or department's codebase – the sum of all the different kinds of code that contribute to your product – is to store each component, service, tool, infrastructure or testing automation separately, each in their own repository. This is clearly inspired by the way open source software development is done, where every small tool or library is built by a different person or team.

In many organisations, this is often not the case. Frequently, the same group of people work on all of these things together, which allows for a much more light-weight management of the various components and their mutual dependencies, enabling the removal of strict repository boundaries and instead of storing all of your codebase in a single repository – a so-called 'monorepo'.

Monorepos have several benefits compared with the traditional strategy, recently dubbed 'poly-repo'. To start with, everyone has visibility of the entire codebase and can contribute (with review) to every part of it, helping to foster a culture of collaboration. Continuous integration and delivery systems can be set up once for everyone. And having a single timeline of changes allows for changes to be made to several components of your overall system, in sync.

Knowing the dependencies between the components, these can be automatically tested, even against all of their known consumers, drastically reducing the feedback time on changes made, when compared to the traditional, semi-manual versioned release workflow. As long as your automated testing is thorough, broken changes don't even make it into the master branch, let alone in front of customers.

With tools like Kubernetes, Istio, and Terraform all the code contributing to your product can be stored together and changed together, allowing you to treat a distributed application like a monolith. The boundaries of various applications and systems are also more flexible this way, enabling easy changes in ownership and helping you avoid the pitfalls of the infamous Conway's Law.

Tooling for monorepos is still not as well developed as we'd like, but there's a clear trend towards teams joining their poly-repos together and the tooling (e.g. GitHub) improves to follow. The only remaining issue to solve if your organisation grows to a large scale (we're talking hundreds of engineers over several years) is scaling the repository management service to match. But fortunately, the industry giants have blazed that trail for us and can offer some solutions.

Our take on monorepos? They make the development of moderately complex systems faster. They make thorough continuous integration and automated testing easier. And they lower the barriers to safely sharing code across projects. Plus, they're a good strategic choice for a flexible source code management strategy – one that'll set you up for the future.

Deconstructed CI services

Xavier Delamotte, Software Engineer
Continuous integration is a common practice in the industry and it's effectively a baseline requirement for continuous delivery. Every commit should automatically trigger a build that will check the code, run automatic tests and eventually deploy the current state of the application for further testing or even to your production environment.

Right now, there are many cloud-based CI services available that are easy to set up. These are sound solutions when you're starting a project. At the other end of the spectrum, for bigger and more complex projects, there's usually already some tooling set up on-premise and specially configured for the needs of your project. Jenkins is a common choice.

Recently, a new breed of CI services has emerged. These split the traditional monolithic CI system into parts, some of which run in the cloud, making them easy to set up. Others can run elsewhere, giving you a finer control over your builds. They effectively deconstruct the CI system into two main parts. First up, orchestration – coordinating with your source code repository and deciding what to build and when. And runtime – actually performing the build, testing, and deployment jobs.

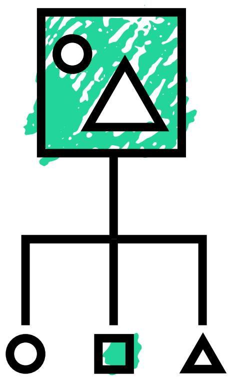
Two of these next gen CI services have caught our attention – Buildkite and GitHub Actions. Buildkite is based on two concepts. Pipelines describe the steps and actions you'd like to execute (managed as code in your codebase). Agents are build runners that will poll the jobs that need to be executed from the orchestration engine at buildkite.com.

You can manage your set of agents directly, run jobs on your own infrastructure, and scale as much as you want. If you need to make more builds, you can just increase the number of agents picking them up. This also means your code never has to leave your network. Pipelines can be defined as static files, saved in your repository, but they can also be dynamic templates. It gives you great flexibility in the ordering of your builds. In our case, in a monorepo we can only build what is needed and independent tasks can run in parallel.

GitHub Actions were introduced during the last GitHub universe and are still in beta. You can define workflow of actions triggered by events on your repository and actions are then executed inside Docker containers of your choosing (described as Dockerfile). Workflows can be created with a visual editor or directly edited as text files, using a subset of the HCL language used by Terraform, for instance. This makes it really easy to build small blocks of operations and tests on your repository orchestrated by the workflow.

There are already many existing open source actions that can be reused in your project, making it easier and faster to set up your own workflow. Unlike Buildkite, Github Actions don't, at the moment, give you control over where or how these actions will be executed – they run on GitHub's runtime.

The common theme in these new services is the understanding that a runtime to run CI jobs isn't that different from a runtime to run your service. And it should really be decoupled from the triggering of jobs and orchestration of what runs in which order. We expect the trend to continue, leading to more flexible, scalable and cheaper-to-operate continuous delivery automation.



Design systems

Rob Brathwaite, Design Director
The term 'design system' has been gaining traction in the last few years. Companies across the globe are either thinking of, are in the process of, or have just finished (if that's even possible) building one. But what's all the fuss about? What are design systems and why do we need them?

In her book, *Thinking in Systems*, Donella H. Meadows described a system as being an 'interconnected set of elements, that are coherently organised to achieve something.' This phrase perfectly encapsulates what a design system is and points to the benefit they provide. That's a great start, but let's get a bit more specific. Nathan Curtis, a giant in the design systems community, crafted this definition of design systems:

> *'Almost always, a design system offers a library of visual style and components documented and released as reusable code for developers and/or tool(s) for designers. A system may also offer guidance on accessibility, page layout, and editorial and less often branding, data viz, UX patterns, and other tools. A design system is adopted by and supported for other teams making experiences. These teams use it to develop and ship features more efficiently to form a more cohesive customer journey. A design system is made by an individual, team, and/or community. While some arise less formally, organisations now dedicate small to large squads to develop and release system versions and processes over time.'*

Based on these definitions, design systems sound really awesome already. But let's look at the value you get for all the investment needed to build one. Picture this – you're a global organisation, building digital products. Your teams are distributed across the world. How do they know what UI code to share? When should they share? How do they share their intent and decisions and make them visible to the rest of the organisation? Design systems have answers to these questions, and more.

Here's a small taster of what a design system can do for you. The ability to onboard a new team member with a link to your design system is powerful. The one-to-one relationship between code in production and description in the system documentation makes your products easily discoverable when onboarding new developers and designers to your team. It will also simplify rotations from one team to another as everything should feel familiar.

A well implemented system will, by its very nature, drive consistency. Having a single point of reference for engineering and UX decisions will smooth some of the kinks in your product delivery and result in a better experience for your users. Time is extremely precious and, let's face it, no one wants to rebuild that red button for the nth time. With a robust set of pre-built components, patterns and guidelines to drive their use, the amount of code and the time needed to develop the UI for your product will decrease. This should also streamline the testing strategy for your app.

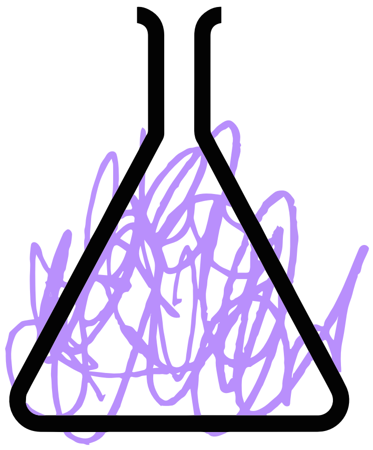
Bringing design and engineering closer to together will ultimately lead to a better product and a healthier culture. Being able to confidently say "The content for this select is XYZ" and have everyone in the room actually know what a select is will minimise confusion, one of the most costly sources of defects.

Design systems are as much about people as they are about building products. A design system is a product that serves delivery teams (groups of people) allowing them to build great products quickly and consistently for their users (also people). Design systems are by no means a silver bullet, but they go a long way towards optimising product delivery and helping teams make brilliant products.

Our engineering principles
Making tech work for you

We're excited to share the guiding principles that underpin all Red Badger's engineering decisions, from our practices and processes to the solutions we design and the tools we choose.

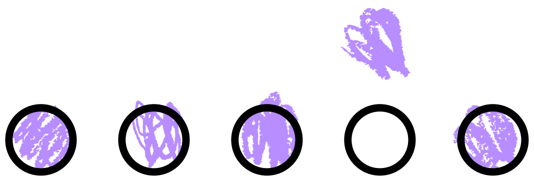
These values empower our teams to experiment and progress autonomously, all while staying aligned. And they help us bring our clients and their customers the best insights and support.



Customer-centric

Customer needs always come first – they're what drives our engineering decisions. This is why you'll always find a UX designer on our teams. Yes, technology can and should support business needs, automating processes, reducing costs and enabling new ways of working. But customers should forever be number one. That's how you'll achieve business success.

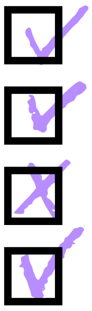
Technology should solve customers' problems in new and better ways. Digital products have an unparalleled ability to gather customer insights and adapt and evolve accordingly. This benefit must be harnessed, to unlock continuous improvement and meet individual users' needs ever more closely.



Autonomy and alignment

Autonomy and alignment is the best way to scale engineering. Alignment focuses teams on the outcomes, not the approach. When this is paired with autonomy, those with the most context on a problem are set up to solve it. This results in better solutions, less reliance on individuals and leads to more efficient and empowered teams.

Clear expectations and accountability are key to achieving alignment, and care needs to be taken to avoid autonomous anarchy. Like in horizontally scalable systems, an engineering organisation successfully built on autonomy and alignment can be scaled by simply adding great teams. Those engineers are also more likely to stick around because they have everything they need to do what they do best.



Continuous delivery and just-in-time decisions

Technological innovation demands constant change. Overhauling digital products once every six months is not enough to keep up, and it's neither smart nor sustainable. We must shorten iterations as much as possible to enable a constant flow of value to customers. Enter continuous deployment – the primary capability engineering organisations should focus on.

Continuous deployment allows responsiveness and extreme agility, and removes an entire class of problems created by scheduled weekly or monthly releases. It also forces building quality into the process and enables an incremental approach to delivery. In turn, this means all decisions can be made just-in-time, when they're needed. Just-in-time decision making allows us to solve problems when they actually arise and take action with as much insight as possible.

Simple solutions and declarative programming

Simplicity matters. When designing systems, complexity should be reduced to the absolute minimum. There are essentially no tools to manage growing complexity, software systems tend to grow orders of magnitude more complex than a single human can understand, and simplifying an existing complex system takes huge amounts of focused effort.

The best way to reduce complexity is to start small and iterate quickly, composing systems from small building blocks that do one thing and do it well. This involves making intentions as explicit as possible at every stage. The approach we favour is declarative programming, focused on outcomes. We describe the goal state and let our tools work out the way to achieve it. When evaluating tools, the ability to declare intent simply and explicitly is the main design feature we look for.

Everything as code

Code is the most complete and explicit specification of a solution. Everything that can be written down as code and automated, should be. This allows us to apply engineering to everything from low-level business logic to the large-scale system design.

What makes code so great? It's stored in version control systems capturing the history and evolution of the solution. And processes written down as code are repeatable and testable. The alternatives are lacklustre – information that's only present in a running system or people's minds, or out-of-date documentation. High levels of automation also allow higher degrees of autonomy. Application logic, testing, configuration, deployment, and infrastructure should all be managed as code.

Always learning

The industry is rapidly changing. At Red Badger, we know the only way to keep up is by staying curious, and always learning. We're constantly discovering new approaches, trying new tools and evaluating the latest technologies. This fuels our engineering capabilities.

Experimentation is key. Others stick to what they know and repeatedly roll out the same strategies. But every time we start a new project, we keep an open mind and look for new tools and technologies. We end up with a balanced blend – around 20% new technologies, and 80% tried and tested favourites.

To push the boundaries of knowledge, it's important to be bold and bet on upcoming, innovative trends. Sometimes it's better to ask for forgiveness rather than permission and learn from safely failing. Experimenting, learning and applying the results is the best way to stay ahead.

A
Agility ... 4, 25
Anarchy ... 6, 24
Android ... 15
Always learning ... 26
Autonomy ... 5, 6, 6, 11, 24, 26
— Autonomous delivery teams ... 18
APIs ... 4, 11, 13, 16

B
Buildkite ... 21

C
Cloud-native applications ... 14
Cluster orchestration ... 10, 13
Component libraries ... 15
Containers ... 10, 13, 17, 21
— Containerised applications ... 13
Continuous delivery ... 5, 6, 8. 9. 10, 11, 20, 21, 25
Continuous deployment ... 25
Continuous integration (CI) ... 17, 20, 21
CI/CD ... 9, 18
CLI (Command-line interface) ... 18
Common problem areas ... 3, 4, 5, 6, 7, 8
Complexity ... 14, 25
Conway's Law ... 20
Cross-functional teams ... 4, 8
Customer-centric ... 24

D
Data-driven product management ... 4
Data transformation ... 16, 22
Declarative programming ... 25
Deconstructed CI services ... 21
Delivery processes ... 4
Design systems ... 15, 22, 23
DevOps ... 2, 12
Distributed systems ... 17
Docker ... 5, 10, 11, 13, 17, 21
Dynamic languages ... 19

E
Elm ... 19
Empowering teams ... 6, 24
Engineering culture ... 7, 8
Engineering principles ... 3, 24, 25, 26
Everything as code ... 11, 26
Experimentation ... 7, 26

F
Flow ... 19

G
Get set for what's next ... 2
GitHub Actions ... 17, 21
Golang (Go) ... 16
GUI (graphical user interface) ... 18

H
HCL ... 18, 21

I
Infrastructure (as code) ... 3, 5, 10, 13, 17
— Infrastructural services ... 16
— Infrastructure solution ... 14
Introduction to TechLab Report ... 2
iOS ... 15
Istio ... 2, 14, 20

J
JavaScript ... 15, 19
Jenkins ... 21
JSON ... 13
Just-in-time decisions ... 25

K
Kubernetes ... 2, 5, 7, 10, 13, 14, 17, 20

M
Microplatforms ... 13
Microservices ... 2, 14
Mobile front-end development ... 19
Monorepos ... 20, 21

N
Native ... 7, 15

O
Open source ... 7, 17, 18, 20
Orchestration ... 2, 10, 13, 21
Observability ... 2, 6, 11, 12

P
Pipelines ... 9, 21
Principles ... 3, 24, 25, 26
Problem areas ... 4-10
Production applications ... 19
Prototyping ... 4, 16, 19
Python ... 16

R
React Native ... 7, 15
ReasonML ... 19
Reliability ... 5
Risk management ... 4
Ruby ... 16

S
Service mesh ... 2, 12, 14
Statically typed ... 16, 19
Silo ... 7, 8
Specialisms ... 9-12
Strategic choice ... 15, 20
Storybook ... 15
Speed of delivery ... 4, 15
Scale ... 5
Strategic direction ... 6

T
Technology ... 5, 13-23
Terraform ... 10, 18, 20, 21
Testing strategy ... 8, 23
— Testing ... 11
Tools and Technology ... 13-23
Typescript ... 19

U
UI library ... 15
User interface ... 15
User interface (UI) ... 17, 20

V
Virtual machine (VM) ... 10, 13

Y
YAML ... 13

Contributors

Thanks to all the Badgers who contributed to this report:

🟩 29 **Ac** Andy Cumine	🟩 97 **Gd** Greg Dorward	🟩 142 **Jk** Jan Kuehle	156 **Ac** Abigail Coe	102 **Ai** Andrei Ionescu	113 **Ah** Andy Haines	42 **Cb** Chris Brookes	46 **Cg** Chris Gray
🟩 76 **Jp** Joe Paice	🟩 52 **Jy** Jon Yardley	🟩 88 **St** Sam Taylor	127 **Cc** Chris Caller	121 **Cw** Cris Wilgenhoff	134 **Dt** Dan Train	143 **Db** David Basalla	138 **Ds** Declan Slevin
🟩 74 **Sw** Sam White	🟩 01 **Sh** Stuart Harris	🟩 120 **Ts** Tim Stott	45 **Dp** Dominik Piatek	168 **Ec** Ed Compton	07 **Hl** Haro Lee	152 **Jr** Jack Rehaag	166 **Js** Jenny Sharps
🟩 148 **Tg** Tom Grimley	🟩 157 **Tk** Tornike Keburia	🟩 125 **Mp** Matt Paul	04 **Js** Joe Stanton	178 **Jp** Joseph Popoola	115 **Jo** Julian Osman	151 **Kp** Kyle Patel	144 **Mh** Mark Holland
🟩 46 **Mf** Monika Ferencz-Szabo	🟩 123 **Rb** Rob Brathwaite	132 **Mi** Mishal Islam	34 **Nc** Nico Castro	140 **Pm** Pedro Martin	161 **Rg** Rane Gowan	25 **Rc** Robbie McCorkell	101 **Ss** Sam Smith
🟩 145 **Xd** Xavier Delamotte	🟩 17 **Vk** Viktor Charypar	09 **Sb** Samera Butt	180 **Sa** Simon Ashbery	165 **Tb** Tom Barwick	128 **Tw** Tracy Wu		

🟩 Special thanks

Want to dive deeper into the topics covered in this report?
Join us at our upcoming TechLab event.

Sign up now