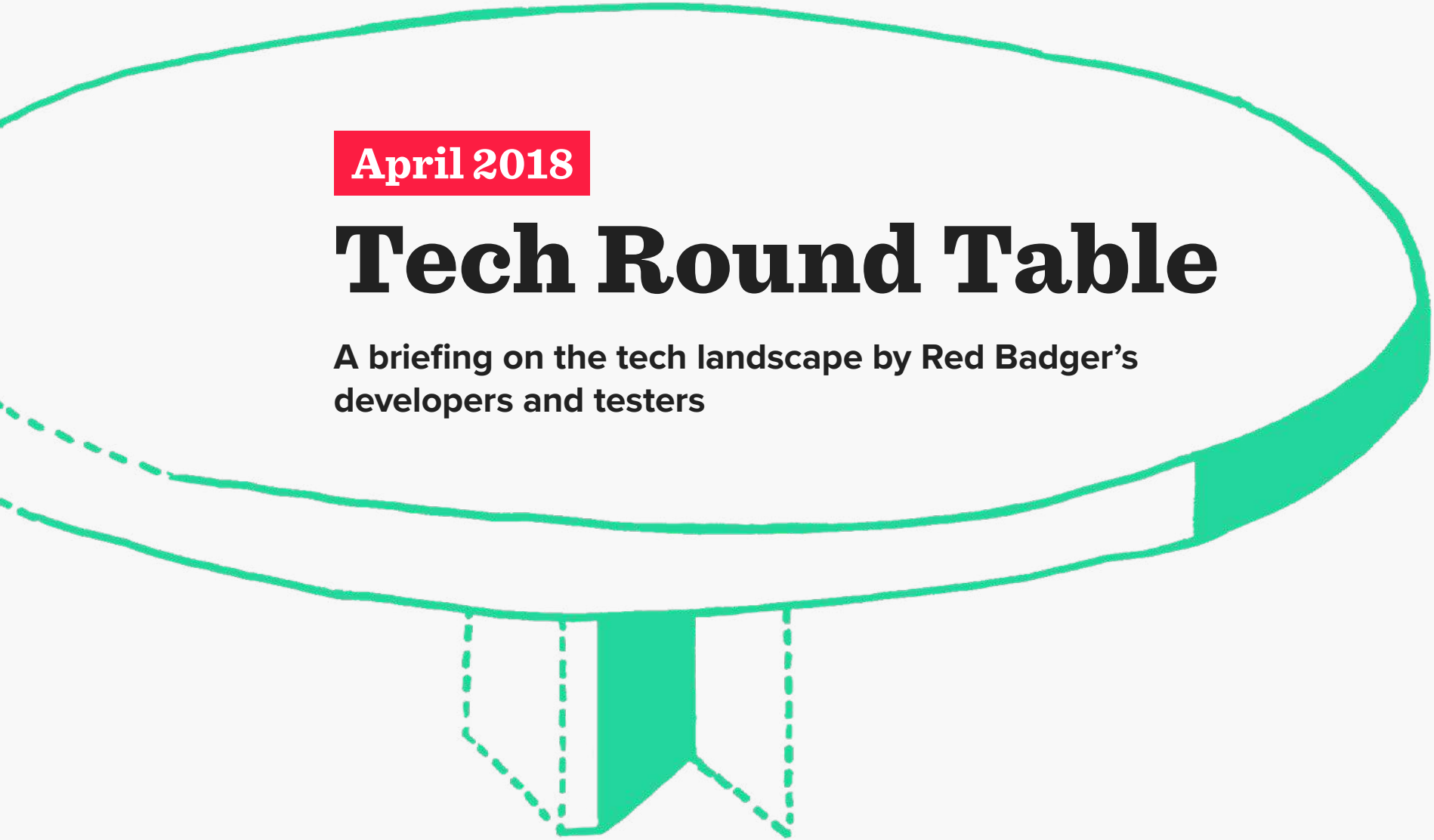
RED BADGER



April 2018

# Tech Round Table

A briefing on the tech landscape by Red Badger's developers and testers

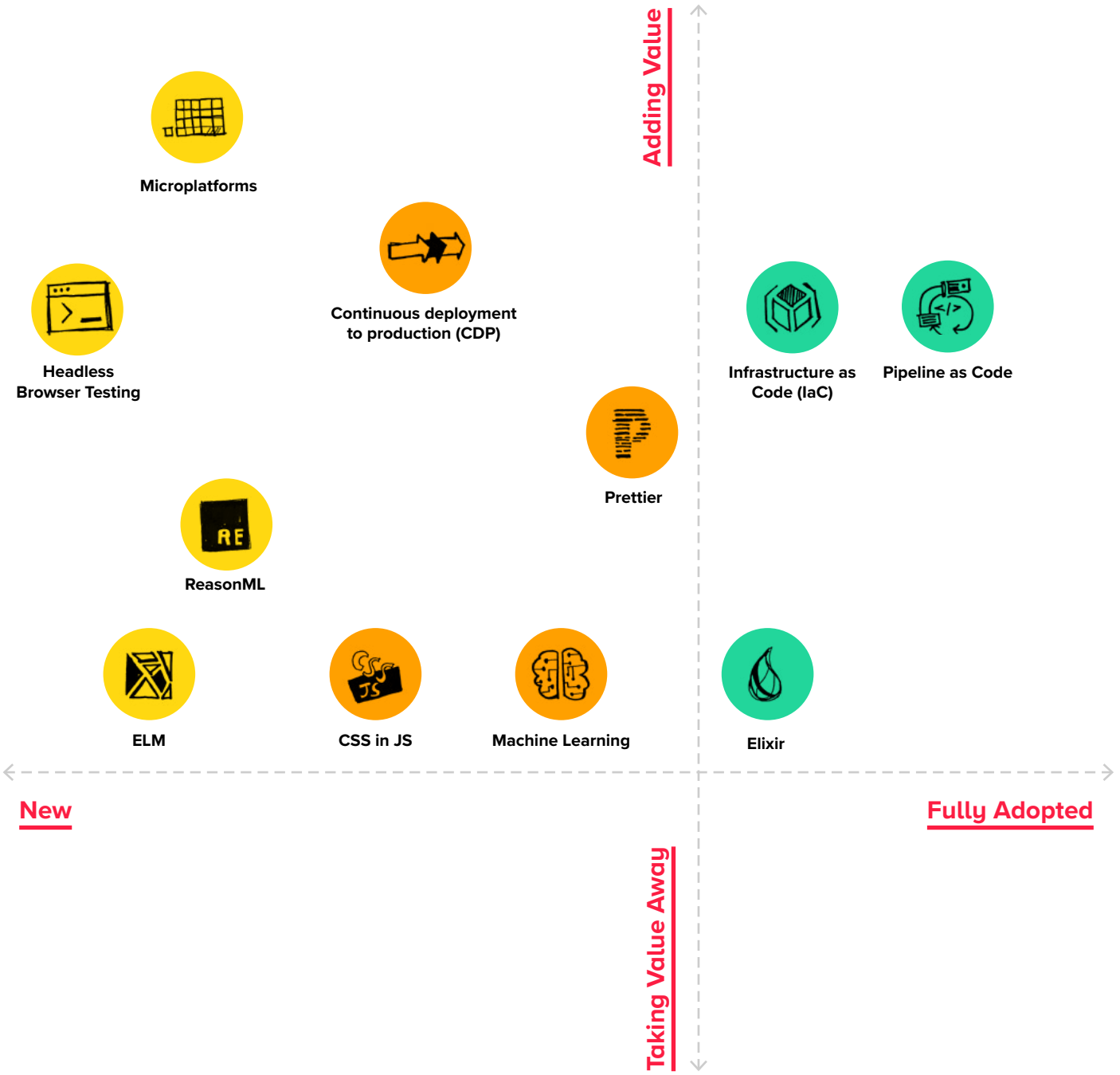
# Introduction

We've always been able to say "There's never been a better time to be a software developer" because innovation in software engineering is continually accelerating. This year, however, it feels even more so. Two things. Firstly, a very mature Open Source (OSS) mentality within the industry and secondly, a huge drive to automate all the things. For the first time, it seems, it's now possible to automate every part of the stack, top to bottom, and to drive that automation from code that evolves in the same, agile, way that application code does. Everything as Code (EaC) is a new mantra. This evolution of the meta is giving us orders of magnitude more efficiency and accuracy as we deliver our products.

Each of the tech we've chosen in this Tech Roundtable drives (or has been driven by) one, or both, of these concepts (OSS and EaC).

There is a multitude of new and exciting languages, tools and techniques whose consistency is driven by global consensus. Finally, we're converging as an industry, and it feels great. Kubernetes, for example, is possibly the fastest growing open source project of all time and, together with Docker, forms a new consensus on container orchestration (see the section on Microplatforms below).

Anyway, enjoy!



# Contents

ADDS VALUE • ADOPTING

# Infrastructure as Code (IaC)

*Infrastructure as Code (IaC) refers to the definition and management of IT infrastructure (e.g. networks, virtual machines, load balancers, firewalls, etc.) in a declarative manner, i.e as source code. Declarative IaC has the powerful property of immutability: one particular set of inputs (i.e. version of source code) will only ever produce one configuration of infrastructure.*

*As a result, there are business advantages in terms of speed, risk, and cost; as well as making it easier to incorporate true DevOps in to a cross-functional team.*

Setting up IT infrastructure used to involve screwdrivers; and typing shell commands in to an appliance using a keyboard attached to a serial port in a noisy, cold data center.

The processes were slow, expensive, and very prone to human error. Although disk imaging helped remove some amount of repetition, each server had to be manually configured with the right values to make it functional and usable. Updates and patches to these machines also had to be applied manually, to each machine, and this effort required dedicated teams. The manual aspect to all this made it incredibly easy to end up with 'Snowflake servers' - ie servers that became unique within the estate.

Virtual Machines brought us a long way forward. Virtual networks even more so.

The dawn of cloud computing hailed in a range of vendors providing us with a full offering of Infrastructure as a service (IaaS): VMs, virtual networks, block storage, firewalls, load balancers, gateways, specialised software bundles (e.g. AWS Lambda), and more. Everything required to host a stack - anything from a single application, to a sprawling orchestration of services - could now be created 'on-demand'.

By themselves, cloud platforms do not change the issues presented by doing things manually: resources can still, of course, be manually provisioned and configured.

Enter stage: declarative Infrastructure as Code (IaC).

Infrastructure as Code (IaC) is the definition and management of IT infrastructure resources in a declarative manner, e.g. source code.

Declarative IaC has the powerful property of immutability. The function that applies the code to the target environment (e.g. a cloud provider) is deterministic, which makes the creation of multiple, identical 'environments' trivial: for performance testing, security testing, etc., you can easily provision an environment that is almost identical to what your production environment will be. Having development and production environments be as-close-as-possible helps to catch bugs earlier which may only appear in an integrated environment.

*(continued on next page)*

← Back to contents

ADDS VALUE • ADOPTING

# Infrastructure as Code (IaC)

*Infrastructure as Code (IaC) refers to the definition and management of IT infrastructure (e.g. networks, virtual machines, load balancers, firewalls, etc.) in a declarative manner, i.e as source code. Declarative IaC has the powerful property of immutability: one particular set of inputs (i.e. version of source code) will only ever produce one configuration of infrastructure.*

*As a result, there are business advantages in terms of speed, risk, and cost; as well as making it easier to incorporate true DevOps in to a cross-functional team.*

*(continued)*

Storing infrastructure in code allows us the same benefits as regular code: immutable version control (e.g. Git), self-documentation, peer review, smaller incremental changes, and more. Version control and peer review also provide audit trails; which is especially important to many larger corporate users.

It is not a substitute for proper understanding of concepts such as networking, load balancing, scaling, and infrastructure security. However, IaC does makes these things far more accessible: bridging the gap between Dev and Ops, helping to incorporate and cultivate a [true] DevOps culture amongst cross-functional teams.

IaC provides us another key benefit: transparency. The infrastructure that surrounds our application is no longer hidden away. There is no more 'magic'. Engineers can understand, reason about, and adapt a configuration far more easily; which is crucial for when things go wrong.

The benefits of IaC are clear, and we would recommend it be used in anger wherever possible. Benefits are brought both to engineering and the business: IaC provides a great means for auditing change; infrastructure can be deployed and changed faster; there is a lower knowledge overhead; risk is lowered compared to manually provisioned infrastructure; and there are clear cost savings (perhaps no need for a dedicated infrastructure team).

It is also analogous to other technologies we've explored in this roundtable - Microplatforms are designed to achieve many of the same goals; and continuous deployment to production (CDP) can also encompass infrastructure.

Looking further ahead, it's easy to imagine how IaC might lose prevalence as cloud platforms provide greater levels of abstraction for our applications and how we organise them: orchestrations of containers (e.g. Kubernetes), and functions as a service (e.g. AWS Lambda) are great examples of where most if not all of the infrastructure concerns are managed for us. However, most solutions to non-trivial problems will still require some level of customised setup - so it's safe to say that IaC is here to stay. ■

Ref: Morris, Infrastructure as Code
Ref: Martin Fowler: Infrastructure as Code

ADDS VALUE • ADOPTING

# Elixir

*Elixir is a dynamic, functional language designed for building scalable and maintainable applications. It runs on the battle-tested Erlang VM, known for running low-latency, distributed and fault-tolerant systems, and is being successfully used in both web development and embedded software.*

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

It runs on the battle-tested Erlang VM and is underpinned by lightweight processes communicating via message passing. This approach, in conjunction with immutable data structures, allows code to be safely executed concurrently, scaled within and across nodes, and supervised for failures.

Fault tolerance in Elixir (and Erlang) applications is achieved by taking an approach to error handling that is completely different to that of most other languages. In Elixir, errors are accepted as a fact of life (particularly when dealing with networks, file systems, and other third-party resources), and processes are designed to crash when unexpected errors occur. However, processes are isolated, so the impact of crashes is minimised, and supervisors also ensure that the essential processes are restarted in a known-good state.

Elixir was created by José Valim, a well-known rubyist and alumnus of the Ruby on Rails core team, and its readable syntax will feel natural to Ruby programmers.

Developers who use the Elixir language will enjoy its modern features and build tools, alongside a set of mature libraries and design patterns that have evolved from the Erlang community's 20 years worth of experience in building robust applications using the open telephone platform (OTP) framework.

Another strength of the Elixir ecosystem is the Phoenix web development framework, which uses Elixir's pattern matching and powerful Lisp-style macro system to produce expressive and performant code that can achieve incredible response times (often measured in microseconds).

On the embedded software front, Nerves is an increasingly popular library that packages Elixir applications into a minimal Linux distribution booting directly to the Erlang VM. Rather than dropping down to a low-level systems programming language, developers can enjoy the productivity and robustness of Elixir whilst targeting pretty much any hardware that can run embedded Linux.

Since our last round table, Elixir and its ecosystem has continued to mature,

*(continued on next page)*

← Back to contents

ADDS VALUE • ADOPTING

# Elixir

*Elixir is a dynamic, functional language designed for building scalable and maintainable applications. It runs on the battle-tested Erlang VM, known for running low-latency, distributed and fault-tolerant systems, and is being successfully used in both web development and embedded software.*

*(continued)*

with new versions of both the language and the Phoenix framework bringing continued improvements to the developer experience. Particularly interesting in the latest version of Phoenix is the introduction of "contexts" that guide the architecture of applications towards business-domain-focussed modules, which offer some of the benefits of loosely-coupled microservices even within a monolithic application. ■

ADDS VALUE • NEW

# Headless Browser Testing

*It's a way to run a browser in a headless environment. Essentially, running Chrome without chrome! It brings all modern web platform features provided by Chromium and the Blink rendering engine to the command line. Other browsers have followed suit, with Firefox now offering a headless mode as well.*

Browser testing is important as it helps to catch issues, which are not easily reproducible in unit tests. These can range from security related issues like CORS to web platform feature support.

Historically, automated browser testing has been hard. The standard solution has been to run Selenium, either singly or within a distributed grid. Selenium brings a common protocol to run the same tests in different browsers - even on remote servers. But running real browsers comes with a cost:

- The browser startup is slow.
- To run the tests on CI servers you need additional configuration for a display server.

To get around these issues we have often used a headless browser like PhantomJS instead. This is nice, because it starts faster and works on CI environments without further configuration. However, it comes with other issues:

- The browser is not up to date and does not support the latest web platform features.
- People don't actually use this browser, so their experience might differ slightly.

The good news is, that both Chrome and Firefox have recently added support for a native headless mode. This means you can easily get real browsers running with fast browser startup and zero CI configuration.

In addition we see browsers and the community developing features and tools around headless mode:

- Chrome and Firefox allow to take screenshots of a webpage from command line.
- Puppeteer is a very easy to use Node.js library to control headless Chrome.
- Rendertron uses a headless Chrome inside Docker to provide a rendering API for modern websites. It can be used to enable server side rendering for progressive web apps. ■

← Back to contents

Machine Learning →

ADDS VALUE • ADOPTING

# Machine Learning

*Machine learning is the science of getting computers to act without being explicitly programmed.*

Artificial intelligence has been the topic of research for decades, and many businesses have been successfully applying the methods of machine learning for many years to solve complex tasks, which are impossible to tackle with traditional algorithmic solutions. Typically these are problems that are easier for a human than for a computer, requiring human-like knowledge, like recognising objects in a picture, driving a car or suggesting a movie a person might like. Applying machine learning used to require a lot of specialist knowledge and custom development, but it is now becoming commoditised and accessible to everyone.

The basic principle of machine learning is building a model of a problem based on a set of examples, often quite large, in a process referred to as learning or training. Machine learning models broadly fall into two categories: classification models, which sort the inputs into predefined categories (e.g. recognise an object in a photo is a dog) and regression models, which learn a relationship between different attributes of a system (e.g. given product information and your previous purchases, how likely are you to buy the product). Recently, deep learning methods, inspired by the structure of biological neural networks, have become popular and quite successful at solving problems previously considered too hard for a computer to become better at than a human. A perfect example is the success of AlphaGo, a program playing the board game Go, which beat one of the world's best players in 2016 and was updated in 2017 to learn by playing against itself, rather than from recorded human plays, surpassing the original in three days.

Machine learning models learn from examples and require training data to provide good results. The underlying mathematical methods work with abstract numbers and the quality of the predictions largely depends on the way real-world inputs are encoded before they are given to the model. Results returned by a model will never be perfect - in a sense, machine learning makes the same kind of mistakes that humans would make, but often surpasses humans in quality by several orders of magnitude. The underlying structure and mechanics of the models also have nothing to do with the problems they

*(continued on next page)*

← Back to contents

ADDS VALUE • ADOPTING

# Machine Learning

*Machine learning is the science of getting computers to act without being explicitly programmed.*

*(continued)*

are applied to, which makes it quite difficult to understand the reason for a particular result and fix specific issues.

Machine Learning is now accessible as a service from multiple cloud services providers, both as generic models applicable to bespoke problems and as services tailored to computer vision, natural language processing and other common applications. They provide hardware acceleration and tight integration with their large data stores to support machine learning applications at scale. All of this makes machine learning easier to use than ever, although some specialist knowledge is likely to still be required in order to achieve good results. ■

ADDS VALUE • NEW

# Elm

*Elm is a pure, functional language that compiles to JavaScript. It is a delightful language to build reliable websites and webapps. It has a very strong emphasis on simplicity, ease-of-use, and quality tooling.*

Elm is a functional language that compiles to JavaScript. It competes with projects like React as a tool for creating websites and web apps. Elm has a very strong emphasis on simplicity, ease-of-use, and quality tooling.

Evan Czaplicki, the creator of Elm, is very considered in his approach to its development. All of the individual pieces of Elm are very well thought out. What makes Elm truly amazing is that the whole of Elm is much better than the sum of its parts.

Elm has a core focus on being easy to learn. This is embodied in both the getting started guide and very much the supportive community around it. The guide is simple and easy to follow. Public Elm packages are well documented. Elm's Slack is alive. This gives developers a wealth of resource to draw from when they have questions and this resource is growing.

Another piece that makes Elm easy to learn is it's compiler. Elm's compiler is very much an assistant rather than an adversary. Not only does it detects bugs, it helps developers understand why. It gives specific hints that helps developers write better code. Ultimately, it makes programming faster and easier.

Elm mitigates classes of problems that in other languages, developers would need to think about. For example, Elm does not allow the use of null or undefined. Instead it captures this intent in an explicit way the use of a "Maybe". By forcing the use of a "Maybe", Elm ensures that developers always handle the "Nothing" case, resulting in explicit and robust code.

Unlike handwritten JavaScript, Elm code does not produce runtime exceptions. Instead Elm uses type inference to detect problems during compilation and give friendly hints when compilation fails. Any problems are resolved during development and never have the opportunity to make it in front of end users.

What really intrigues me is what happens to an Elm project over time. Because Elm enforces semantic versioning, upgrading dependencies is relatively trivial, especially for minor and patch level changes. For those trickier upgrades, changes and refactors the compiler has the developers back and guides them to success. As a side effect, Elm projects tend to stay more nimble, enabling development teams to keep developing without accruing tech debt.

*(continued on next page)*

← Back to contents

ADDS VALUE • NEW

# Elm

*Elm is a pure, functional language that compiles to JavaScript. It is a delightful language to build reliable websites and webapps. It has a very strong emphasis on simplicity, ease-of-use, and quality tooling.*

*(continued)*

The future of Elm looks bright. The current roadmap is focused is on creating single-page apps in Elm. This includes:

- Server-side rendering
- Tree shaking (trimming out unused code)
- Code splitting (cutting up code into smaller chunks for better caching)
- Lazy loading (only sending the code chunks needed for a particular page)
- Expanding web platform support
- Unfortunately progress can be excruciatingly slow and is frustrating for those waiting. Consideration takes time. That aside Elm is an amazing tool to build web applications with. We are definitely looking for an opportunity to use it when the right problem comes along. ■

ADDS VALUE • NEW


# ReasonML

*ReasonML, created by the same person that originally created React, is a new syntax for the OCaml language that is meant to be welcoming to JavaScript developers. Reason lets you write simple, fast and quality type safe code while leveraging both the JavaScript & OCaml ecosystems.*

ReasonML is a powerful functional language built by Facebook, and created by Jordan Walke who initially created React. It is not a brand new language, but a syntax over the language OCaml.

OCaml is a very mature, battle tested language with extremely fast native compilation. OCaml provides a functional syntax, with immutable data structures and 100% type safety. The first prototypes of React were actually built in a language called SML, a distant cousin of OCaml.

Reason provides a syntax on top of OCaml that caters to developers more familiar with Javascript, whilst maintaining the powerful features of the underlying Ocaml language. Out of the box Reason uses Ocaml's native compiler, to compile to bytecode that can run on a wide variety of platforms. In addition provides the capability to compile to Javascript with the help of a tool called BuckleScript.

Bucklescript is a tool built by Bloomberg which compiles OCaml or Reason into readable Javascript with smooth interoperability with existing Javascript libraries. In fact due to optimisations that Bucklescript can make when using immutable, type safe data structures, code written in Reason or Ocaml and compiled by BuckleScript into Javascript can have better performance out of the box than the same functionality written in Javascript.

With the tools that Bucklescript provides incorporated into the language, Reason can be fully interoperable with existing Javascript codebases and npm packages. Using plugins for Babel or Webpack, Reason can be incrementally introduced into another Javascript codebase, allowing developers to gradually introduce Reason's benefits over time.

Along with providing a familiar syntax for Javascript developers, Reason also aims to provide first class support for React. Reason React is a separate library that provides React bindings for use in Reason applications compiled to Javascript. Reason React expands on the React framework and leverages the features of the Reason language to provide language level application routing, data management and component composition. Writing React applications in Reason using Reason React

*(continued on next page)*

← Back to contents

ADDS VALUE • NEW


# ReasonML

*ReasonML, created by the same person that originally created React, is a new syntax for the OCaml language that is meant to be welcoming to JavaScript developers. Reason lets you write simple, fast and quality type safe code while leveraging both the JavaScript & OCaml ecosystems.*

*(continued)*

allows you to build applications that are safe, statically typed, simple and lean.

By providing a familiar syntax, and gathering together an ecosystem of tools and libraries, ReasonML has the potential to push the Javascript community further towards functional programming, strong type safety, immutable data structures and native compilation with all of the technical and productivity benefits those features provide. With its tight integration with React, Reason might become the recommended way to write React applications in the future. ■

ADDS VALUE • ADOPTING

# Pipeline as Code

*This forms part of the 'Everything as Code' evolution, enabling the same code repository to not only describe and implement the required functionality (through source code), but also how this functionality will be built, tested and deployed to production systems.*

Having the build and deployment pipeline as code (PaC) builds on the benefits of having Everything as Code (EaC), e.g. immutable version control, audit trails, peer reviews, textual representation, and knowledge sharing. Furthermore, allowing the product team to control the pipeline helps to embed professional engineering practices that help the team understand the full journey of their artefact, from dev, build and production deployment.

At a high level, a pipeline consists of three parts, namely: building and testing the artefact, assuring quality, and orchestrating deployment to production. The build and test stage of the pipeline compiles the code, runs initial tests (e.g. unit tests, linting) and outputs an artefact (or library, module, UI, application, etc) for storage in a repository. The quality assurance part of the pipeline may run additional processes across the codebase - think security, cyclomatic complexity, bug detection, code coverage metrics, etc. The final stage is the orchestration of deployment to production (through a number of lower environments - dev, staging, production for example). The quality assurance stage also runs against each of the environments as the deployment is orchestrated. For example, integration tests are performed in higher environments, with testing being performed against more production-like systems.

PaC provides an point-in-time representation that describes not only the functionality of the deployment artefact (i.e. the actual programming source code), but how it is built and deployed. This textual representation, when under control of the product team, resides within the source controlled codebase so that any member of the team (or indeed organisation) can review, comment upon, and improve.

An additional benefit of having PaC is that this drives engineers to think beyond the code that they are writing to the wider context of how this code will be used - both from a pipeline and end-user perspective. Engineers now have to take into consideration the build and quality controls that go into professionally deploying an artefact into production. This is especially useful when the team is also on support - ensuring that there

*(continued on next page)*

← Back to contents

ADDS VALUE • ADOPTING

# Pipeline as Code

*This forms part of the 'Everything as Code' evolution, enabling the same code repository to not only describe and implement the required functionality (through source code), but also how this functionality will be built, tested and deployed to production systems.*

*(continued)*

are no knowledge gaps or dependencies in how the entire application interacts within the organisation's technical estate. More knowledge leads to a more rapid turnaround time to fix production issues - as there are no external dependencies to rely upon - and allows the team to integrate additional learnings (extra quality controls, deeper tests) into the pipeline that reduces errors and include better quality controls. ■

← Back to contents

Continuous deployment to production (CDP) →

ADDS VALUE • ADOPTING

# Continuous deployment to production (CDP)

*Continuous Deployment (CD) is the process that takes validated Features from Continuous Integration and deploys them into the production environment, where they are tested and readied for release.*

Continuous integration and continuous deployment (CI/CD) are well-known practices which are become more widely used by tech teams to deliver software. Quite often, however, these pipelines only go as far as a QA 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. Whilst this is great for the development process you can then hit the same issues you are trying to avoid during development in your deployment into production.

Continuous deployment to production essentially takes these practices and extrapolates them through to production. Simply put this means when a feature is "done" and merged into the master branch of a repository,the build pipeline kicks in to run unit tests, test automation suites, and any other automated quality checks that you may have in place, eventually ending with a production deployment.

In order to continuously deliver software into production, whilst maintaining quality, it is paramount to ensure that your entire build and deployment pipeline is automated. Making small changes little and often has been proven to help de-risk deployments. Small changes made in the morning and deployed in the afternoon, for example, allow for any issues after deployment to be easily tracked down, debugged and fixed quickly although if your automated checks are designed in the right way these events will be a rarity.

Having a high level of confidence in what you are shipping is something that is a prerequisite for CDP to work, and work well. If deployments to production only run every three months, running these scripts, or in some cases, manual deployment can be scary, risky and problematic. Automating this entire process and running it multiple times a day helps to maintain confidence in the pipeline resulting in the triviality of deployments into production. Immutability in your applications and even your infrastructure allows for deterministic deployments which adds another level of confidence.

The benefits of constantly shipping small changes to production are not only limited to the realm of technology. User experience

*(continued on next page)*

← Back to contents

ADDS VALUE • ADOPTING

# Continuous deployment to production (CDP)

*Continuous Deployment (CD) is the process that takes validated Features from Continuous Integration and deploys them into the production environment, where they are tested and readied for release.*

*(continued)*

and visual designers also benefit from these changes as the iteration loop, given your process, can be significantly reduced allowing a real measure and learn approach to how you develop your applications. ■

ADDS VALUE • ADOPTING

# Prettier

*Prettier is an opinionated code formatter with support for multiple languages and frameworks. It removes all original styling and ensures that all outputted code is consistent, increasing readability and eliminating arguments (think the 'religious' wars over tabs vs. spaces).*

Prettier is an opinionated code formatter. It was created by James Long to format modern JavaScript and JSX, inspired by Go's `gofmt` and Reason's `refmt` tools. It now supports typed JavaScript flavours such as Flow and TypeScript as well as CSS, Less and SCSS.

Enforcing a consistent code style across projects makes code easier to read and understand for all team members. It can be especially helpful for newcomers unfamiliar with a project's codebase. However coming to a consensus on the style to adopt can be difficult, sparking seemingly endless "tabs vs. spaces" type debates that are irrelevant to the product being built. Policing the style is also a challenge, resulting in a lot of "nitpicking" in code reviews.

Static analysis tools such as ESLint can help flag up style problems alongside other rules but require extensive configuration and have very limited ability to fix issues automatically. Prettier takes a different approach by taking the maximum line length into account. It parses the code into an Abstract Syntax Tree, throwing away all the original formatting, and reprints it from scratch using an algorithm based on a paper by Philip Wadler.

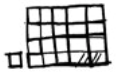
Extensions are available for popular code editors which can format your code with Prettier the moment you press "save", allowing you to stop worrying about formatting and focus on application logic. This is remarkably freeing!

Prettier provides very few configuration options and requires your team to adopt its opinionated code style. This may not match your team's preferences exactly, however Prettier's automatic formatting is so tremendously useful that it soon wins you over. Wide adoption of Prettier in the community is beginning to result in a consistent code style across many open source projects.

In existing projects the automatic formatting of files can sometimes make pull requests more difficult to understand and review due to many small formatting changes alongside logic changes. This can be mitigated by regularly running Prettier across the codebase.

Prettier is one of those tools which is so useful you look back and wonder how you ever did without it. ■

ADDS VALUE • NEW



# Microplatforms

*Extremely high levels of automation in the area of infrastructure provisioning and container orchestration have recently enabled a capability we are calling microplatforms - small, and fully capable platforms for microservices applications. This is a concept that allows a cross-functional team to manage, create and destroy (but not modify, because Immutable Infrastructure) their own short-lived platforms on which they can choreograph a collection of microservices. 100% automation (terraform, Dockerfile, Kubernetes, yaml config etc) ensures environments are identical, repeatable, disposable and cheap. The best enabler for Continuous Deployment into Production we have seen to date.*

Until recently it made sense for large organisations to build shared platforms by creating teams that, often manually, provisioned servers, VMs, and supporting software. It was a high cost activity so it made sense to amortise that cost across the business. Now that cloud providers have mastered infrastructure provisioning by API, full automation is possible. Today that time is better invested in writing code that declaratively specifies what the platforms, networks and supporting infrastructure should look like, down to the last detail. This makes it possible to create in seconds or minutes what would have taken months before. Because it's fast and declarative, it becomes cheap, repeatable, reliable and fully auditable.

In the last few years, containers (Docker) and container orchestrators (Kubernetes) have pushed that level of automation up the stack, giving us these same advantages all the way to our applications and their individual microservices.

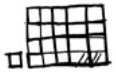
This top to bottom automation now allows cross-functional teams to create, manage and destroy environments for their applications with little effort, saving time and money, whilst improving on reliability and reducing MTTR. Their conversations with the shared "platform teams" move from "Please can you install MongoDB on this VM?" to "Please will you accept this Pull Request?". Now everyone can take advantage of the change and there is code that can be evolved in the community. For already accepted practices, no coordination outside the team is necessary.

When everything becomes code, that's where the sharing and reuse happens. That's where we improve things. And then we just build and deploy our code to do those things. Things like creating VMs, subnets, clusters, service meshes and microservices. It's a forward-only paradigm that always starts with changing *source* code. If a subnet needs a bigger CIDR block, we change the Terraform code and re-apply it. If a microservice needs an egress route to a service on the internet, we change the Istio yaml file and re-apply that. If a microservice has a bug, we change its source code and re-apply that (through the CI/CD pipeline).

*(continued on next page)*

← Back to contents

ADDS VALUE • NEW



# Microplatforms

*Extremely high levels of automation in the area of infrastructure provisioning and container orchestration have recently enabled a capability we are calling microplatforms - small, and fully capable platforms for microservices applications. This is a concept that allows a cross-functional team to manage, create and destroy (but not modify, because Immutable Infrastructure) their own short-lived platforms on which they can choreograph a collection of microservices. 100% automation (terraform, Dockerfile, Kubernetes, yaml config etc) ensures environments are identical, repeatable, disposable and cheap. The best enabler for Continuous Deployment into Production we have seen to date.*

*(continued)*

If we ensure that every time we re-apply our code, it's idempotent (i.e. we can apply it as many times as we like and the net result will be the same), then we can re-apply the whole stack if we want to, and only the things we changed will be replaced.

This is the core concept behind microplatforms. The ability to splat our application (and if necessary, it's supporting infrastructure) onto a provider-agnostic "substrate" whenever we want to. We can create and destroy production-identical, ephemeral environments to do performance testing, for example, in minutes. Even production environments can (and should) be short-lived. You wouldn't create new production environments on every deployment (there could be hundreds of those each day), but they would be recreated as the underlying code evolves (e.g. a security patch is released).

Microplatforms have a small blast radius, are cheap to create and destroy, enable continuous deployment into production, reduce risk and MTTR, increase reliability and repeatability, can be managed within the cross-functional team. They can massively improve a team's velocity and help prevent large organisations from grinding to a halt behind inter-team dependencies. ■

ADDS VALUE • ADOPTING

# CSS in JS

*As the name would suggest, this is all about defining cascading stylesheets in JavaScript. It provides an abstraction of CSS using JavaScript as a language to describe styles in a declarative and maintainable way. The CSS in JS is compiled either at runtime or server-side providing high performance with very low overhead.*

When CSS debuted in 1996, the premise was that content should be divorced from style, allowing the author to concentrate on the former, and the end-user (reader) able to tweak the latter to taste. Comic Sans became popular not too soon afterwards, surely related, perhaps the first real indication that this was in fact a major mistake..

Fast forward almost two decades to 2014, when Christoper 'vjeux' Chedeau's gave a well received 2014 talk on CSS in Javascript, attempting to fix many of the problems that resulted from this early decision. A proliferation of libraries followed, aimed at improving the styling of component based applications and once again tying together content and style more closely together.

By offering a locally scoped alternative to globally scoped traditional CSS (something that CSS Modules also give us), we can do away with many of the naming conventions (BEM, SMACSS etc) used to combat the problems caused by global scoping, such as the accidental overriding of styles. This results in a significantly simplified developer experience, being able to ensure that individual changes do not have unexpected flow-on effects..

CSS in Javascript libraries largely fall into two categories - those that make use of tagged template literals, such as Styled Components, and those that use built-in Javascript data structures such as Glamorous and Styletron.

There is a fair amount of debate within the Badger camp as to which, or indeed any of these options are advantageous. Reflecting this, the current state of CSS in Javascript is extremely fragmented, with a total of 59 libraries currently listed on Michele Bertoli's useful comparison of options available to React users, a clear winner yet to emerge.

This perhaps explains why Emotion, despite being relatively recent addition, has rapidly picked up stars on github, offering the option of both object and css string approaches, and wrapper around it's core functionality with a Styled Component-like feel should you wish to use it use it over the primary 'css' function.

Given this flexibility, if you are looking to give CSS in JS a go, Emotion would probably be a good place to start in 2018, allowing you to decide for yourself which approach to take. ■

← Back to contents

# Interested to find out more?

Founded in 2010 by Cain, Dave and Stu, Red Badger is an independently owned digital consultancy. On a mission to make things better for our clients, we're digital transformation experts who innovate and deliver. We aim to choose the right tech for the job and help clients navigate the open source revolution to increase their speed to market, drive efficiency and deliver customer value faster.

## Our previous roundtables

2017 June

2016

2015 June

2015 March

2013

2012


www.red-badger.com

Say hi to us:
hello@red-badger.com