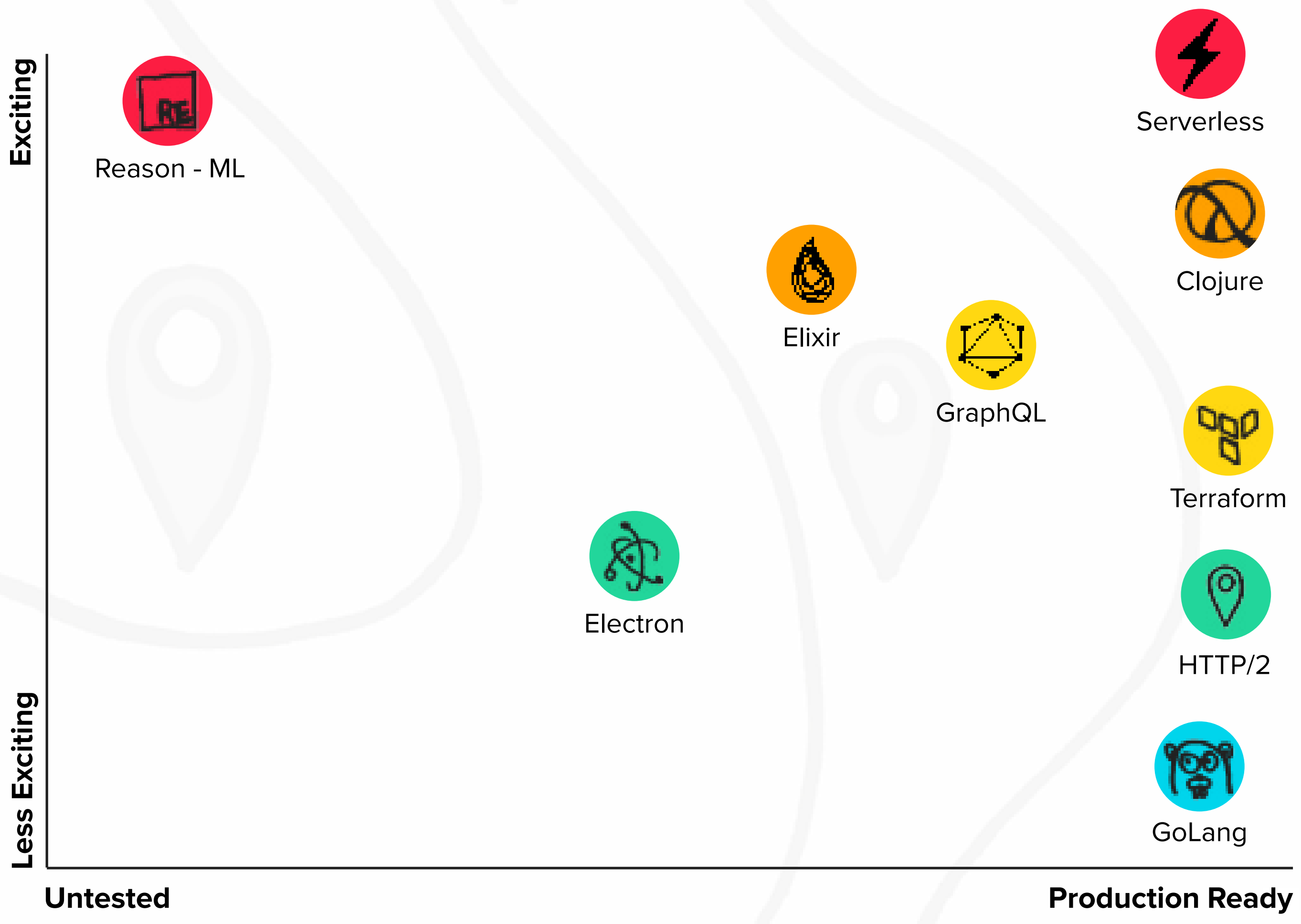
RED BADGER

The tech landscape

Two times a year we get together as a team of developers and testers to take a look at what's happening in tech and how it's implemented. We examine what we've been using, what others are using, and what's on the horizon.

There are two filters – how exciting the tech is to us and how production ready it is. The purpose of this is to ensure that we're aware of what's happening so we can identify potentially disruptive tech, and do it early. It also helps us set the priorities for the tech and techniques we already use, advocate for, and introduce our clients to. This strategic approach leads to tech choices that have a robust business case and deliver clear, measurable benefits.



Technology overview

Elixir

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...

Clojure

Clojure is a functional, general purpose programming language which brings the wisdom of the Lisp language together with an extremely pragmatic approach to problem solving...

GraphQL - NV

GraphQL is a query language for your web APIs. It isn't tied to any specific database or storage-engine and is instead backed by your existing code and data...

GoLang - NV

Go is a snappy language that works incredibly well for fast moving projects due to an emphasis on simple, clear and incredibly pragmatic code that is easy to understand and maintain...

Terraform - NV

Terraform is an 'infrastructure-as-code' tool built by Hashicorp. This means it describes standard infrastructures (servers, load balancers, security groups etc.) as code...

Serverless - NV

Serverless architecture removes the need for the traditional "always on" infrastructure sitting behind an application. With a serverless infrastructure, you only need to care about what events trigger your code to run in the cloud, and not worry about the servers that it runs on...

Reason ML - NV

Reason is a new interface to the functional programming language OCaml, which is a robust, functional type system, often used for high performance server side applications where correctness is especially important...

Electron - NV

Electron (formerly known as Atom Shell) is an open source framework built by GitHub which allows you to write cross-platform desktop applications using JavaScript, HTML and CSS...

HTTP/2 - NV

HTTP/2 is a major upgrade to the HTTP protocol currently used to serve all content on the web. It promises the ability to run far faster websites, providing powerful features to improve performance across the board...

Technology in detail

Elixir

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 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).

Used in the right way, this language is a great way to build impressive tech, fast, saving time and money without sacrificing on quality.

Clojure

Clojure is a functional, general purpose programming language which brings the wisdom of the Lisp language together with an extremely pragmatic approach to problem solving. It relies on battle tested language runtimes such as the Java Virtual Machine, .NET and JavaScript engines in order to provide powerful and simple tools and allows developers and organisations to not be tied to a specific platform or vendor.

Clojure makes developers focus on the business requirements of a feature rather than worrying about the implementation details of the solution. It achieves this by providing a small yet focused and powerful API, ensuring the development is faster and the software is more reliable.

Another great benefit is that Clojure has interop with the host runtime (e.g. JVM or JavaScript), allowing for easy integration with existing architectures. So, Clojure can leverage the rich ecosystem of the JVM and Java applications can leverage Clojure modules. This is equally true for Javascript engines (browser, node.js, etc). Clojure also provides the most powerful concurrency primitives, this makes it one of the best candidates for enterprise high throughput data analysis.

While developing for concurrency brings complexity for infrastructure and system testing Clojure's ability to design provides faster applications and easier formation of clusters.

GraphQL - NV

GraphQL is a query language for your web APIs. It isn’t tied to any specific database or storage-engine and is instead backed by your existing code and data. So, unlike RESTful APIs, GraphQL lets API consumers specify exactly what data they need and fetch all the resources they require in a single request; vastly reducing latency cost on slow networks. What used to mean tens or even hundreds of REST endpoints to satisfy multiple different clients or even multiple versions of a single client application (e.g. mobile apps) can be supported by a single GraphQL endpoint. This endpoint becomes a perfect separation point between your backend services and the client applications and can even serve as an API gateway,providing authorisation, caching, etc.

GraphQL schemas can gradually be evolved by adding capabilities and deprecating old ones. This completely decouples backend (services and capabilities) and front-end (websites and applications) and allows them to evolve independently at a much higher pace. It also means older clients are supported for as long as they need to be without a huge technical debt. While GraphQL makes data fetching very easy and performant, it is strongly geared towards fetching and the mechanism for writing data through mutations can become tedious. On the other hand, GraphQL forces you to think about and enumerate existing state transition. You also have to consider what part of your data is affected by them.

GraphQL was open sourced over a year ago by Facebook after being used internally for almost four years. Since then, implementations in several languages have been built and it was adopted by several big API providers, including GitHub, Pinterest or Shopify. There are also alternative technologies in the same space, for example Falcor, built by Netflix, which builds similar capabilities with a REST compatible interface.

GoLang - NV

The Programming language choice you use in a project is fundamental as its design decisions extend to the tools and libraries that get built on top of it. One we love at Red Badger is Google's "Go" which was developed to address some of the issues faced by big organisations with large-scale Java and C++ codebases.

It's a snappy language that works incredibly well for fast moving projects due to an emphasis on simple, clear and incredibly pragmatic code that is easy to understand and maintain.

Go is also a response to growing trends to reduce the mental overhead (decision making time) that programming language require, for instance through coding style enforcement by using "go fmt". It's incredible how much time this saves an individual engineer, let alone a team who would otherwise have to come to an agreement on their own.

Go's syntax is simple and would be familiar to Java and C/C++ programmers, yet it contains some big wins, such as a well designed set of APIs that make programming super productive and code consistent in style and built in features for concurrency. This means that programmers can hop from one codebase to another and see familiar implementations of common tasks such as queueing network requests and buffering data, reducing the onboarding time required for engineers new to the codebase

While some developers find the language design is ruthlessly simple, that very same simplicity affords programming newcomers a short learning curve and can ensure productivity after even a day's reading.

Terraform - NV

Terraform is an ‘infrastructure-as-code’ tool built by Hashicorp. This means it describes standard infrastructures (servers, load balancers, security groups etc.) as code. So the systems and devices that are used to run your software can be treated as if they themselves are software, and you can write code to manage them and help automate process. It can be used across a wide variety of cloud providers such as AWS, Azure and Google Cloud. Terraform scripts are therefore more maintainable and vendor agnostic than vendor specific counterparts such as Cloud Formation.

Terraform is also an enabler of a core principle of DevOps, “immutable infrastructure”. This means building infrastructure that is fixed so that when updates need to be made, components are replaced rather than altered in order to ensure that changes do not affect the integrity of the existing deployment. The benefits of this include: lower IT complexity and failures; improved security and easier troubleshooting.

Terraform creates and updates infrastructure in parallel, making it fast enough to keep developers highly productive whilst using it. Infrastructure can be built and tested iteratively, and the impact of all changes can be reviewed in detail before they are executed, which all contributes to projects being delivered by Red Badger on time and within budget.

Serverless - NV

Serverless architecture removes the need for the traditional “always on” infrastructure sitting behind an application. With a serverless infrastructure, you only need to care about what events trigger your code to run in the cloud, and not worry about the servers that it runs on. You also save a huge amount of time and money not having to worry about system administration and server uptime. In particular, Serverless can be incredibly cost effective in that you only pay for processing time, unlike the “always on” costs of physical server vendors.

Serverless functions can scale instantly and infinitely to your needs as events take place. Functions are triggered by events of your choosing, like API calls or database changes.If no events are taking place, and no code is running, you pay nothing. This way, serverless options can give you the best of both worlds, providing performance when needed whilst reducing the cost of infrastructure management.

Running code on Serverless also brings a revolution to your infrastructure with frameworks like serverless.js, claudia.js and Apex making deployments even easier. Provision pipelines of environments and deploy to them instantly without having to worry about uptime. Those frameworks also allows you to focus on the code of your application and forget about the infrastructure it runs on, ensuring a smoother, faster build and even more cost efficiency.

Reason ML - NV

Reason is a new interface to the functional programming language OCaml, which is often used for high performance server side applications where correctness is especially important.

Reason provides an expressive dialect of the OCaml language (with a syntax similar to JavaScript) and a modern build toolchain making it easier to learn and contribute to an OCaml codebase. The main developer experience improvements are: an approachable syntax; powerful automatic source code formatting; can be adopted incrementally with JavaScript/C interop via BuckleScript; ahead of time compilation to assembly - without a language level VM and it is able to rapidly develop and share projects.

While Reason is not yet mature enough to recommend for production apps, it is evolving in the open as a community collaboration, so we feel that it is a language to keep a close eye on and one that could bring OCaml into the mainstream.

Electron - NV

Electron (formerly known as Atom Shell) is an open source framework built by GitHub which allows you to write cross-platform desktop applications using JavaScript, HTML and CSS. It does this through the use of front and back end components originally developed for web applications: Node.js runtime for the back end and the Chromium browser engine for the front. This forms a powerful combination that allows developers to use all of the libraries available to Node.js and exposes all the web standards supported by Chromium. Some other great benefits include: ability to produce native applications with web standards; good integration with MacOS, Windows and Linux; working with the same tools used for web development.

Electron also allows desktop applications to share logic and user interfaces among different operating systems by exposing a common set of APIs. This allow developers to create the application on one platform and have it run on many without having to change code.

Examples of desktop applications built using Electron and web technologies are the Slack desktop client, Github Atom editor and Microsoft VS Code.

HTTP/2 - NV

HTTP/2 is a major upgrade to the HTTP protocol currently used to serve all content on the web and was developed from the earlier, experimental SPDY protocol, that originally came from Google.

The main benefit of HTTP/2 is that it promises the ability to run far faster websites, providing powerful features to improve performance across the board, and functioning particularly well on high latency mobile connections.

Features such as multiplexing (using one connection to fetch all resources for a page) as well as the ability to “push” resources to the client rather than rely on them to be requested, means that providing a great customer experience will become easier than it is today. Especially on mobile.

However, designing applications to take advantage of this new protocol whilst maintaining backwards compatibility is a new challenge, one which requires rethinking the current best practice that could currently be detrimental under HTTP/2, for instance bundling and domain sharding. Adoption of HTTP/2 also requires taking up encryption across the board, which could have cost and performance implications.

Red Badger continues to explore this technology to ensure best fit and application in terms of project work.

Techniques overview

TDD

Test Driven Development (TDD) encourages good development practice down to the lowest level. By following the three laws of TDD, each unit of code is built in a short feedback loop driven by tests, which ensures focus on the end goal...

Feature Teams vs Platform Teams

Technologies such as React Native and Microservice architectures are reducing the need for dedicated platform teams. Where separate Web, Android and iOS teams may have once been needed, a single feature team now have the tools at their disposal to tackle a cross-platform project...

Mono Repo

For highly modular codebases such as microservice architectures, a trend is emerging in which the entire codebase of a system is stored in a single monolithic repository, a monorepo... euismod semper. Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Donec id elit non mi porta gravida at eget metus.

Techniques in detail

TDD

Test Driven Development (TDD) encourages good development practice down to the lowest level. By following the three laws of TDD, outlined here, each unit of code is built in a short feedback loop driven by tests, which ensures focus on the end goal. By retaining such focus, wasteful code is eliminated, as everything written serves a clear purpose. The practice of writing just enough code to pass a test, before refactoring to improve quality and readability, results in a cleaner, better structured, less coupled codebase and unparalleled unit test coverage. These small iterations are a powerful method of increasing coding speed, as solutions are less likely to be over-engineered and difficult to change.

TDD requires both a skilled development team, and a willingness to invest in a learning curve. The benefits outlined above increment over time, so the patience of those who master this technique is rewarded.

Frameworks such as Mocha.js and Jasmine are allowing developers to write unit tests in a more simplistic, readable format than ever. When these tests are well constructed and maintained they can serve a second purpose as executable specification and living documentation. An Agile team can utilise the outcome to collaborate and scope out the higher level of tests needed to form a well-structured testing pyramid. For this reason, TDD can be the foundation of an aligned development and testing strategy on a project.

Feature Teams vs Platform Teams

Technologies such as React Native and Microservice architectures are reducing the need for dedicated platform teams. Where separate Web, Android and iOS teams may have once been needed, a single feature team now have the tools at their disposal to tackle a cross-platform project. Consequently, the waste and duplication of effort that can arise from multiple teams working on variants of the same product can be avoided by adopting a feature team model.

A Feature Team takes full ownership of product development, with little reliance on other departments. Tools such as Docker and a variety of continuous integration solutions are taking away the dependencies a team may have once had on dedicated departments or individuals. Although specialist roles still exist within a feature team to an extent, individuals are encouraged to work across disciplines and platforms, collaborating and learning any skills necessary to reach their end goal. Barriers that once warranted specialisms and dedicated teams are being lowered with each new emerging technology, and that's a trend that is set to continue.

To experience the full benefits of a Feature Team, each one should be long-lived, multi-disciplined and afforded the stability needed to allow for continuous learning and improvement. A move towards Feature Teams is a long-term investment in your delivery process that aligns structure to methodology and technological advancements, and drives consistency in cross-channel products.

Mono Repo

For highly modular codebases such as microservice architectures, a trend is emerging in which the entire codebase of a system is stored in a single monolithic repository, a monorepo. A monrepo is all about moving fast and getting things done more efficiently to increase developer productivity and is an approach used successfully by global companies with massive codebases like Google or Facebook. A particular benefit here is that monorepo is compatible with the ebb and flow of large organisations and large software projects. Teams come and go, so if a business is maintaining separate repositories it's going be having a hard time mapping the ever-changing organisational topology.

A monorepo also makes it vastly easier to manage dependencies between your system's components, while still building, deploying and scaling them separately. Instead of managing an ever growing set of versioned build artifacts, all deployable applications and services are built from source, from a single version snapshot of the state of your project. Whilst growing a monorepo beyond a certain scale could prove a challenge, the benefits ensure that boundaries between components of the system are easier to change; large scale refactorings across boundaries become possible; code reuse is easier and automated tests can be run not only for the changed component but also for everything that depends on it. This makes the feedback loop much faster and removes the time wasted bumping dependency versions, allowing your delivery teams to focus on work that brings customer value instead.

While there are existing build systems (e.g. Google's Bazel or Facebook's Buck) geared towards large highly modular codebases, most off-the-shelf Continuous Integration software isn't set up for monolithic repositories and will require some initial investments in configuration and tooling.

Want to know more?

Red Badger was born and developed by Cain, Stu and Dave in 2010. They had a shared ambition to create a business that turned the traditional consulting model on its head: no sales team; no RFP processes, just great people doing great work that focused on quality, value and collaboration. So they left their jobs to turn the dream into reality and set up Red Badger in Cain's bedroom.

www.red-badger.com

hello@red-badger.com

+44 20 3567 0555