Microservices

a definition of this new architectural term

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of <mark>independently deployable services</mark>. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

25 March 2014

James Lewis

James Lewis is a Principal Consultant at ThoughtWorks and member of the Technology Advisory Board. James' interest in building applications out of small collaborating services stems from a background in integrating enterprise systems at scale. He's built a number of systems using microservices and has been an active participant in the growing community for a couple of years.

Martin Fowler

Martin Fowler is an author, speaker, and general loud-mouth on software development. He's long been puzzled by the problem of how to componentize software systems, having heard more vague claims than he's happy with. He hopes that microservices will live up to the early promise its advocates have found.

"Microservices" - yet another new term on the crowded streets of software architecture. Although our natural inclination is to pass such things by with a contemptuous glance, this bit of terminology describes a style of software systems that we are finding more and more appealing. We've seen many projects use this style in the last few years, and results so far have been positive, so much so that for many of our colleagues this is becoming the default style for building enterprise applications. Sadly, however, there's not much information that outlines what the microservice style is and how to do it.

<mark>In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.</mark> These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

My Microservices Resource Guide provides links to the best articles, videos, books, and podcasts about microservices.

<mark>To start explaining the microservice style it's useful to compare it to the monolithic style</mark>: a monolithic application built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a *monolith* - a single logical executable[2]. Any changes to the system involve building and deploying a new version of the server-side application.

Such a monolithic server is a natural way to approach building such a system. All your logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a

deployment pipeline to ensure that changes are properly tested and deployed into production. You can horizontally scale the monolith by running many instances behind a load-balancer.

==Monolithic applications can be successful, but increasingly people are feeling frustrations with them== - especially as more applications are being deployed to the cloud . Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource.
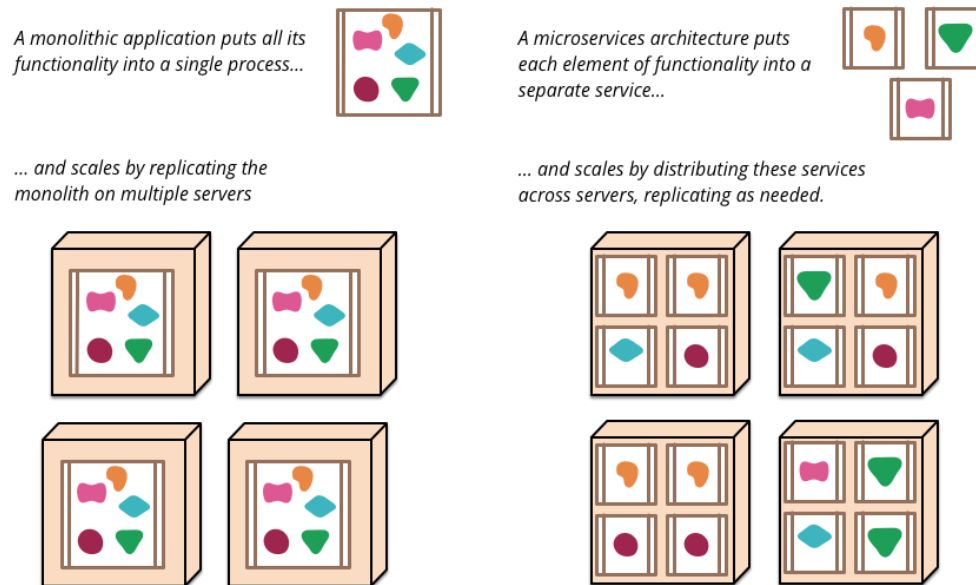

Figure 1: Monoliths and Microservices

==These frustrations have led to the microservice architectural style: building applications as suites of services. As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams== .

We do not claim that the microservice style is novel or innovative, its roots go back at least to the design principles of Unix. But we do think that not enough people consider a microservice architecture and that many software developments would be better off if they used it.

Characteristics of a Microservice Architecture

We cannot say there is a formal definition of the microservices architectural style, but we can attempt to describe what we see as common characteristics for architectures that fit the label. As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics. While we authors have been active members of this rather loose community, our intention is to attempt a description of what we see in our own work and in similar efforts by teams we know of. In particular we are not laying down some definition to conform to.

Componentization via Services

For as long as we've been involved in the software industry, there's been a desire to build systems by plugging together components, much in the way we see things are made in the physical world. During the last couple of decades we've seen considerable progress with large compendiums of common libraries that are part of most language platforms.

When talking about components we run into the difficult definition of what makes a component. Our definition is that a **component** is a unit of software that is independently replaceable and upgradeable.

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define **libraries** as components that are linked into a program and called using in-memory

function calls, while **services** are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. (This is a different concept to that of a service object in many OO programs [3].)

One main reason for using services as components (rather than libraries) is that services are independently deployable. If you have an application [4] that consists of a multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

Another consequence of using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining an explicit Published Interface. Often it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms.

Using services like this does have downsides. Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use. If you need to change the allocation of responsibilities between components, such movements of behavior are harder to do when you're crossing process boundaries.

At a first approximation, we can observe that services map to runtime processes, but that is only a first approximation. A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service.

Organized around Business Capabilities

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law[5] in action.

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*
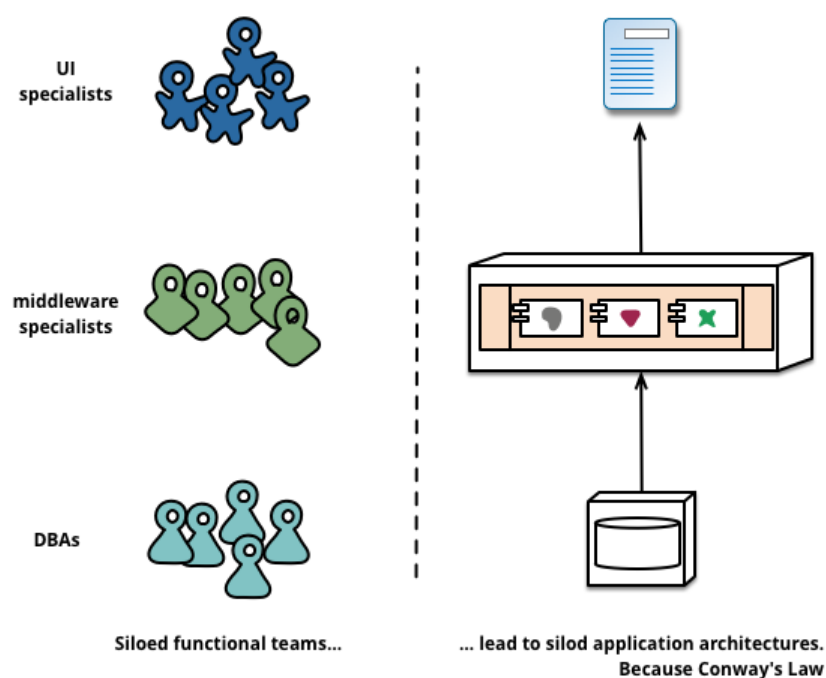>
> *-- Melvin Conway, 1968*

Figure 2: Conway's Law in action

The microservice approach to division is different, splitting up into services organized around **business capability**. Such services take a broad-stack implementation of software for that business area, including user-interface, persistant storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.
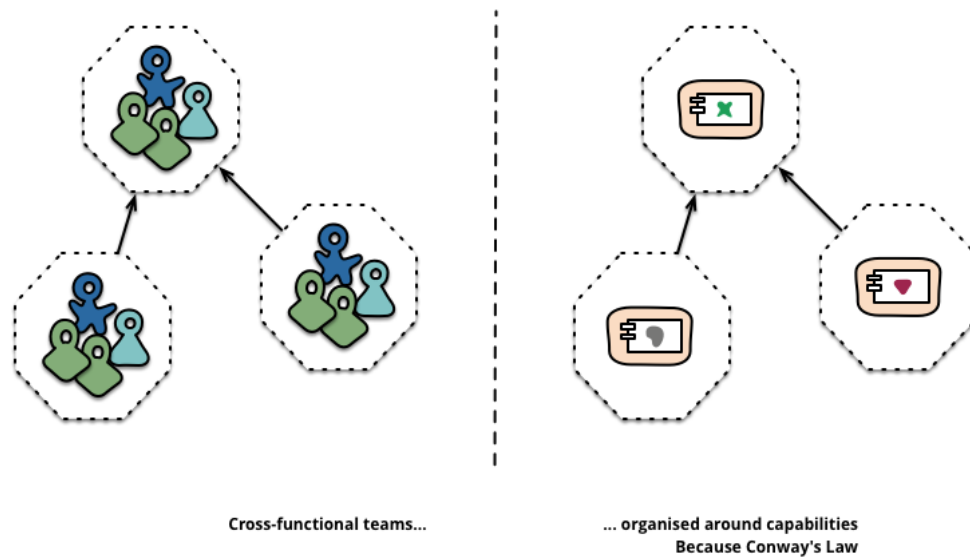

Figure 3: *Service boundaries reinforced by team boundaries*

How big is a microservice?

Although "microservice" has become a popular name for this architectural style, its name does lead to an unfortunate focus on the size of service, and arguments about what constitutes "micro". In our conversations with microservice practitioners, we see a range of sizes of services. The largest sizes reported follow Amazon's notion of the Two Pizza Team (i.e. the whole team can be fed by two pizzas), meaning no more than a dozen people. On the smaller size scale we've seen setups where a team of half-a-dozen would support half-a-dozen services.

This leads to the question of whether there are sufficiently large differences within this size range that the service-per-dozen-people and service-per-person sizes shouldn't be lumped under one microservices label. At the moment we think it's better to group them together, but it's certainly possible that we'll change our mind as we explore this style further.

One company organised in this way is www.comparethemarket.com. Cross functional teams are responsible for building and operating each product and each product is split out into a number of individual services communicating via a message bus.

Large monolithic applications can always be modularized around business capabilities too, although that's not the common case. Certainly we would urge a large team building a monolithic application to divide itself along business lines. The main issue we have seen here, is that they tend to be organised around *too many* contexts. If the monolith spans many of these modular boundaries it can be difficult for individual members of a team to fit them into their short-term memory. Additionally we see that the modular lines require a great deal of discipline to enforce. The necessarily more explicit separation required by service components makes it easier to keep the team boundaries clear.

Products not Projects

Most application development efforts that we see use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime. A common inspiration for this is Amazon's notion of "you build, you run it" where a development team

takes full responsibility for the software in production. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

The product mentality, ties in with the linkage to business capabilities. Rather than looking at the software as a set of functionality to be completed, there is an on-going relationship where the question is how can software assist its users to enhance the business capability.

There's no reason why this same approach can't be taken with monolithic applications, but the smaller granularity of services can make it easier to create the personal relationships between service developers and their users.

Smart endpoints and dumb pipes

When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.

Microservices and SOA

When we've talked about microservices a common question is whether this is just Service Oriented Architecture (SOA) that we saw a decade ago. There is merit to this point, because the microservice style is very similar to what some advocates of SOA have been in favor of. The problem, however, is that SOA means too many different things, and that most of the time that we come across something called "SOA" it's significantly different to the style we're describing here, usually due to a focus on ESBs used to integrate monolithic applications.

In particular we have seen so many botched implementations of service orientation - from the tendency to hide complexity away in ESB's [6], to failed multi-year initiatives that cost millions and deliver no value, to centralised governance models that actively inhibit change, that it is sometimes difficult to see past these problems.

Certainly, many of the techniques in use in the microservice community have grown from the experiences of developers integrating services in large organisations. The Tolerant Reader pattern is an example of this. Efforts to use the web have contributed, using simple protocols is another approach derived from these experiences - a reaction away from central standards that have reached a complexity that is, frankly, breathtaking. (Any time you need an ontology to manage your ontologies you know you are in deep trouble.)

This common manifestation of SOA has led some microservice advocates to reject the SOA label entirely, although others consider microservices to be one form of SOA [7], perhaps *service orientation done right*. Either way, the fact that SOA means such different things means it's valuable to have a term that more crisply defines this architectural style.

The microservice community favours an alternative approach: *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging[8]. The best expression of the first is

> *Be of the web, not behind the web*
>
> -- Ian Robinson

Microservice teams use the principles and protocols that the world wide web (and to a large extent, Unix) is built on. Often used resources can be cached with very little effort on the part of developers or operations folk.

The second approach in common use is messaging over a lightweight message bus. The infrastructure chosen is typically dumb (dumb as in acts as a message router only) - simple implementations such as RabbitMQ or ZeroMQ don't do much

more than provide a reliable asynchronous fabric - the smarts still live in the end points that are producing and consuming messages; in the services.

In a monolith, the components are executing in-process and communication between them is via either method invocation or function call. The biggest issue in changing a monolith into microservices lies in changing the communication pattern. A naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well. Instead you need to replace the fine-grained communication with a coarser -grained approach.

Decentralized Governance

One of the consequences of centralised governance is the tendency to standardise on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer. We prefer using the right tool for the job and while monolithic applications can take advantage of different languages to a certain extent, it isn't that common.

Splitting the monolith's components out into services we have a choice when building each of them. You want to use Node.js to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine. You want to swap in a different flavour of database that better suits the read behaviour of one component? We have the technology to rebuild him.

Of course, just because you *can* do something, doesn't mean you *should* - but partitioning your system in this way means you have the option.

Teams building microservices prefer a different approach to standards too. Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing. These tools are usually harvested from implementations and shared with a wider group, sometimes, but not exclusively using an internal open source model. Now that git and github have become the de facto version control system of choice, open source practices are becoming more and more common in-house .

Netflix is a good example of an organisation that follows this philosophy. Sharing useful and, above all, battle-tested code as libraries encourages other developers to solve similar problems in similar ways yet leaves the door open to picking a different approach if required. Shared libraries tend to be focused on common problems of data storage, inter-process communication and as we discuss further below, infrastructure automation.

For the microservice community, overheads are particularly unattractive. That isn't to say that the community doesn't value service contracts. Quite the opposite, since there tend to be many more of them. It's just that they are looking at different ways of managing those contracts. Patterns like Tolerant Reader and Consumer-Driven Contracts are often applied to microservices. These aid service contracts in evolving independently. Executing consumer driven contracts as part of your build increases confidence and provides fast feedback on whether your services are functioning. Indeed we know of a team in Australia who drive the build of new services with consumer driven contracts. They use simple tools that allow them to define the contract for a service. This becomes part of the automated build before code for the new service is even written. The service is then built out only to the point where it satisfies the contract - an elegant approach to avoid the 'YAGNI'[9] dilemma when building new software. These techniques and the tooling growing up around them, limit the need for central contract management by decreasing the temporal coupling between services.

Many languages, many options

The growth of JVM as a platform is just the latest example of mixing languages within a common platform. It's been common practice to shell-out to a higher level language to take advantage of higher level abstractions for decades. As is dropping down to the metal and writing performance sensitive code in a lower level one. However, many monoliths don't need this level of performance optimisation nor are DSL's and higher level abstractions that common (to our dismay). Instead monoliths are usually single language and the tendency is to limit the number of technologies in use [10].

Perhaps the apogee of decentralised governance is the build it / run it ethos popularised by Amazon. Teams are responsible for all aspects of the software they build including operating the software 24/7. Devolution of this level of

responsibility is definitely not the norm but we do see more and more companies pushing responsibility to the development teams. Netflix is another organisation that has adopted this ethos[11]. Being woken up at 3am every night by your pager is certainly a powerful incentive to focus on quality when writing your code. These ideas are about as far away from the traditional centralized governance model as it is possible to be.

Decentralized Data Management

Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics.

Battle-tested standards and enforced standards

It's a bit of a dichotomy that microservice teams tend to eschew the kind of rigid enforced standards laid down by enterprise architecture groups but will happily use and even evangelise the use of open standards such as HTTP, ATOM and other microformats.

The key difference is how the standards are developed and how they are enforced. Standards managed by groups such as the IETF only *become* standards when there are several live implementations of them in the wider world and which often grow from successful open-source projects.

These standards are a world apart from many in a corporate world, which are often developed by groups that have little recent programming experience or overly influenced by vendors.

This issue is common between applications, but can also occur *within* applications, particular when that application is divided into separate components. A useful way of thinking about this is the Domain-Driven Design notion of Bounded Context. DDD divides a complex domain up into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify, and as we describe in the section on business capabilities, reinforce the separations.

As well as decentralizing decisions about conceptual models, microservices also decentralize data storage decisions. While monolithic applications prefer a single logical database for persistant data, enterprises often prefer a single database across a range of applications - many of these decisions driven through vendor's commercial models around licensing. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence. You can use polyglot persistence in a monolith, but it appears more frequently with microservices.

monolith - single database   microservices - application databases
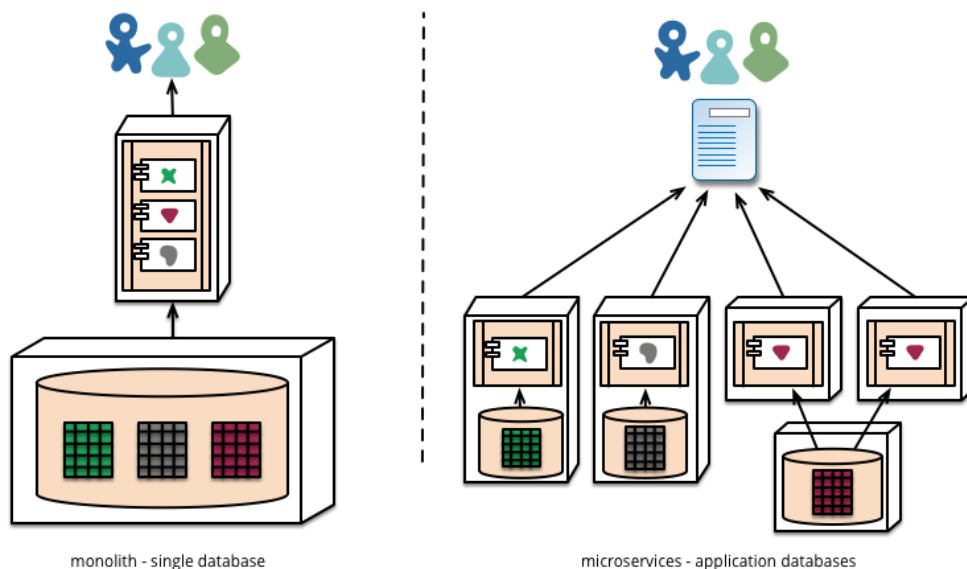
Decentralizing responsibility for data across microservices has implications for managing updates. The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths.

Using transactions like this helps with consistency, but imposes significant temporal coupling, which is problematic across multiple services. Distributed transactions are notoriously difficult to implement and as a consequence microservice architectures emphasize transactionless coordination between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

Choosing to manage inconsistencies in this way is a new challenge for many development teams, but it is one that often matches business practice. Often businesses handle a degree of inconsistency in order to respond quickly to demand, while having some kind of reversal process to deal with mistakes. The trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency.

Infrastructure Automation

Infrastructure automation techniques have evolved enormously over the last few years - the evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices.

Many of the products or systems being build with microservices are being built by teams with extensive experience of Continuous Delivery and it's precursor, Continuous Integration. Teams building software this way make extensive use of infrastructure automation techniques. This is illustrated in the build pipeline shown below.
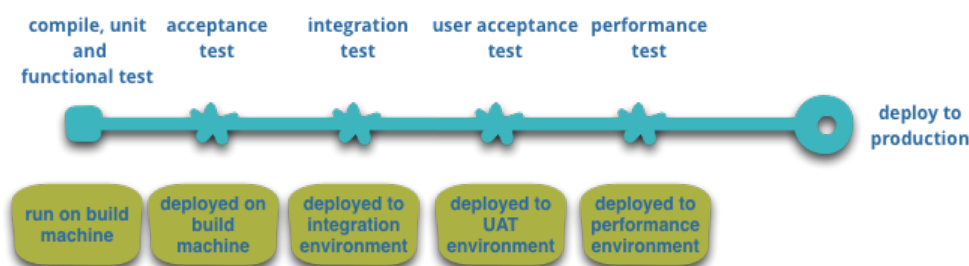

Figure 5: basic build pipeline

Since this isn't an article on Continuous Delivery we will call attention to just a couple of key features here. We want as much confidence as possible that our software is working, so we run lots of **automated tests**. Promotion of working software 'up' the pipeline means we **automate deployment** to each new environment.

Make it easy to do the right thing

One side effect we have found of increased automation as a consequence of continuous delivery and deployment is the creation of useful tools to help developers and operations folk. Tooling for creating artefacts, managing codebases, standing up simple services or for adding standard monitoring and logging are pretty common now. The best example on the web is probably Netflix's set of open source tools, but there are others including Dropwizard which we have used extensively.

A monolithic application will be built, tested and pushed through these environments quite happlily. It turns out that once you have invested in automating the path to production for a monolith, then deploying *more* applications doesn't seem so scary any more. Remember, one of the aims of CD is to make deployment boring, so whether its one or three applications, as long as its still boring it doesn't matter[12].

Another area where we see teams using extensive infrastructure automation is when managing microservices in production. In contrast to our assertion above that as long as deployment is boring there isn't that much difference between monoliths and microservices, the operational landscape for each can be strikingly different.
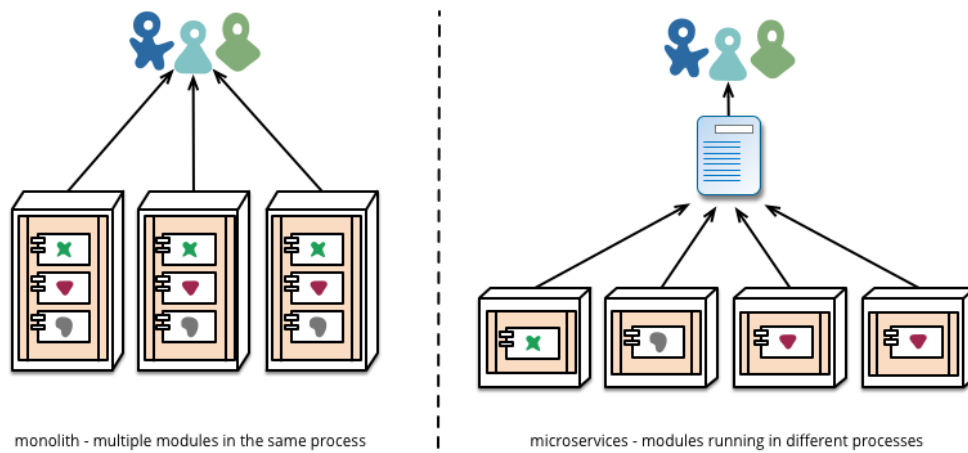
Figure 6: Module deployment often differs

Design for failure

A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it. The consequence is that microservice teams constantly reflect on how service failures affect the user experience. Netflix's Simian Army induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

The circuit breaker and production ready code

Circuit Breaker appears in Release It! alongside other patterns such as Bulkhead and Timeout. Implemented together, these patterns are crucially important when building communicating applications. This Netflix blog entry does a great job of explaining their application of them.

This kind of automated testing in production would be enough to give most operation groups the kind of shivers usually preceding a week off work. This isn't to say that monolithic architectural styles aren't capable of sophisticated monitoring setups - it's just less common in our experience.

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

This is particularly important to a microservices architecture because the microservice preference towards choreography and event collaboration leads to emergent behavior. While many pundits praise the value of serendipitous emergence, the truth is that emergent behavior can sometimes be a bad thing. Monitoring is vital to spot bad emergent behavior quickly so it can be fixed.

Synchronous calls considered harmful

Any time you have a number of synchronous calls between services you will encounter the multiplicative effect of downtime. Simply, this is when the downtime of your system becomes the product of the downtimes of the individual components. You face a choice, making your calls asynchronous or managing the downtime. At www.guardian.co.uk they have implemented a simple rule on the new platform - one synchronous call per user request while at Netflix, their platform API redesign has built asynchronicity into the API fabric.

Monoliths can be built to be as transparent as a microservice - in fact, they should be. The difference is that you absolutely need to know when services running in different processes are disconnected. With libraries within the same process this kind of transparency is less likely to be useful.

Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples we often encounter in the wild.

Evolutionary Design

Microservice practitioners, usually have come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change. Change control doesn't necessarily mean change reduction - with the right attitudes and tools you can make frequent, fast, and well-controlled changes to software.

Whenever you try to break a software system into components, you're faced with the decision of how to divide up the pieces - what are the principles on which we decide to slice up our application? The key property of a component is the notion of independent replacement and upgradeability[13] - which implies we look for points where we can imagine rewriting a component without affecting its collaborators. Indeed many microservice groups take this further by explicitly expecting many services to be scrapped rather than evolved in the longer term.

The Guardian website is a good example of an application that was designed and built as a monolith, but has been evolving in a microservice direction. The monolith still is the core of the website, but they prefer to add new features by building microservices that use the monolith's API. This approach is particularly handy for features that are inherently temporary, such as specialized pages to handle a sporting event. Such a part of the website can quickly be put together using rapid development languages, and removed once the event is over. We've seen similar approaches at a financial institution where new services are added for a market opportunity and discarded after a few months or even weeks.

This emphasis on replaceability is a special case of a more general principle of modular design, which is to drive modularity through the pattern of change [14]. You want to keep things that change at the same time in the same module. Parts of a system that change rarely should be in different services to those that are currently undergoing lots of churn. If you find yourself repeatedly changing two services together, that's a sign that they should be merged.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process. The downside is that you have to worry about changes to one service breaking its consumers. The traditional integration approach is to try to deal with this problem using versioning, but the preference in the microservice world is to only use versioning as a last resort. We can avoid a lot of versioning by designing services to be as tolerant as possible to changes in their suppliers.

Are Microservices the Future?

Our main aim in writing this article is to explain the major ideas and principles of microservices. By taking the time to do this we clearly think that the microservices architectural style is an important idea - one worth serious consideration for enterprise applications. We have recently built several systems using the style and know of others who have used and favor this approach.

Microservice Trade-Offs

Many development teams have found the microservices architectural style to be a superior approach to a monolithic architecture. But other teams have found them to be a productivity-sapping burden. Like any architectural style, microservices bring costs and benefits. To make a sensible choice you have to understand these and apply them to your specific context.

by Martin Fowler   **Read more...**

Those we know about who are in some way pioneering the architectural style include Amazon, Netflix, The Guardian, the UK Government Digital Service, realestate.com.au, Forward and comparethemarket.com. The conference circuit in 2013 was full of examples of companies that are moving to something that would class as microservices - including Travis CI. In addition there are plenty of organizations that have long been doing what we would class as microservices, but without ever using the name. (Often this is labelled as SOA - although, as we've said, SOA comes in many contradictory forms. [15])

Despite these positive experiences, however, we aren't arguing that we are certain that microservices are the future direction for software architectures. While our experiences so far are positive compared to monolithic applications, we're conscious of the fact that not enough time has passed for us to make a full judgement.

Often the true consequences of your architectural decisions are only evident several years after you made them. We have seen projects where a good team, with a strong desire for modularity, has built a monolithic architecture that has decayed over the years. Many people believe that such decay is less likely with microservices, since the service boundaries are explicit and hard to patch around. Yet until we see enough systems with enough age, we can't truly assess how microservice architectures mature.

There are certainly reasons why one might expect microservices to mature poorly. In any effort at componentization, success depends on how well the software fits into components. It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries. Moving code is difficult across service boundaries, any interface changes need to be coordinated between participants, layers of backwards compatibility need to be added, and testing is made more complicated.

Another issue is If the components do not compose cleanly, then all you are doing is shifting complexity from inside a component to the connections between components. Not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services.

Finally, there is the factor of team skill. New techniques tend to be adopted by more skillful teams. But a technique that is more effective for a more skillful team isn't necessarily going to work for less skillful teams. We've seen plenty of cases of less skillful teams building messy monolithic architectures, but it takes time to see what happens when this kind of mess occurs with microservices. A poor team will always create a poor system - it's very hard to tell if microservices reduce the mess in this case or make it worse.

One reasonable argument we've heard is that you shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem. (Although this advice isn't ideal, since a good in-process interface is usually not a good service interface.)

So we write this with cautious optimism. So far, we've seen enough about the microservice style to feel that it can be a worthwhile road to tread. We can't say for sure where we'll end up, but one of the challenges of software development is that you can only make decisions based on the imperfect information that you currently have to hand.

Footnotes

1: The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on "microservices" as the most appropriate name. James presented some of these ideas as a case study in March 2012 at 33rd Degree in Krakow in Microservices - Java, the Unix Way as did Fred George about the same time. Adrian Cockcroft at Netflix, describing this approach as "fine grained SOA" was pioneering the style at web scale as were many of the others mentioned in this article - Joe Walnes, Daniel Terhorst-North, Evan Botcher and Graham Tackley.

2: The term monolith has been in use by the Unix community for some time. It appears in The Art of Unix Programming to describe systems that get too big.

3: Many object-oriented designers, including ourselves, use the term service object in the Domain-Driven Design sense for an object that carries out a significant process that isn't tied to an entity. This is a different concept to how we're using "service" in this article. Sadly the term service has both meanings and we have to live with the polyseme.

4: We consider an application to be a social construction that binds together a code base, group of functionality, and body of funding.

5: The original paper can be found on Melvin Conway's website here

6: We can't resist mentioning Jim Webber's statement that ESB stands for "Egregious Spaghetti Box".

7: Netflix makes the link explicit - until recently referring to their architectural style as fine-grained SOA.

8: At extremes of scale, organisations often move to binary protocols - protobufs for example. Systems using these still exhibit the characteristic of smart endpoints, dumb pipes - and trade off *transparency* for scale. Most web properties and certainly the vast majority of enterprises don't need to make this tradeoff - transparency can be a big win.

9: "YAGNI" or "You Aren't Going To Need It" is an XP principle and exhortation to not add features until you know you need them.

10: It's a little disengenuous of us to claim that monoliths are single language - in order to build systems on todays web, you probably need to know JavaScript and XHTML, CSS, your server side language of choice, SQL and an ORM dialect. Hardly single language, but you know what we mean.

11: Adrian Cockcroft specifically mentions "developer self-service" and "Developers run what they wrote"(sic) in this excellent presentation delivered at Flowcon in November, 2013.

12: We are being a little disengenuous here. Obviously deploying more services, in more complex topologies is more difficult than deploying a single monolith. Fortunately, patterns reduce this complexity - investment in tooling is still a must though.

13: In fact, Daniel Terhorst-North refers to this style as *Replaceable Component Architecture* rather than microservices. Since this seems to talk to a subset of the characteristics we prefer the latter.

14: Kent Beck highlights this as one his design principles in Implementation Patterns.

15: And SOA is hardly the root of this history. I remember people saying "we've been doing this for years" when the SOA term appeared at the beginning of the century. One argument was that this style sees its roots as the way COBOL programs communicated via data files in the earliest days of enterprise computing. In another direction, one could argue that microservices are the same thing as the Erlang programming model, but applied to an enterprise application context.

References

While this is not an exhaustive list, there are a number of sources that practitioners have drawn inspiration from or which espouse a similar philosophy to that described in this article.

Blogs and online articles

- Clemens Vasters' blog on cloud at microsoft
- David Morgantini's introduction to the topic on his blog
- 12 factor apps from Heroku
- UK Government Digital Service design principles
- Jimmy Nilsson's blogand article on infoq about Cloud Chunk Computing
- Alistair Cockburn on Hexagonal architectures

Books

- Release it
- Rest in practice
- Web API Design (free ebook). Brian Mulloy, Apigee.
- Enterprise Integration Patterns
- Art of unix programming
- Growing Object Oriented Software, Guided by Tests
- The Modern Firm: Organizational Design for Performance and Growth
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
- Domain-Driven Design: Tackling Complexity in the Heart of Software

Presentations

- [Architecture without Architects](#). Erik Doernenburg.
- [Does my bus look big in this?](#). Jim Webber and Martin Fowler, QCon 2008
- [Guerilla SOA](#). Jim Webber, 2006
- [Patterns of Effective Delivery](#). Daniel Terhorst-North, 2011.
- [Adrian Cockcroft's slideshare channel](#).
- [Hydras and Hypermedia](#). Ian Robinson, JavaZone 2010
- [Justice will take a million intricate moves](#). Leonard Richardson, Qcon 2008.
- [Java, the UNIX way](#). James Lewis, JavaZone 2012
- [Micro services architecture](#). Fred George, YOW! 2012
- [Democratising attention data at guardian.co.uk](#). Graham Tackley, GOTO Aarhus 2013
- [Functional Reactive Programming with RxJava](#). Ben Christensen, GOTO Aarhus 2013 (registration required).
- [Breaking the Monolith](#). Stefan Tilkov, May 2012.

Papers

- L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", 1978 http:// research.microsoft.com/en-us/um/people/lamport/pubs/implementation.pdf
- L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", 1982 (available at) http:// www.cs.cornell.edu/courses/cs614/2004sp/papers/lsp82.pdf
- R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 2000 http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- E. A. Brewer, "Towards Robust Distributed Systems", 2000 http://www.cs.berkeley.edu/ ~brewer/cs262b-2004/PODC-keynote.pdf
- E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed", 2012, http:// www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Further Reading

The above list captures the references we used when we originally wrote this article in early 2014. For an up to date list of sources for more information, take a look at the [Microservice Resource Guide](#).