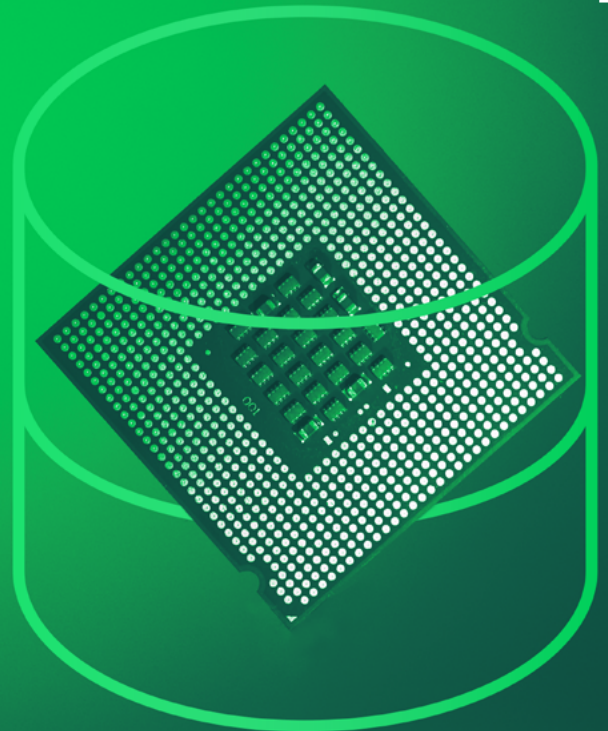
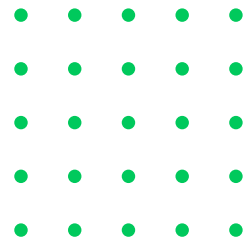


The Object Oriented Guide to Microservices and Serverless Architecture

From Microservices to
Serverless Functions



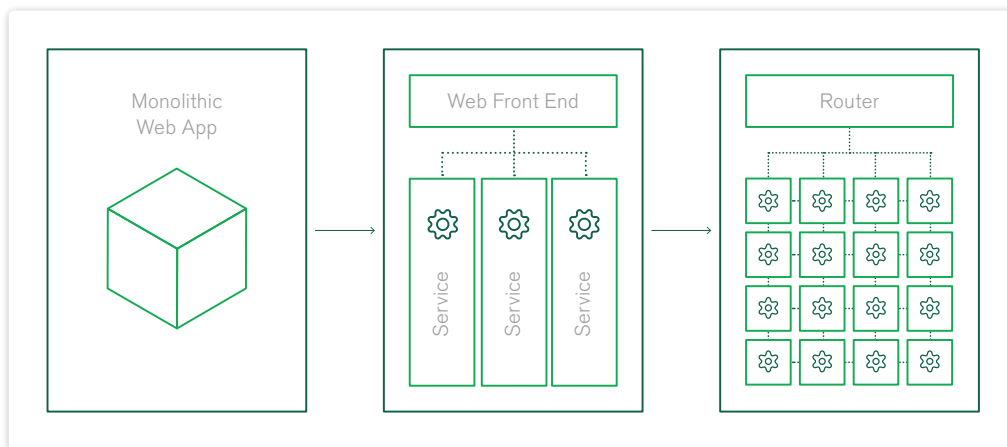


From Microservices to Serverless Functions



Like each generation of the microprocessor, each new generation of server-side architecture seems to challenge physics itself.

From compiled monoliths of yore to three-tiered server constructs, from service-oriented architectures to microservices, we keep shrinking the scope of a single server's responsibility. Our goals at each iteration are the same: simplify the task of writing maintainable, high-quality, high-performance software while growing the complexity of the overall systems in which our applications reside.



From monoliths, to service-oriented architectures, to microservices, and beyond.

In this context, “serverless” architecture (which, of course, still relies on servers, albeit somewhere down below the abstraction level at which application developers think and work) sounds like a departure from this trend. In fact, it is yet another step in shrinking the size of the atoms with which we compose our systems.

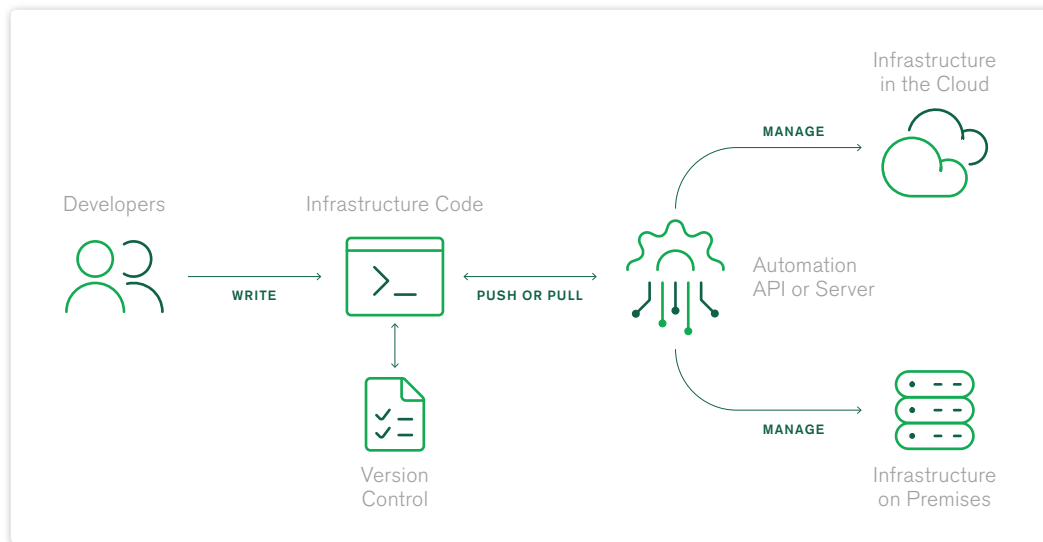
But what does “serverless” really mean? And is it worth it to make the switch to serverless architecture?



Infrastructure as Code: An Object Oriented Approach

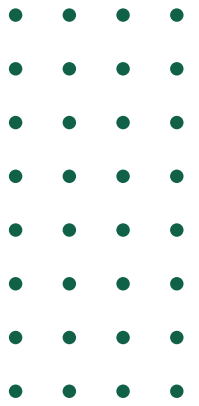
As software infrastructure has grown in complexity, it's started to make sense to apply the thinking used inside of individual software projects to larger distributed software systems.

With service-oriented architectures, however, the problem space shifts to delineating boundaries between services and finding efficient, scalable ways to compose them in order to achieve larger business goals.



Virtualization and automation allow infrastructure to be deployed just like software.

If all this talk of composing systems from discrete components sounds a bit like Object Oriented Programming (OOP) to you, you're far from alone! In this exploration of microservices and the transition to serverless architecture, we found the lessons of Object Oriented Design (OOD) to be useful in talking about what's happening with these technologies.



OOD Principles

Let's look at a few principles of OOD, and how they can be applied as easily to infrastructure as to code:

	OOP	INFRASTRUCTURE
The Single-Responsibility Principle	A class should have one — and only one — reason to change.	A <u>service</u> should have one — and only one — reason to change.
The Interface Segregation Principle	Clients should not be forced to depend upon interfaces that they do not use.	Clients should only have to understand the parts of a service they consume.

These are powerful ideas: that modules (objects, services, or otherwise) should be designed such that each one has a single, clear purpose, and that its interface is as minimal as possible, while fulfilling that purpose.

The result is that modules should be, in essence, as small, and as understandable, as possible, so that the larger system can easily consume them. It's a good way to think about the boundaries of any given microservice: that it should have a single purpose, and expose a simple API that makes sense in context and is as small as possible.





The Ever-Shrinking Service

It's no coincidence that microservice architectures and container-based infrastructures have evolved hand in hand since their beginnings in 2012.

They're symbiotic in a sense; microservices have the smallest application footprint and containers provide a stripped down, reproducible, and atomic deployment. Together they provide a single unit of the application.

This enables complex compositions, independent release cycles, and isolation, all of which allow distributed and loosely coupled teams to iterate rapidly (without, in theory, having to constantly resolve code-level conflicts).

At its core, the ideas behind microservices promise to make life easier for us, providing simple solutions to sophisticated application deployment challenges.



Let's look at how Single-Responsibility and Interface Segregation impact the design of microservices:

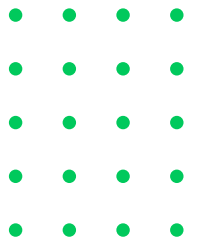
MICROSERVICES

The Single-Responsibility Principle

In order to ensure that a given service does, in fact, have one clear responsibility, it makes sense to shrink the service to its smallest logical surface area. This decreases the mental burden at a system level, making programmers' jobs much simpler. As soon as a service begins to contain functionality that isn't clearly tied to its core purpose, it should be split into a set of smaller services such that the Single Responsibility Principle is followed.

The Interface Segregation Principle

Each service should expose only the API that it needs to expose. Additionally, new features and functions should be considered candidates for new services if they don't make sense in the context of the existing interface.



By thinking about infrastructure in the same way we think about application code, it becomes obvious that microservices are a strong model providing increased transparency, maintainability, and overall enhanced system reliability.

Inevitably, we took those ideas as far as we could and learned a lot about ways in which we shouldn't use containers (the hard way, of course). As an industry, we continue to seek out ways to further simplify the task of developing systems that increasingly revolve around pipelines for processing data at never-before-seen speeds and volumes.



Just as Moore's Law has run its course, so, too, have the size and responsibilities of services. They cannot shrink any further.

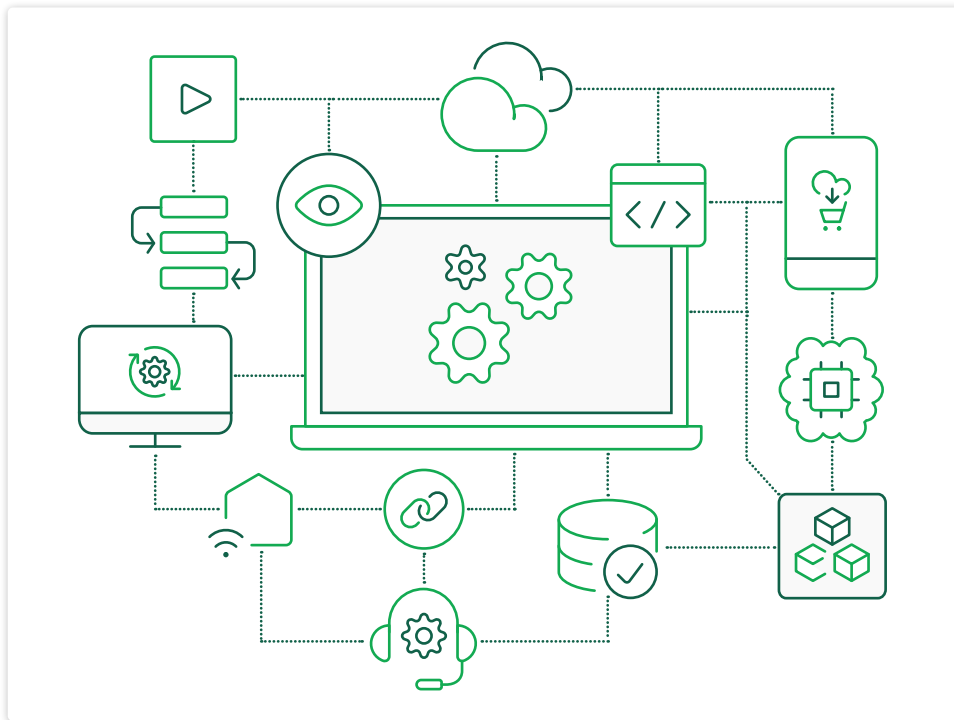


As microservices have grown in popularity across nearly every industry and platform, we're facing diminishing returns. Just as Moore's Law has run its course, so, too, have the size and responsibilities of services. They cannot shrink any further.





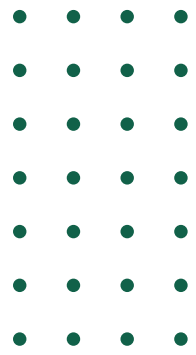
We now find ourselves battling new complexities: fleets of containers, sprawling codebases, and divergent platforms and frameworks:



Too much of a good thing is still too much.

But what if we could reduce much of the complexity that containers (and, indeed, microservices) have brought to our infrastructures while also shrinking and simplifying the problem even further?





The Ever-Growing Weight of Containerization

Increasingly DevOps is seen as a shared set of responsibilities that include developers and traditional operations teams, intentionally blurring the line between them.

The hope is that responsibility can be decentralized enough so that every team is fully independent and that, when necessary, individual services can change and fail gracefully without causing cascading failures. For many teams (namely, ones that may have never worked at the infrastructure layer), this means learning a new context that's both broad and deep. This is far from a trivial task and certainly not a welcome burden.



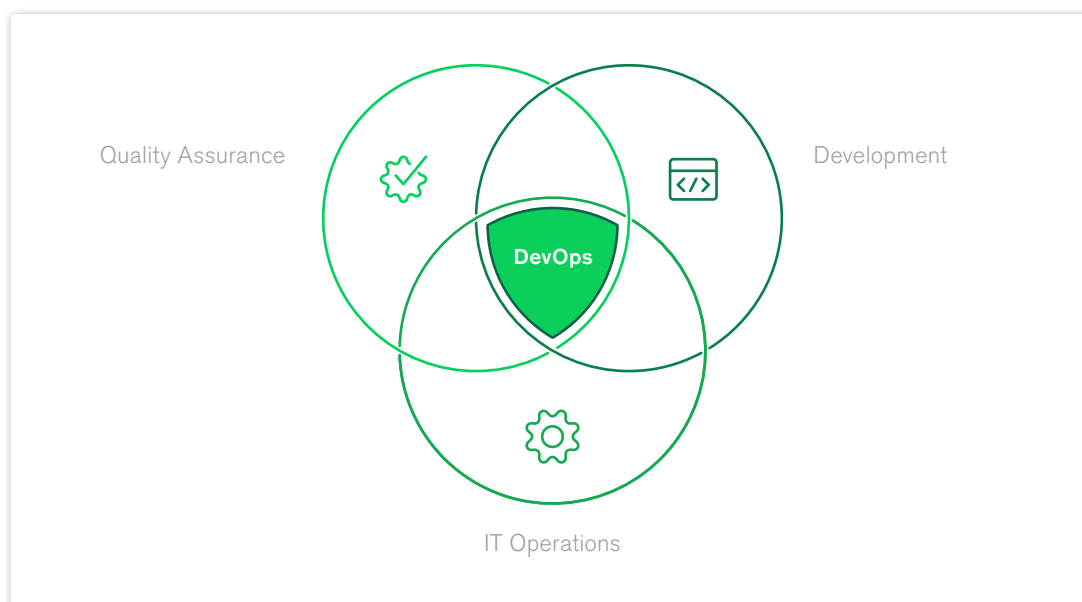
It's clear that we've experienced tremendous change. Still, it's unclear what has really happened.

While some of us enjoy the esoteric ins and outs of container orchestration, from the outside, it's clear that we've experienced tremendous change. Still, it's unclear what has really happened. Some even long for the simplicity of the older monolithic architectures. Defining what we've gained from containers and microservices can be difficult.



For the most part, we understand that we don't install things locally anymore.

However, we're still building things from the Internet – this includes having to download large filesystems for our Docker containers, and the same libraries in Docker that we used to download to our PCs . Docker-compose is used for local development, Terraform modules are used for deploying to our Cloud, Kubernetes for our Docker containers when we're not local, CircleCI Orbs are triggered from GitHub to deploy it all...wasn't this supposed to be easier?

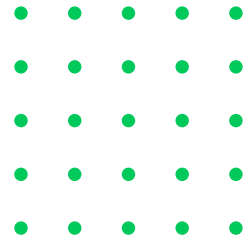


■ Developers are now being asked to perform three separate jobs.

We have more important things to do.

At its core, containers provide their users with clean environments in which they can install anything they want – regardless of operating system, libraries, or programming environments. The additional flexibility gained from this model is profound: creating, modifying, and deploying any single service is delightfully simple. From the point of view of data centers, everyone appears to have agreed that we'll communicate via HTTP from now on and rely on years of traffic shaping wisdom to manage capacity and elastic scaling. It's a model that undeniably works, but it has its drawbacks.





For anyone looking to run code in one of those infrastructures, this meant we had to take the other side of this agreement and provide access to our work via an HTTP interface.

One way to look at what that means: a computer that can do anything that boils down to receiving and reacting to input; because of this, we can fill our containers with any number of an infinite array of possible applications. In principle, microservices open wide the possibilities for systems development.

In practice, this tends to mean that every team must be able to provide meaningful answers to a set of complex questions. It seems unreasonable, for instance, that a machine learning (ML) algorithm specialist should have to understand why Nginx's buffering is worth putting in front of a Unicorn Python service.

It's reasonable for someone who's never built a multimedia API before to have no idea that there's an important difference between throttling threads and throttling coroutines. They'll have no choice but to learn these lessons the hard way as their coroutines compound, downloading large files and snowballing into a mess that overflows RAM and grinds to a halt.

If every team is truly a DevOps team, we need easier options with less to think about. This allows us to focus on the parts of computing that drive each of us. We know we could learn it all if we put in the time — but we also know we won't. And, ultimately, we shouldn't have to!



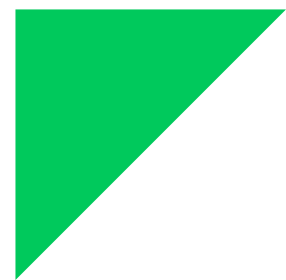
Functions: Atomic Units of Service Architecture

Building infrastructure around functions (assumed to be request handlers for HTTP or similar) is novel, and frankly unintuitive at first glance.



In reality, it follows the familiar pattern of the Unix Philosophy, which has been around for 40+ years. While it feels like the logical conclusion of all these years of simplifying infrastructure, it removes so much from the list of things we're used to thinking about that it's hard to take it seriously. One function? Is it even worth having a server for that at all? And how many containers are we talking about here?

The whole point of “serverless” architecture is, in fact, allowing us to stop thinking about containers and services entirely. Rather than requiring every development team to accumulate expertise on Docker Compose and Terraform, years of investments in automated container management have allowed us to step back and let the system take over on that front. Just as there's a role for C programmers who grok the nuances of `malloc()` and `vmalloc()`, there will always be a need for people who understand the complexities of Docker port mapping and service deployment. But it's just something that most of us don't need to spend our time thinking about anymore.



So let's apply our OOD principles to serverless:

The Single-Responsibility Principle

When the size of an endpoint can be shrunk to a single function, it seems clear that we have reached the logical extreme of a single modular service. While shared library code might underpin a set of related endpoints, each one can be independently created, modified, or removed in response to consumer needs.

The Interface Segregation Principle

Similarly, if a function is the equivalent of a service, there literally cannot be a smaller interface. Consuming a serverless endpoint doesn't require that you know anything about any other endpoint.



Databases in the Serverless World

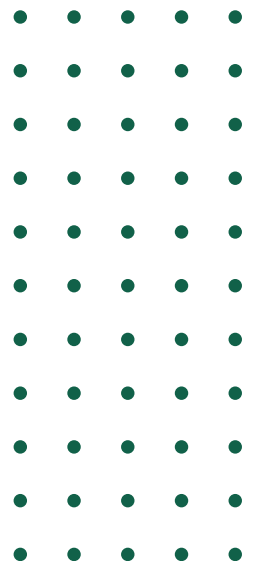
Yes, microservices make your organization nimble, thanks to loosely coupled, independently deployable applications.

But their siloed nature can make using self-managed databases cumbersome at best and impossible at worst. Each function can't have its own database and as your architecture becomes more complex, your database must as well.

A central, managed database like MongoDB Atlas solves this issue seamlessly. Much like microservice architecture, today's modern managed databases were designed to allow for rapid-fire development of the scalable, distributed, loosely coupled services that define the global web.

Document databases like MongoDB are ideally suited to rapid development, providing broad flexibility that allows them to effectively fill most roles traditionally held down by relational databases or single-purpose stores. MongoDB Atlas, on top of that, allows you to simply roll out new collections and even clusters with a button click, scale them up (and down) automatically with load, and leverage integrated full-text search and federated querying, to boot.

Both microservice and serverless architectures make dramatic gains when paired with a managed database service like Atlas.





Reduce Complexity, Increase Productivity


Looking at serverless architecture as the natural evolution of microservices, it becomes obvious why this shift is compelling:

We get to delete thousands of lines of boilerplate code, reduce the cognitive load on developers throughout the system, and drastically simplify the unit of software we're shipping.

The good news is that you don't have to jump into the deep end right away. It's easy to take a single service endpoint and try running it on a serverless platform. There are a few "gotchas" — namely, in understanding how things like database connections and state work.

That said, the learning curve is reasonably short, and the payoff is tremendous. Serverless systems respond much more quickly to changes in demand, they're generally much more efficient when it comes to resource usage, and they take away heaps of complexity and pain from the software development and deployment process.

It's definitely worth a try.



Ready to increase productivity
and give MongoDB Atlas a try?

Not yet ready to try it out? Have questions? [Contact us](#)

