# NGINX

BLOG  TECH          Chris Richardson of Eventuate, Inc.  May 19, 2015

# Introduction to Microservices

🏷 *microservices*, *monolithic application*, *Docker*, *cloud*

***Editor*** *– This seven-part series of articles is now complete:*

1. *Introduction to Microservices (this article)*

2. *Building Microservices: Using an API Gateway*

3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*

4. *Service Discovery in a Microservices Architecture*

5. *Event-Driven Data Management for Microservices*

6. *Choosing a Microservices Deployment Strategy*

7. *Refactoring a Monolith into Microservices*

*You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment.*

*Also check out our Microservices Solutions page.*

Microservices are currently getting a lot of attention: articles, blogs, discussions on social media, and conference presentations. They are rapidly heading towards the peak of inflated expectations on the Gartner Hype cycle. At the same time, there are skeptics in the software community who dismiss microservices as nothing new. Naysayers claim that the idea is just a rebranding of SOA. However, despite both the hype and the skepticism, the Microservices Architecture pattern has significant benefits – especially when it comes to enabling the agile development and delivery of complex enterprise applications.

This blog post is the first in a seven-part series about designing, building, and deploying microservices. You will learn about the approach and how it compares to the more traditional Monolithic Architecture pattern. This series will describe the various elements of a microservices architecture. You will learn about the benefits and drawbacks of the Microservices Architecture pattern, whether it makes sense for your project, and how to apply it.

Let's first look at why you should consider using microservices.

## Building Monolithic Applications

Let's imagine that you were starting to build a brand new taxi-hailing application intended to compete with Uber and Hailo. After some preliminary meetings and requirements gathering, you would create a new project either manually or by using a generator that comes with Rails, Spring Boot, Play, or Maven. This new application would have a modular hexagonal architecture, like in the following diagram:

What brought you to nginx.com today?
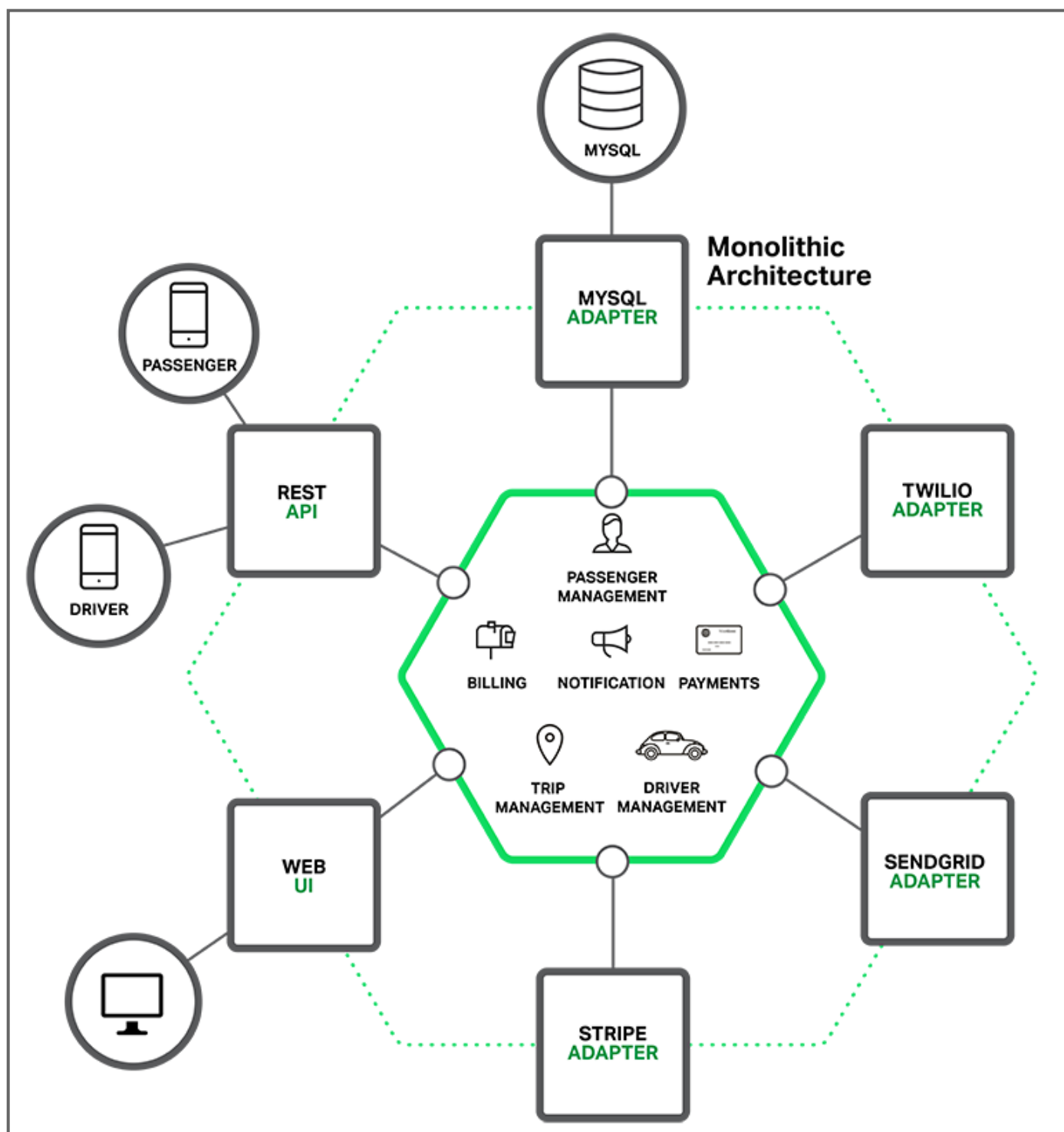
At the core of the application is the business logic, which is implemented by modules that define services, domain objects, and events. Surrounding the core are adapters that interface with the external world. Examples of adapters include database access components, messaging components that produce and consume messages, and web components that either expose APIs or implement a UI.

Despite having a logically modular architecture, the application is packaged and deployed as a monolith. The actual format depends on the application's language and framework. For example, many Java applications are packaged as WAR files and deployed on application servers such as Tomcat or Jetty. Other Java applications are packaged as self-contained executable JARs. Similarly, Rails and Node.js applications are packaged as a directory hierarchy.

Applications written in this style are extremely common. They are simple to develop since our IDEs and other tools are focused on building a single application. These kinds of applications are also simple to test. You can implement end-to-end testing by simply launching the application and testing the UI with Selenium. Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server. You can also scale the application by running multiple copies behind a load balancer. In the early stages of the project it works well.

# Marching Towards Monolithic Hell

Unfortunately, this simple approach has a huge limitation. Successful applications have a habit of growing over time and eventually becoming huge. During each sprint, your development team implements a few more stories, which, of course, means adding many lines of code. After a few years, your small, simple application will have grown into a monstrous monolith. To give an extr
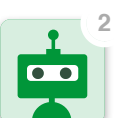
example, I recently spoke to a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million line of code (LOC) application. I'm sure it took the concerted effort of a large number of developers over many years to create such a beast.

Once your application has become a large, complex monolith, your development organization is probably in a world of pain. Any attempts at agile development and delivery will flounder. One major problem is that the application is overwhelmingly complex. It's simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. What's more, this tends to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly. You will end up with a monstrous, incomprehensible big ball of mud.

The sheer size of the application will also slow down development. The larger the application, the longer the start-up time is. For example, in a recent survey some developers reported start-up times as long as 12 minutes. I've also heard anecdotes of applications taking as long as 40 minutes to start up. If developers regularly have to restart the application server, then a large part of their day will be spent waiting around and their productivity will suffer.

Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment. Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith since you must redeploy the entire application in order to update any one part of it. The lengthy start-up times that I mentioned earlier won't help either. Also, since the impact of a change is usually not very well understood, it is likely that you have to do extensive manual testing. Consequently, continuous deployment is next to impossible to do.

Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances. Another module might be an in-memory database and best suited for EC2 Memory-optimized instances. However, because these modules are deployed together you have to compromise on the choice of hardware.

Another problem with monolithic applications is reliability. Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.

Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages. For example, let's imagine that you have 2 million lines of code written using the XYZ framework. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer ABC framework, even if that framework was considerably better. As a result, there is a huge barrier to adopting new technologies. You are stuck with whatever technology choices you made at the start of the project.

To summarize: you have a successful business-critical application that has grown into a monstrous monolith that very few, if any, developers understand. It is written using obsolete, unproductive technology that makes hiring talented developers difficult. The application is difficult to scale and is unreliable. As a result, agile development and delivery of applications is impossible.

So what can you do about it?

# Microservices – Tackling the Complexity

What brought you to nginx.com today?

Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting what is now known as the Microservices Architecture pattern. Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an API that's consumed by other microservices or by the application's clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud VM or a Docker container.

For example, a possible decomposition of the system described earlier is shown in the following diagram:
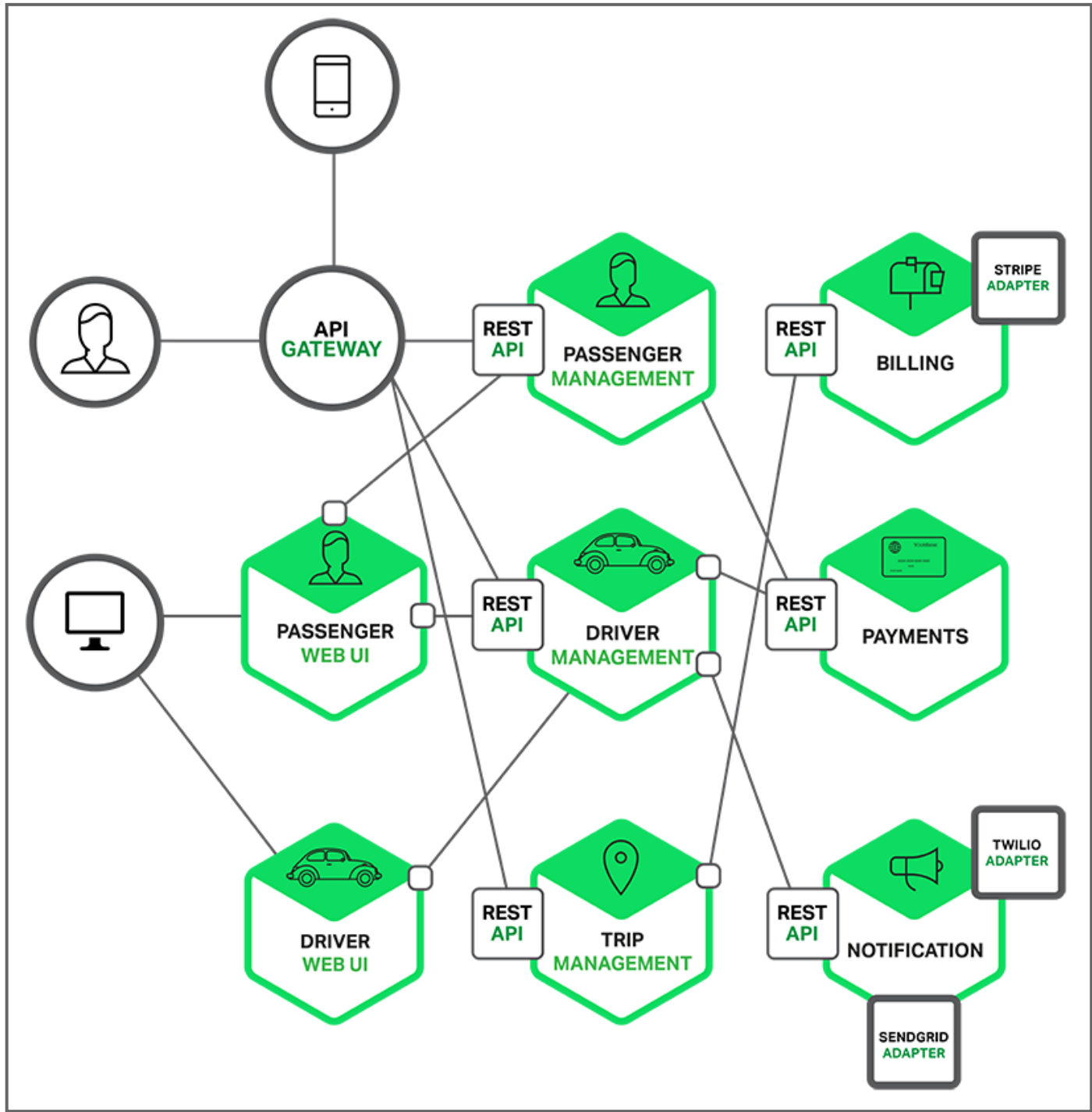


Each functional area of the application is now implemented by its own microservice. Moreover, the web application is split into a set of simpler web applications (such as one for passengers and one for drivers in our taxi-hailing example). This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip. The UI services invoke the other services in order to render web pages. Services might also use asynchronous, message-based communication. Inter-service communication will be covered in more detail later in this series.

Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The don't, however, have direct access to the backend services. Instead, communication is mediated an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load

balancing, caching, access control, API metering, and monitoring, and can be implemented effectively using NGINX. Later articles in the series will cover the API Gateway.
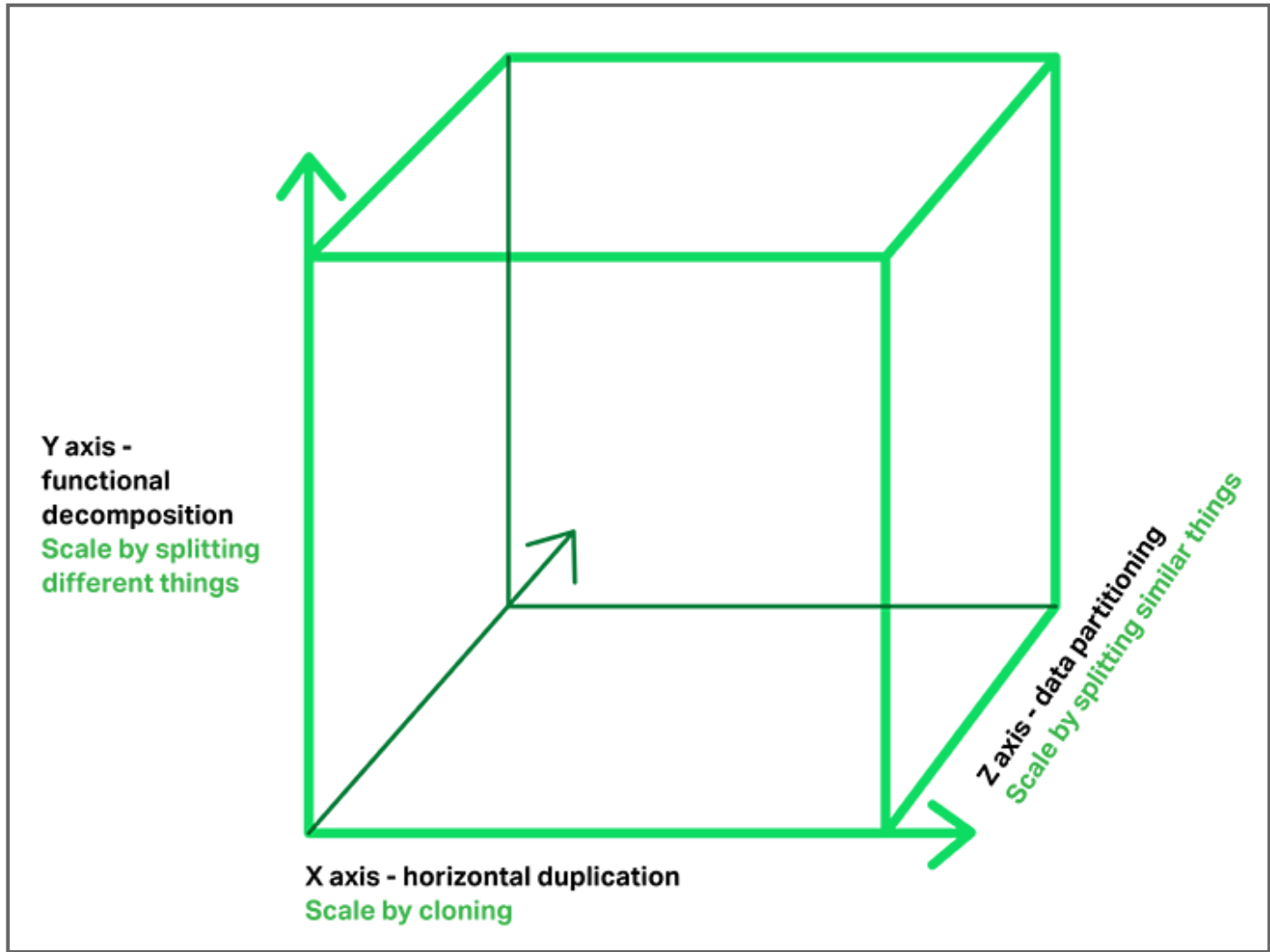


The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube, which is a 3D model of scalability from the excellent book *The Art of Scalability*. The other two scaling axes are X-axis scaling, which consists of running multiple identical copies of the application behind a load balancer, and Z-axis scaling (or data partitioning), where an attribute of the request (for example, the primary key of a row or identity of a customer) is used to route the request to a particular server.

Applications typically use the three types of scaling together. Y-axis scaling decomposes the application into microservices as shown above in the first figure in this section. At runtime, X-axis scaling runs multiple instances of each service behind a load balancer for throughput and availability. Some applications might also use Z-axis scaling to partition the services. The following diagram shows how the Trip Management service might be deployed with Docker running on Amazon EC2.

What brought you to nginx.com today?
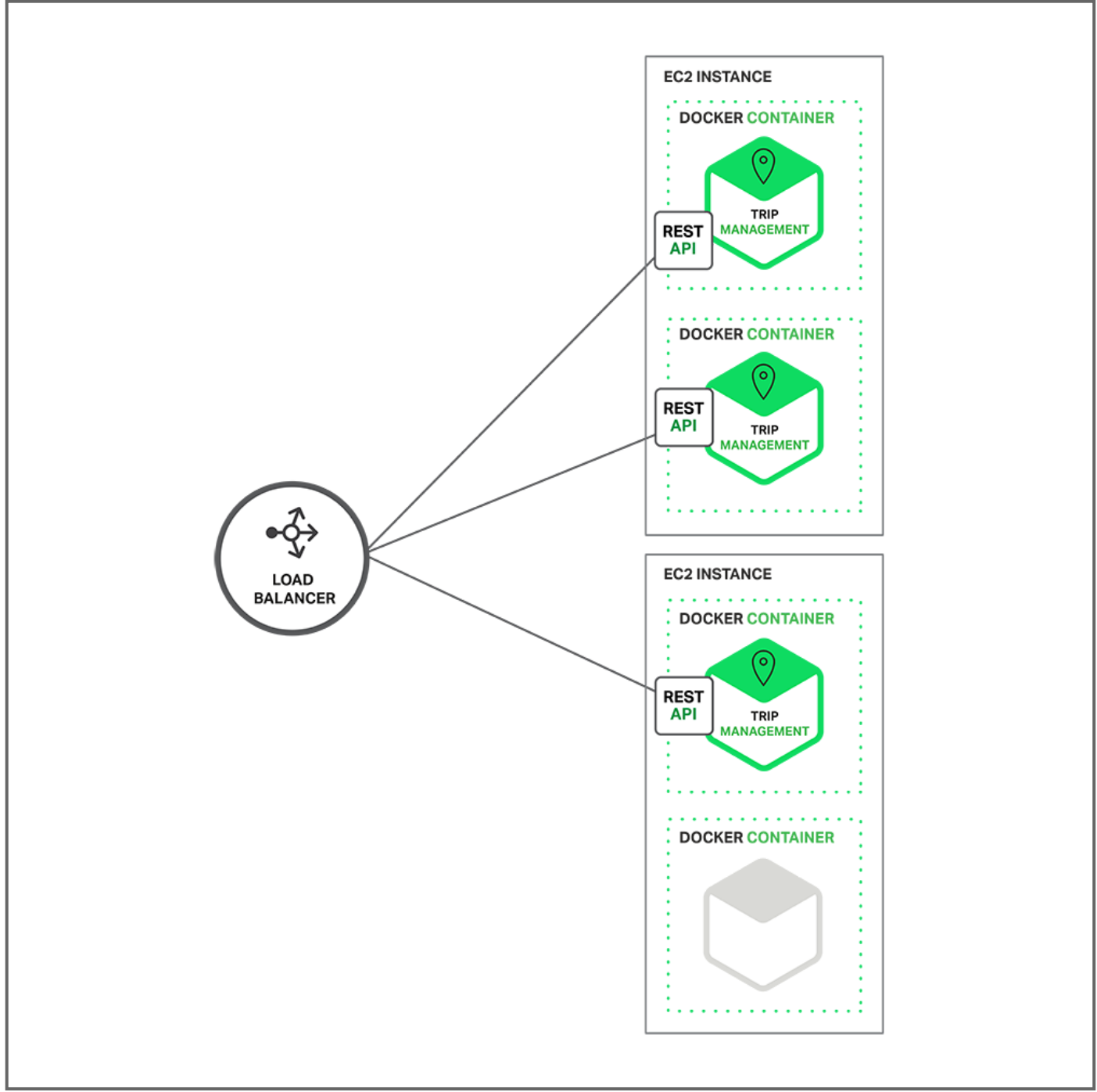
At runtime, the Trip Management service consists of multiple service instances. Each service instance is a Docker container. In order to be highly available, the containers are running on multiple Cloud VMs. In front of the service instances is a load balancer such as NGINX that distributes requests across the instances. The load balancer might also handle other concerns such as caching, access control, API metering, and monitoring.

The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling. The following diagram shows the database architecture for the example application.

What brought you to nginx.com today?
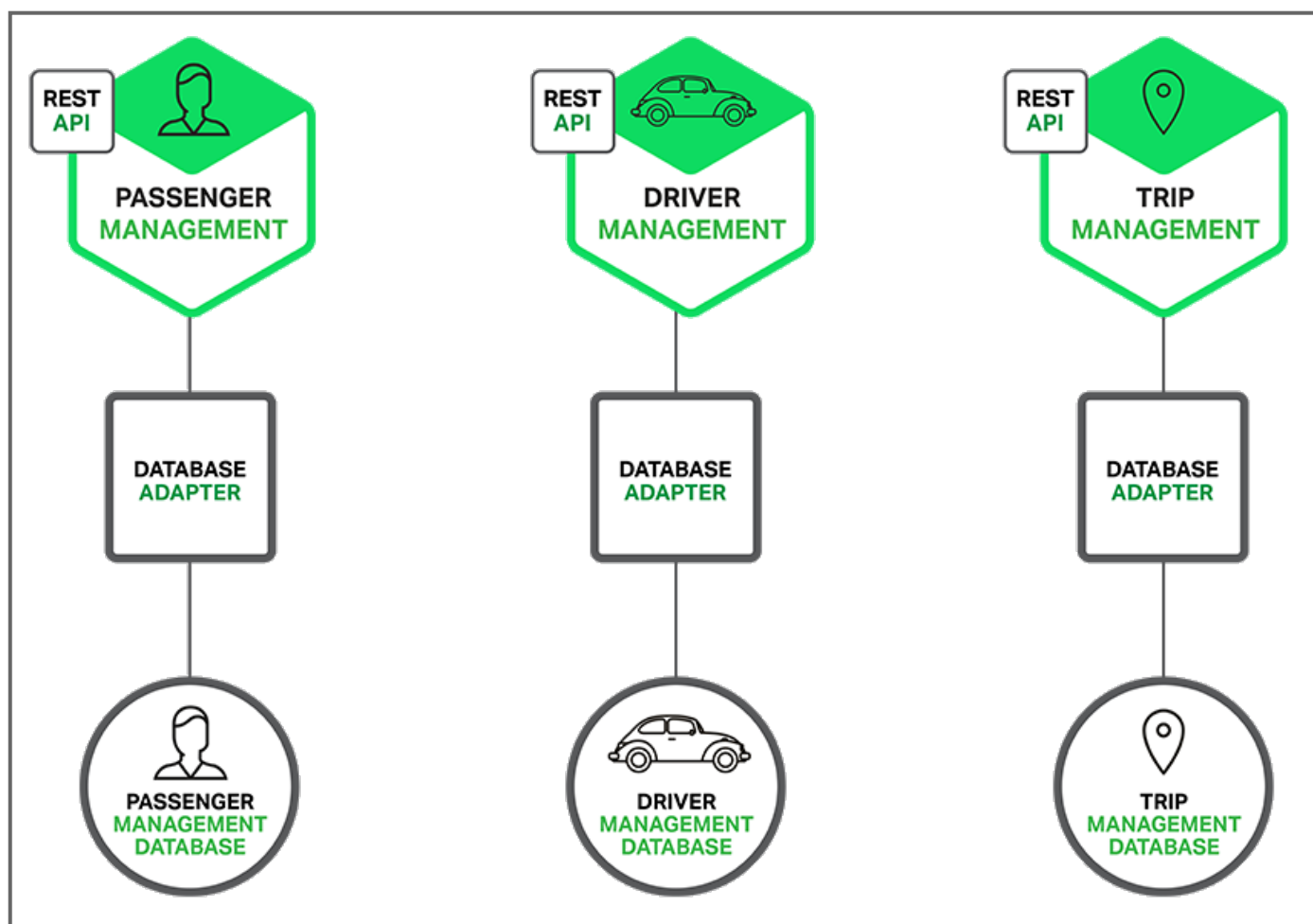
Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture. For example, Driver Management, which finds drivers close to a potential passenger, must use a database that supports efficient geo-queries.

On the surface, the Microservices Architecture pattern is similar to SOA. With both approaches, the architecture consists of a set of services. However, one way to think about the Microservices Architecture pattern is that it's SOA without the commercialization and perceived baggage of web service specifications (WS-*) and an Enterprise Service Bus (ESB). Microservice-based applications favor simpler, lightweight protocols such as REST, rather than WS-*. They also very much avoid using ESBs and instead implement ESB-like functionality in the microservices themselves. The Microservices Architecture pattern also rejects other parts of SOA, such as the concept of a canonical schema.

## The Benefits of Microservices

The Microservices Architecture pattern has a number of important benefits. First, it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services. Each service has a well-defined boundary in the form of an RPC- or message-driven API. The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain.

Second, this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract. Of course, most organizations would want to avoid complete anarchy and limit technology options. However, this freedom means that developers are no longer obligated to use the possibly obsolete technologies that existed at the start of a new project. When writing a new service, they have the option of using current technology. Moreover, since services are relatively small it becomes feasible to rewrite an old service using current technolo

What brought you to nginx.com today?

Third, the Microservices Architecture pattern enables each microservice to be deployed independently. Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested. The UI team can, for example, perform A/B testing and rapidly iterate on UI changes. The Microservices Architecture pattern makes continuous deployment possible.

Finally, the Microservices Architecture pattern enables each service to be scaled independently. You can deploy just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, you can use the hardware that best matches a service's resource requirements. For example, you can deploy a CPU-intensive image processing service on EC2 Compute Optimized instances and deploy an in-memory database service on EC2 Memory-optimized instances.

# The Drawbacks of Microservices

As Fred Brooks wrote almost 30 years ago, there are no silver bullets. Like every other technology, the Microservices architecture has drawbacks. One drawback is the name itself. The term *microservice* places excessive emphasis on service size. In fact, there are some developers who advocate for building extremely fine-grained 10–100 LOC services. While small services are preferable, it's important to remember that they are a means to an end and not the primary goal. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

Another major drawback of microservices is the complexity that arises from the fact that a microservices application is a distributed system. Developers need to choose and implement an inter-process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure since the destination of a request might be slow or unavailable. While none of this is rocket science, it's much more complex than in a monolithic application where modules invoke one another via language-level method/procedure calls.

Another challenge with microservices is the partitioned database architecture. Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices-based application, however, you need to update multiple databases owned by different services. Using distributed transactions is usually not an option, and not only because of the CAP theorem. They simply are not supported by many of today's highly scalable NoSQL databases and messaging brokers. You end up having to use an eventual consistency based approach, which is more challenging for developers.

Testing a microservices application is also much more complex. For example, with a modern framework such as Spring Boot it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast, a similar test class for a service would need to launch that service and any services that it depends upon (or at least configure stubs for those services). Once again, this is not rocket science but it's important to not underestimate the complexity of doing this.

Another major challenge with the Microservices Architecture pattern is implementing changes that span multiple services. For example, let's imagine that you are implementing a story that requires changes to services A, B, and C, where A depends upon B and B depends upon C. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In contrast, in a Microservices Architecture pattern you need to carefully plan and coordinate the rollout of changes to each of the services. For example, you would need to update service C, followed by service B, and then finally service A. Fortunately, most changes typically impact only one service and multi-service changes that require coordination are relatively rare.

What brought you to nginx.com today?

Deploying a microservices-based application is also much more complex. A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer. Each application instance is configured with the locations (host and ports) of infrastructure services such as the database and a message broker. In contrast, a microservice application typically consists of a large number of services. For example, Hailo has 160 different services and Netflix has over 600 according to Adrian Cockcroft [Editor – Hailo has been acquired by MyTaxi.]. Each service will have multiple runtime instances. That's many more moving parts that need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism (discussed in a later post) that enables a service to discover the locations (hosts and ports) of any other services it needs to communicate with. Traditional trouble ticket-based and manual approaches to operations cannot scale to this level of complexity. Consequently, successfully deploying a microservices application requires greater control of deployment methods by developers, and a high level of automation.

One approach to automation is to use an off-the-shelf PaaS such as Cloud Foundry. A PaaS provides developers with an easy way to deploy and manage their microservices. It insulates them from concerns such as procuring and configuring IT resources. At the same time, the systems and network professionals who configure the PaaS can ensure compliance with best practices and with company policies. Another way to automate the deployment of microservices is to develop what is essentially your own PaaS. One typical starting point is to use a clustering solution, such as Kubernetes, in conjunction with a technology such as Docker. Later in this series we will look at how software-based application delivery approaches like NGINX, which easily handles caching, access control, API metering, and monitoring at the microservice level, can help solve this problem.

# Summary

Building complex applications is inherently difficult. A Monolithic architecture only makes sense for simple, lightweight applications. You will end up in a world of pain if you use it for complex applications. The Microservices architecture pattern is the better choice for complex, evolving applications despite the drawbacks and implementation challenges.

In later blog posts, I'll dive into the details of various aspects of the Microservices Architecture pattern and discuss topics such as service discovery, service deployment options, and strategies for refactoring a monolithic application into services.

Stay tuned...

**Editor** – This seven-part series of articles is now complete:

1. Introduction to Microservices (this article)

2. Building Microservices: Using an API Gateway

3. Building Microservices: Inter-Process Communication in a Microservices Architecture

4. Service Discovery in a Microservices Architecture

5. Event-Driven Data Management for Microservices

6. Choosing a Microservices Deployment Strategy

7. Refactoring a Monolith into Microservices

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment.

What brought you to nginx.com today?
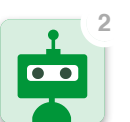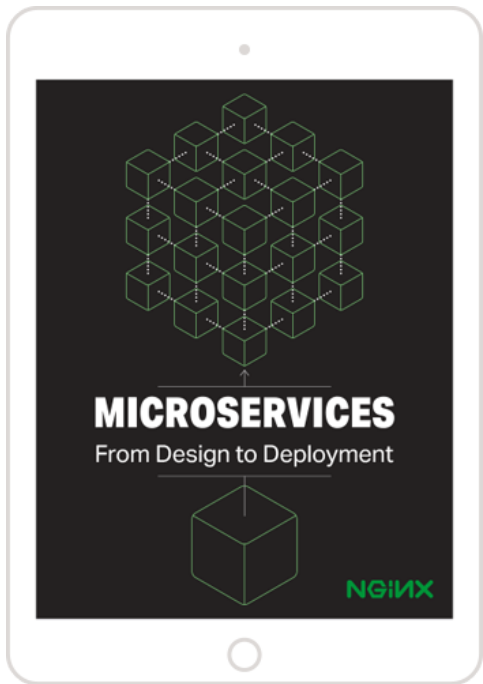
# Microservices: From Design to Deployment

The complete guide to microservices development

DOWNLOAD NOW

---

**38 Comments**　　**NGINX**　　　　　　　　　🔴 **1**　**Login**　▾

♡ **Recommend**　25　　　🐦 **Tweet**　　f **Share**　　　　　**Sort by Best** ▾

Join the discussion…

**LOG IN WITH**　　　**OR SIGN UP WITH DISQUS** ⑦

Name

---

**Nino Martincevic** • 3 years ago

A very nice overview.
But sadly in many ascpects as wrong as many others of the same kind.

Whenever I read something like the following paragraphs in a post about Microservices I am kind of frightened, because the real essence of Microservices hasn't been grasped right - and we are probably facing a new wave of MicroSOA:

"The term microservice places excessive emphasis on service size. ..."

Most authors think of services in a technical way. But that's simply wrong.
Services are build around (business) capabilities. Determine the boundary of that capability and you get the right size.

"Another major challenge with the Microservice architecture pattern is implementing changes that span multiple services. For example, let's imagine that you are implementing a story that requires changes to services A, B, and C, where A depends upon B and B depends upon C. "

What brought you to nginx.com today?

If Microservices "depend" on others, they are simply not decomposed right.
A Microservice may USE another one (more precise: should use the abstraction of another), but never depend on it. If it's the case then it's in fact one service, wrongly decomposed into two.

If you introduce dependencies into Microservice you've lost almost any of the benefits.

9 ∧ | ∨ · Reply · Share ›

**Rudra** ➜ Nino Martincevic · 3 years ago

Thanks for providing additional thoughts on the wonderfully described original topic.
I have a question on the following point
"If Microservices "depend" on others, they are simply not decomposed right.
A
Microservice may USE another one (more precise: should use the
abstraction of another), but never depend on it. If it's the case then
it's in fact one service, wrongly decomposed into two."

Let me know if I am wrong, isn't it correct to say that when Microservice may USE another one(abstraction or not) it is already depending on it by merely using it?
Taking from example model described in the article, The Trip Management Service will use the Driver and Passenger service and any business logic changes related to how Drivers and Passengers are definitely going to impact dependent services...
For the sake of an example: A business need is proposed on selecting Driver based on Passenger Spoken Language priority will need updating both Driver and Trip Management Service.

How do you propose decomposing the above model without having a dependency?

∧ | ∨ · Reply · Share ›

**Nino Martincevic** ➜ Rudra · 3 years ago

"Using" A1 also means to be able to decide to use A2 or A1 again. That's not a (hard) dependency.

Why the Trip Management Service (TMS) has to use the other two?
"any business logic changes related to how Drivers and Passengers are definitely going to impact dependent services"
This is exactly what SHOULD NOT happen. That's a sign of an unclear or unneeded boundary.

The Drivers service handles all about drivers and you can reference a driver by it's ID from outside. But in the TMS you only need that reference, you don't change Drivers from there. Simply give the TMS anything it needs for fulfilling its responsibility, don't make it dependant on others services.

E.g. Adding passengers to a trip does not mean Trip->addPassenger(new Passenger...)) but Trip->addPassenger(passengerId)

You example, selecting driver on language is easy
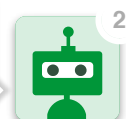Trip->selectDriver('en');
Do I need to talk or depend to Passengers for that? No.

Carve out the domains and set clear boundary to them, regarding their responsibility on state changes. They talk with each other with the simplest possible protocols and data, in most case just with the references (aggregate IDs).

∧ | ∨ · Reply · Share ›

What brought you to nginx.com today?

**Rudra** ➜ Nino Martincevic • 3 years ago

Thanks Nino for explaining that in detail...:)
Do taking this question further(hoping that I am
not bothering you and I am just seeking
clarification because I feel one need to
understand the subtle logic behind the
tool/technology rather than jumping the wagon
everyone is doing this).
So taking the example that we are discussing:
Trip will need information like passenger-
>getPreferredLang() to identify the Passenger
preference.

We may suggest a method Trip-
>addPassenger(passengerId, lang)

But someone needs to extract lang from
passenger->getPreferredLang(), will this be
done by someone aggregating the service
externally or the Trip service should internally
handle this?

Second Trip->selectDriver('en') will need to call
Driver->getDrivers(with lang and other criteria)
making Trip dependent on Driver...
or Trip service can directly got to data store
and fetch Drivers - Is this allowed as per Micro
Services are concerned.

3 ⌃ | ⌄ • Reply • Share ›

**Rommel Oramas** • 3 years ago

What tool do you use for your diagrams ?

3 ⌃ | ⌄ • Reply • Share ›

**Peter Guagenti** ➜ Rommel Oramas • 3 years ago

We use Omnigraffle for most of our diagrams.

⌃ | ⌄ • Reply • Share ›

**Yusef Ho** ➜ Peter Guagenti • a year ago

Thanks.
Can you share the materials for drawing the diagrams?
Or any resources/tutorials

⌃ | ⌄ • Reply • Share ›

**Nemo Wong** ➜ Rommel Oramas • 3 years ago

I would also like to know.

⌃ | ⌄ • Reply • Share ›

**Richard Rodger** • 4 years ago

Martin Fowler recommends a Monolith First approach to building
microservice systems. That is, build a "small" monolith to move
quickly at the start, and then break it up while you still have good
internal boundaries. Your outline of the taxi application in this article
seems to follow that approach, or I am reading too much into it?

I've found that Monolith First is unnecessary if you abstract inter-
service communication from the start. This abstraction layer removes
the friction that networked microservices introduce. On the other
hand, you're not facing up to the need to get your automation in
order right from the start. Another thing that helps is to be message-
focused, rather than service-focused. That way you aren't stuck with
bad service decisions and can re-group messages into more
appropriate services.

Microservices as mini-hexagonal architectures is a great way to
describe them.

Upgrading a service dependency chain is actually not so hard in
practice, so long as you practice Postel's law. In your service A, B, C
example, we would add version 2 of C into the load balancer
alongside version 1 of C, and then measure correctness and
performance. Once you've happy, turn of C version 1. Rinse and

What brought you to nginx.com today?

performance. Once you've happy, turn of O version 1. Rinse and repeat in discrete steps and you completely the nasty coupled deployments gotcha.

3 ∧ | ∨ • **Reply** • **Share ›**

**mrkris** → Richard Rodger • 4 years ago

I agree with Martin Fowler in situations where the product is loosely defined. On the other hand, if your product is well defined and unlikely to change significantly, implementing microservices from the start isn't such a bad idea.

3 ∧ | ∨ • **Reply** • **Share ›**

**Diogo** → mrkris • 4 years ago

Most startups start with a team of what 3-5 guys? You need to iterate really quickly to get your product and idea right, and for most of these companies it takes what, 1-3 years before the product vision is aligned? Implementing microservices from the get go is an horrible idea, given that setting up the entire production, testing and develoment environments, the infrastrcture costs, non-linear development, etc. will only take precious time and focus that would otherwise be wisely used to iterate over the product, in a lean and fast way. There is nothing wrong with going Monolith first, and its A-OK to throw the entire thing out one or two years down the road and rebuild the entire thing. If the technical lead did his/her job while the Monolith was being built, there should be no reason why it cannot be fragmented into microservices throughout time given that internally the Monolith was already using services in the first place.

4 ∧ | ∨ • **Reply** • **Share ›**

**mrkris** → Diogo • 4 years ago

Like I said, if you know the domain and it's not changing, then it's not a bad idea. This isn't often the case, but it does happen. I'm not against the monolith approach at all, it's what I do a good 95% of the time.

2 ∧ | ∨ • **Reply** • **Share ›**

**Web Team** Mod → mrkris • 4 years ago

I agree. Monoliths are fine for prototypes and early betas. But, it's time to move to a microservices architecture as soon as performance and overall user experience matters.

2 ∧ | ∨ • **Reply** • **Share ›**

**crash025** → Richard Rodger • 4 years ago

Starting with microservices out of the bat before you have a system up and running is a nightmare.

You don't have a consistent deployment practice in place, your domain is changing, you're going to have to write a bunch of clients, do a lot of testing on the services/clients, and you're going to have to maintain a lot of the development infrastructure. It's a nightmare.

Its much easier to break things up later and claim a huge win than to start later.

1 ∧ | ∨ • **Reply** • **Share ›**

**Amol Deshpande** → Richard Rodger • 4 years ago

Richard, if I'm recalling correctly Martin pointed out that of the success stories that he has noted, they have been the Monolith to start with, but that's not necessarily a recommendation.

∧ | ∨ • **Reply** • **Share ›**

**Tad Christiansen** → Amol Deshpande • 3 years ago

For what it is worth, here is the exact quote: "One

What brought you to nginx.com today?

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.)" (http://martinfowler.com/art...

∧ | ∨ · Reply · Share ›

**Matthew Lowery** · 2 years ago

Microservices vs SOA is not the question here. Most people don't actually get this.

What is, is the breakdown between a high level agnostic service, not tied to any company or technology, vs those detailed delivery mechanisms/services that are also not tied to any company or technology vendor. What makes a monolithic architecture is the payload.

1 ∧ | ∨ · Reply · Share ›

**Vicky** · a year ago

nice article ..for microservices ..

∧ | ∨ · Reply · Share ›

**Nikita Ekaterinchuk** · 2 years ago

What to do if I have some core module (maybe with big size) and my microservices depend on it? Do I need install it with each service? Isn't it considered as code duplication?

∧ | ∨ · Reply · Share ›

**Chris Richardson** ➔ Nikita Ekaterinchuk · 2 years ago

Depends on what you mean by "core module".
If it is generic functionality, e.g. a workflow library, then that is most likely ok.
Sharing utility libraries makes sense.

However, if that module contains business logic then that is likely to be a problem.
For example, whenever the business rules implemented by that module change then you would have to update and redeploy multiple services.
That defeats the goal of the microservice architecture.
Business logic that needs to be shared should be a service.

∧ | ∨ · Reply · Share ›

**Christopher Stetson** ➔ Nikita Ekaterinchuk · 2 years ago

Microservices, by design, sacrifice centralized code and code reuse for ease of testing, deployment and scalability. Just as the container has large parts of the OS duplicated, having code duplicated in each image is ok and normal.

∧ | ∨ · Reply · Share ›

**MaxWell** · 3 years ago

This is an awesome intro but can you fix the broken images pretty please?

∧ | ∨ · Reply · Share ›

**Tony Mauro** Mod ➔ MaxWell · 3 years ago

Please be more specific about how the images are "broken". Are they missing, distorted, cutoff, or what? Are you using a desktop/laptop or a smartphone/tablet to read the article?

∧ | ∨ · Reply · Share ›

**MaxWell** ➔ Tony Mauro · 3 years ago

They are broken image links. Opening the images in a new tab gives me a 404 page. I am viewing in Chrome on an MBP

∧ | ∨ · Reply · Share ›

**Faisal Memon** Mod ➔ MaxWell · 3 years ago

Everything seems ok to me.

What brought you to nginx.com today?

^ | ˅ • Reply • Share ›

**MaxWell** ➜ Faisal Memon • 3 years ago
Tried on both phone and laptop on my ISP and mobile data and both times images are failing to load.

^ | ˅ • Reply • Share ›

**Kumar Vishesh Srivastava** ➜ MaxWell
• 3 years ago
Hello, the images are indeed broken - they return a 404. eg:
https://www.nginx.com/wp-co...

I am accessing this from London

^ | ˅ • Reply • Share ›

**Chris Richardson** ➜ Kumar Vishesh Srivastava
• 3 years ago
This is a very strange problem. Works for me in the San Francisco area:

$ curl -I https://www.nginx.com/wp-co...
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 09 Aug 2016 15:38:40 GMT

^ | ˅ • Reply • Share ›

**Kumar Vishesh Srivastava** ➜ Chris
Richardson • 3 years ago
Strange indeed Chris. I rechecked and I am still getting a 404.

---

<div align="center">

[ **TRY NGINX PLUS FOR FREE** ]   [ **ASK US A QUESTION** ]

</div>

**Products**

NGINX Plus

NGINX Controller

NGINX Unit

NGINX Amplify

NGINX Web Application Firewall

**NGINX on Github**

NGINX Open Source

NGINX Unit

NGINX Amplify

NGINX Kubernetes Ingress Controller

nginMesh

NGINX Microservices Reference Architecture

NGINX Crossplane

**Connect With Us**

**Solutions**

ADC / Load Balancing

Microservices

Cloud

Security

Web & Mobile Performance

API Gateway

**Resources**

Documentation

Ebooks

Webinars

Datasheets

Success stories

Blog

FAQ

Learn

Glossary

**Support**

Professional Services

Training

Customer Portal Login

**Partners**

Amazon Web Services

Google Cloud Platform

IBM

Microsoft Azure

Red Hat

Find a Partner

Certified module program

**Company**

About NGINX

Careers

Leadership

Press

Events

What brought you to nginx.com today?

STAY IN THE LOOP

What brought you to nginx.com today?