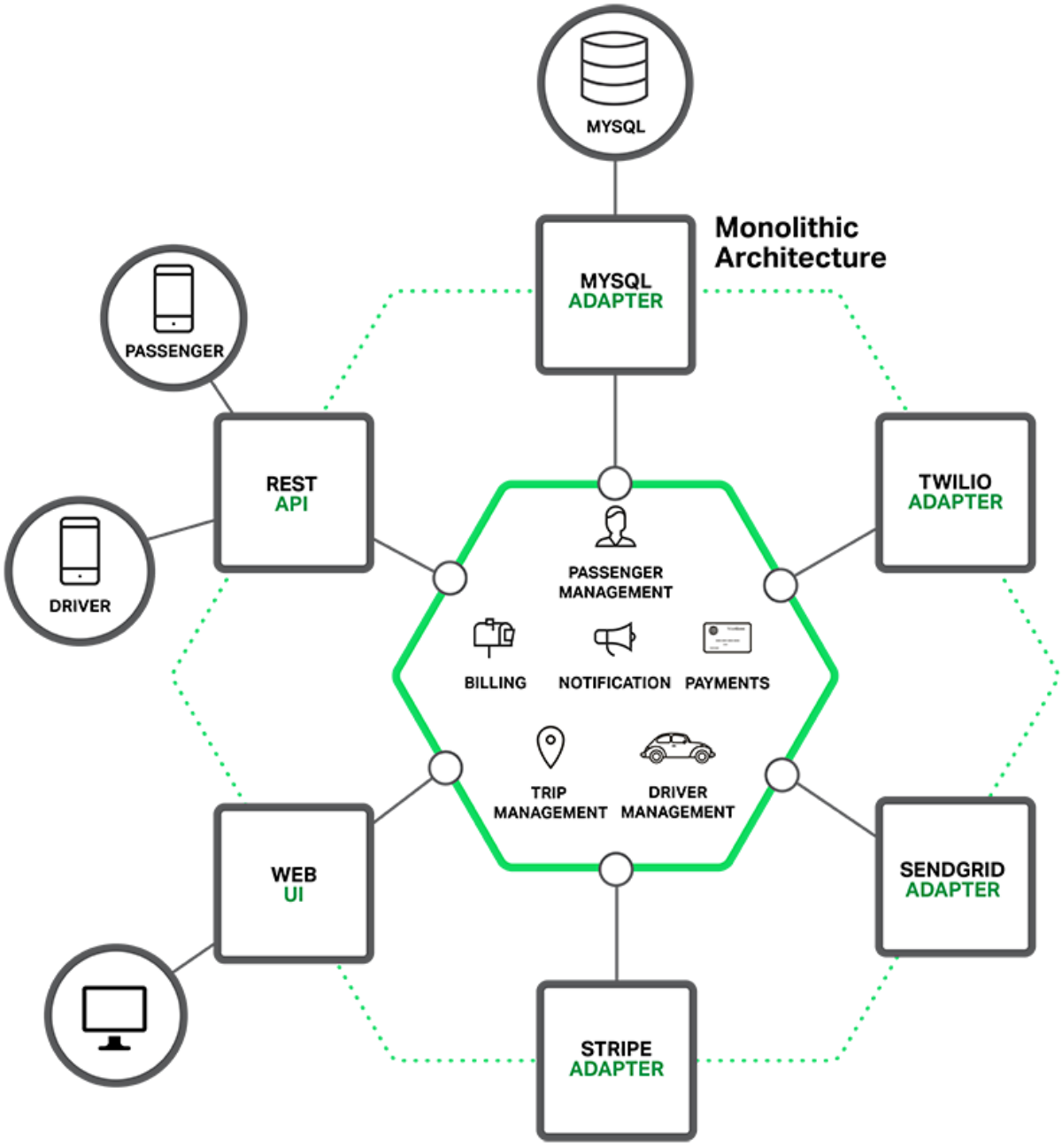
Introduction to Microservices

1. Building Monolithic Applications

Let's imagine that you were starting to build a new taxi-hailing application. 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:



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.

Towards Monolithic Architecture

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 extreme 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.

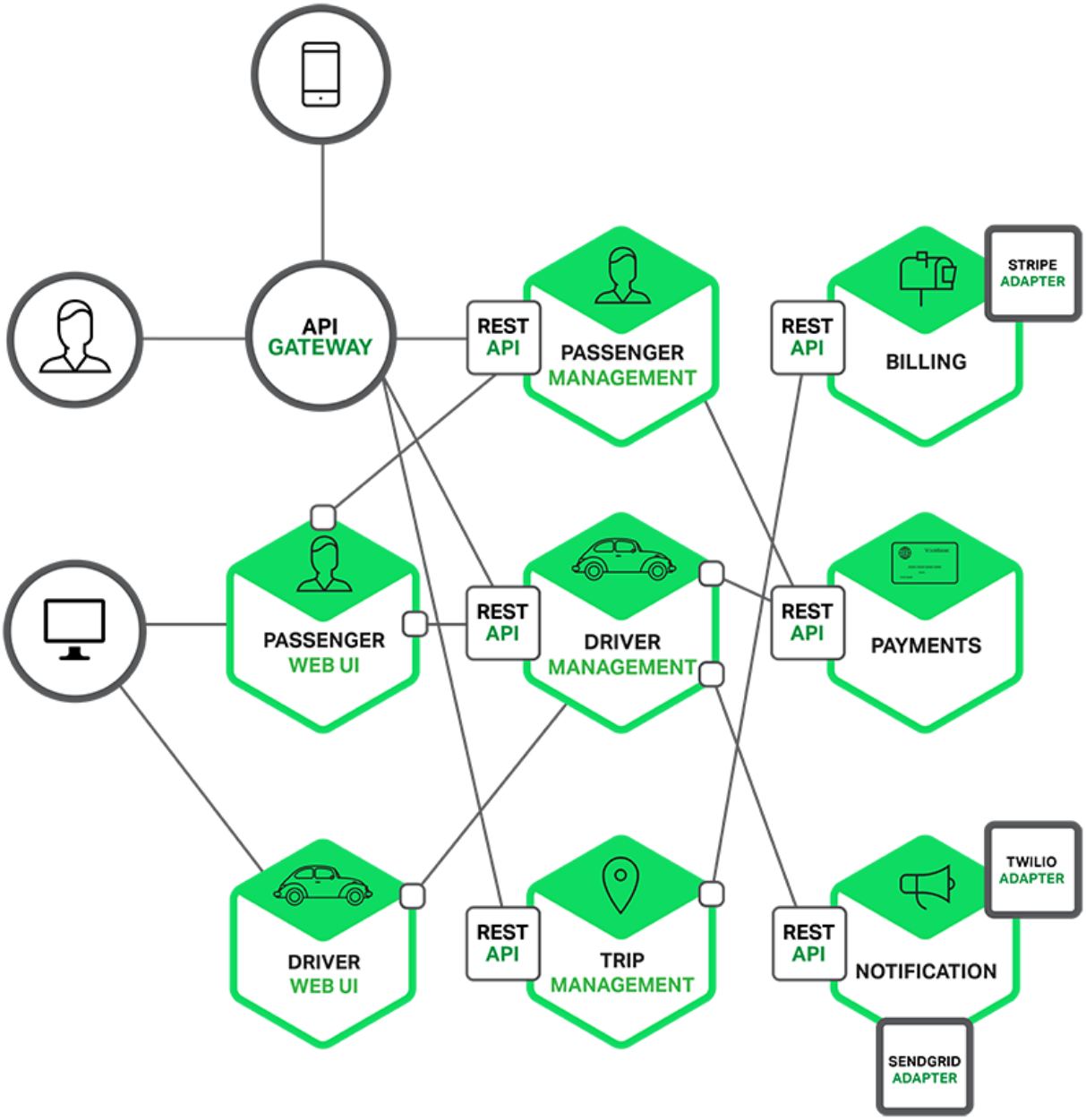
To summarize: you have a successful business-critical application that has grown into a monstrous monolith that very few, if any, developers understand. It is written using obsolete, unproductive technology that makes hiring talented developers difficult. The application is difficult to scale and is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

Microservices – Tackling the Complexity

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:

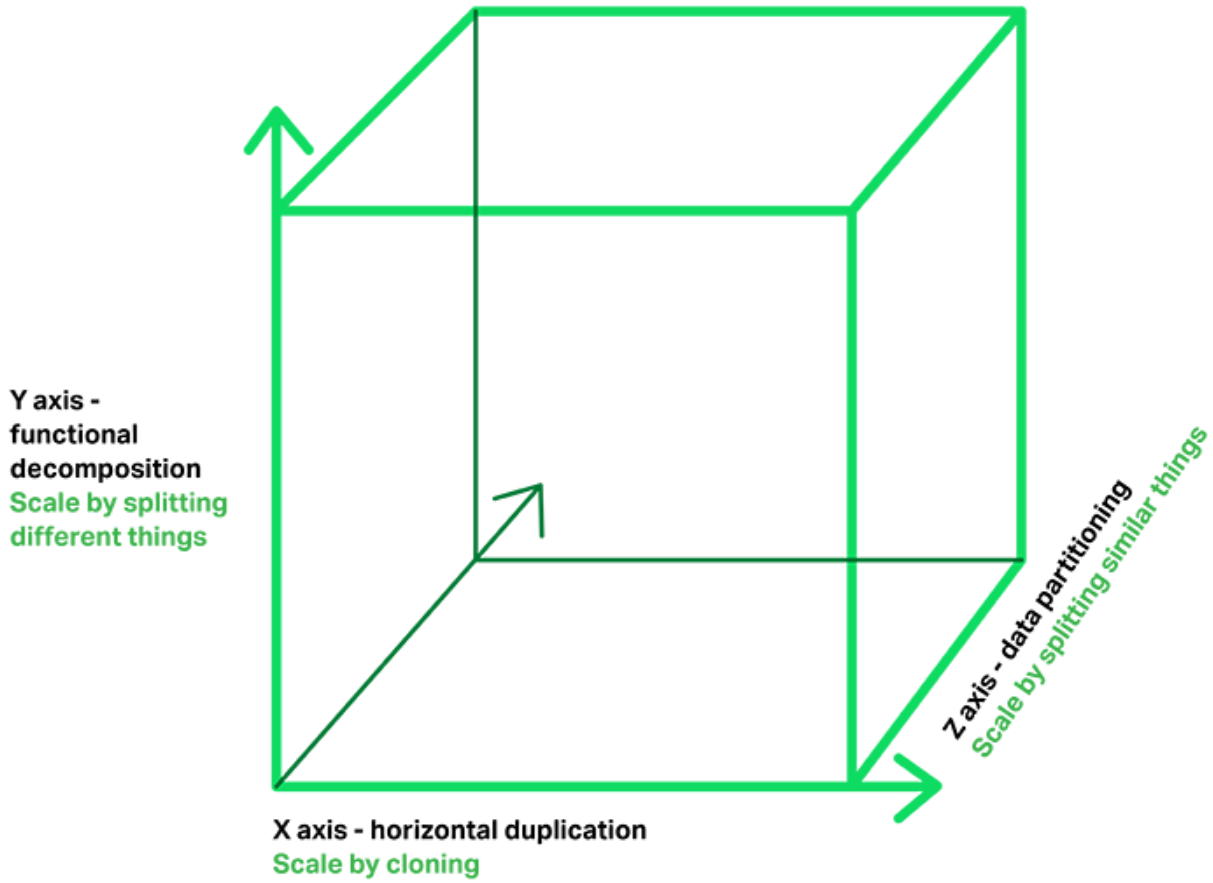


Microservices architecture for a sample ride-for-hire app, with each microservice presenting a RESTful API

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.

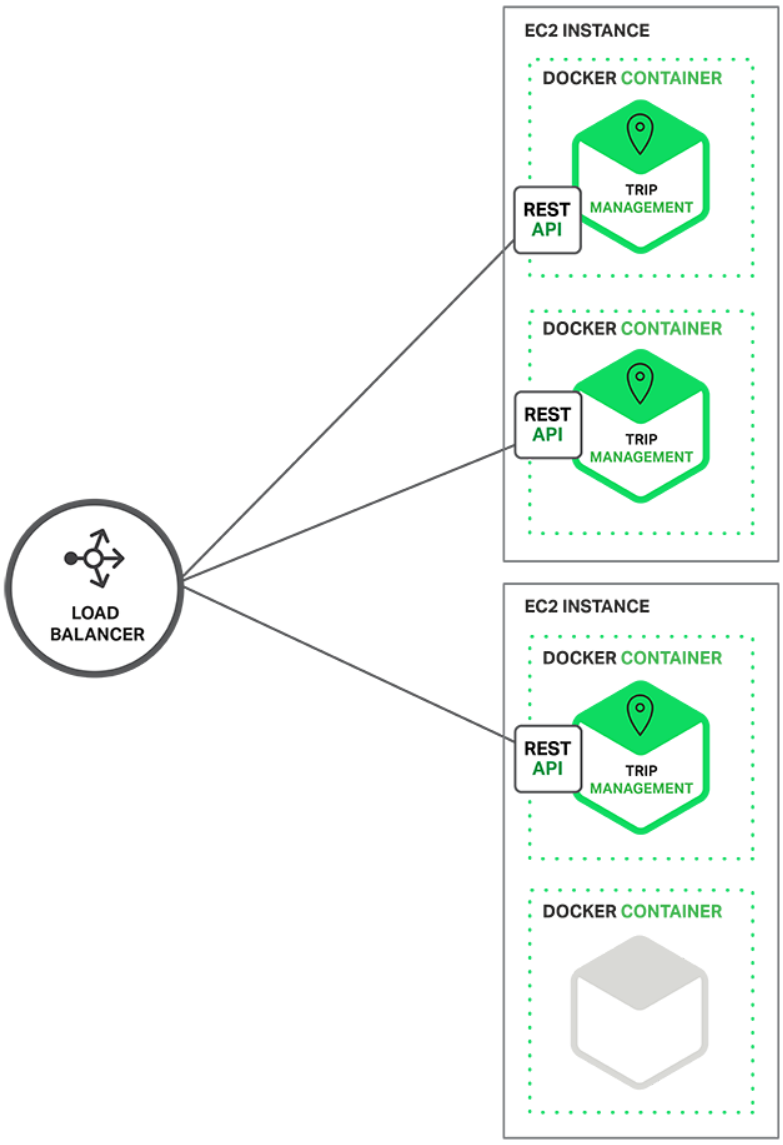
Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The apps don't, however, have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring, and can be implemented effectively using NGINX. Later articles in the series will cover the API Gateway.



The 'Scale Cube' with functional decomposition into microservices on the Y-axis

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.



Sample microservices app for ride-for-hire service, deployed in Docker containers and fronted by a load balancer

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.

Database architecture in sample microservices application for ride service

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.