Building Monolithic Applications

* **Difficult to scale** when different modules have conflicting resource requirements.

For example: one module implement CPU intensive image processing logic and would

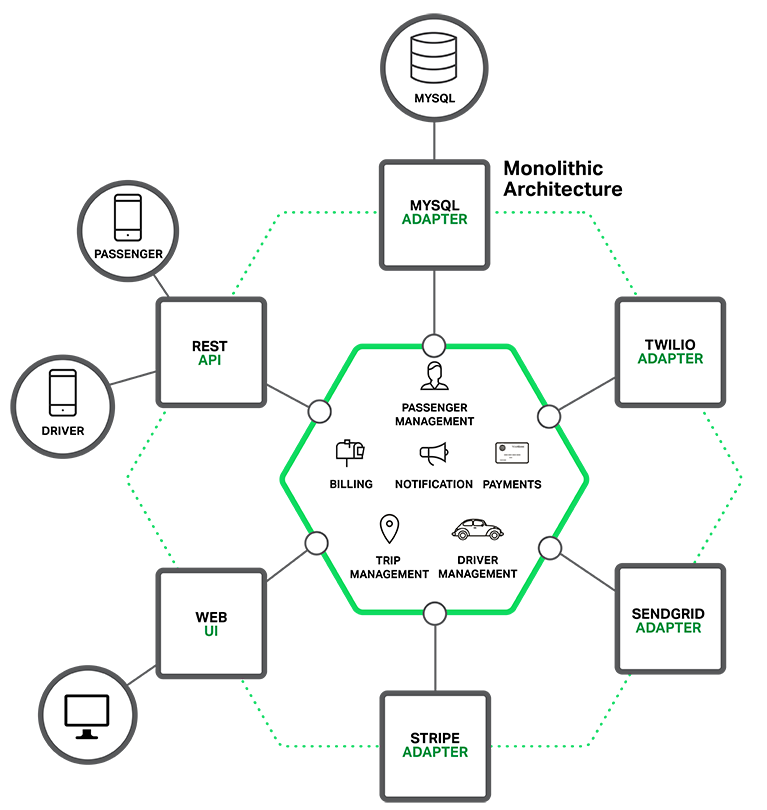
Ideally be deployed in EC2 compute optimized instanc . Another module might be an in-memory database and best suited for EC2 memory optimized instance. However these modules

Are deployed together you have to compromise on the choice of hardware.

* Another problem with Monolithic applications is **RELIABILITY**,

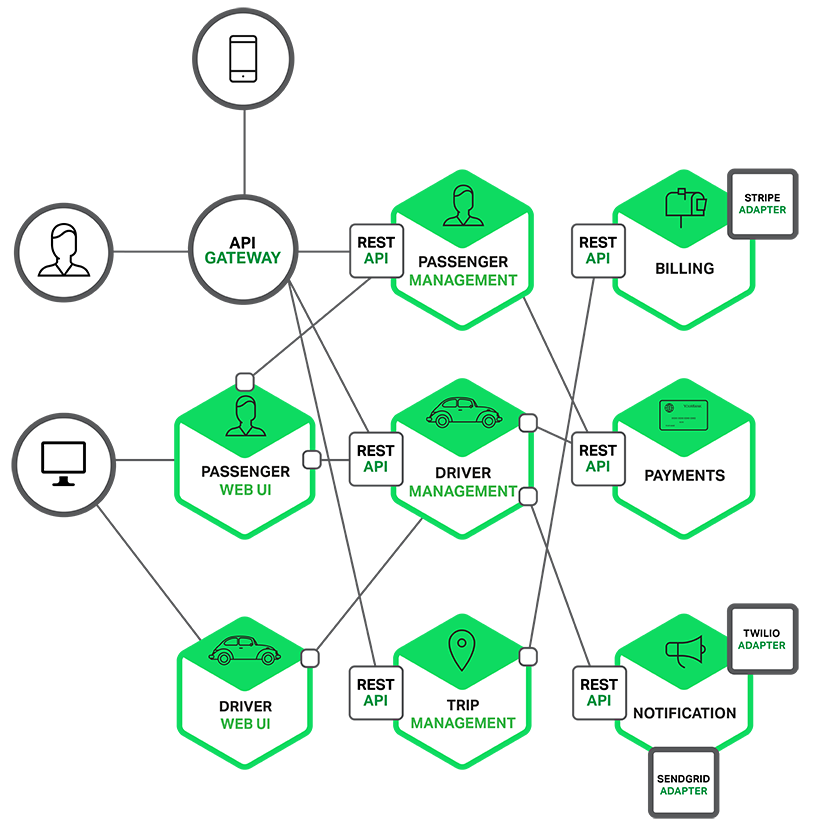
Because all modules are running within the same process, a bug in any module, such as memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the **AVAILABILITY** of the entire application.

* 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.



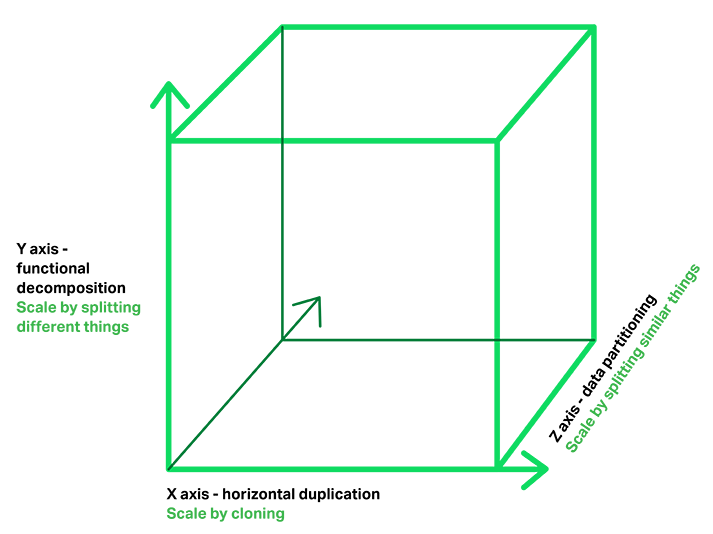
## Microservices – Tackling the Complexity

 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.



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

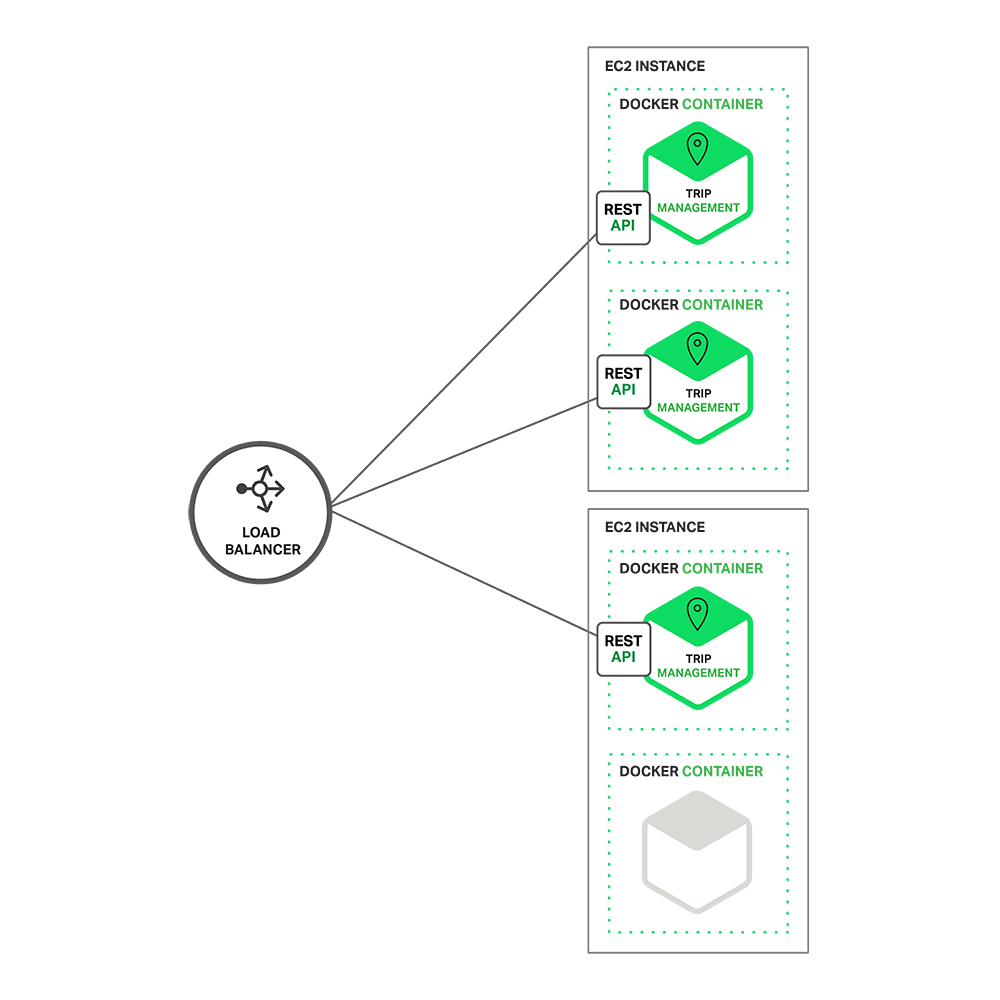
Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip. The UI services invoke the other services in order to render web pages. Services might also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



The Microservices Architecture pattern corresponds to the Y‑axis scaling of the [Scale Cube](http://microservices.io/articles/scalecube.html), which is a 3D model of scalability from the excellent book [The Art of Scalability](http://theartofscalability.com/). 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

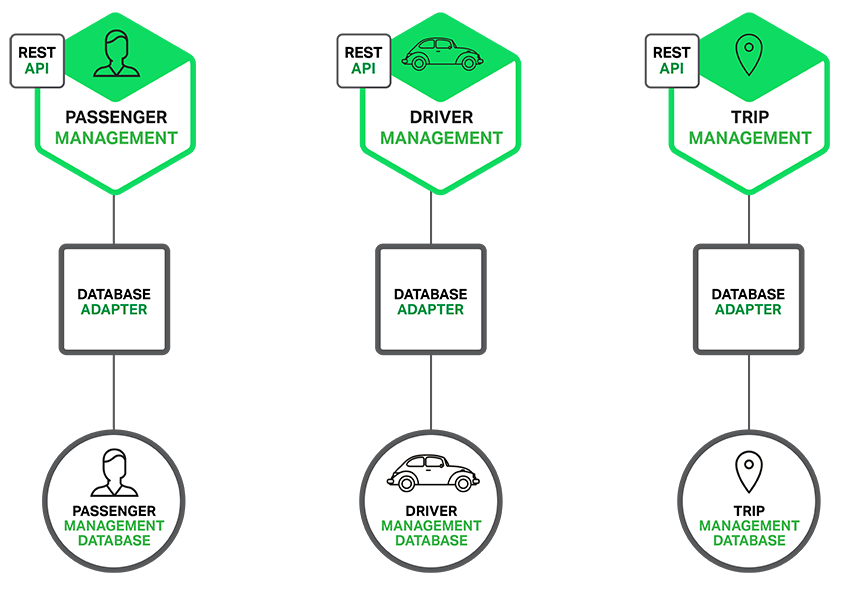
Applications typically use the three types of scaling together. Y‑axis scaling decomposes the application into microservices as shown above in the first figure in this section. At runtime, X‑axis scaling runs multiple instances of each service behind a load balancer for throughput and availability. Some 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.



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](http://www.nginx.com/solutions/load-balancing/) that distributes requests across the instances. The load balancer might also handle other concerns such as [caching](http://www.nginx.com/resources/admin-guide/content-caching/), [access control](http://www.nginx.com/resources/admin-guide/restricting-access/), [API metering](http://www.nginx.com/solutions/api-gateway/), and [monitoring](http://www.nginx.com/products/live-activity-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.



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**](http://en.wikipedia.org/wiki/List_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

 it tackles the problem of complexity

It decomposes what would otherwise be a monstrous monolithic application into a set of 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.

**https://www.nginx.com/blog/introduction-to-microservices/**