A Quick Guide to

Microservice Architecture



Why this emerging trend is the future of software development

MENLO TECHNOLOGIES

Why Choose a Microservice Architecture?

As the approach to software development evolves, the industry as a whole and the technical leaders and engineers in the industry constantly look for areas that could benefit from continuous improvement.

- How do we get solutions to customers faster?
- How do we make changes quicker?
- How do we reduce the risk of regression defects during a release?
- How do we isolate performance issues?

There is an emerging trend in the industry that serves to answer these very questions and more. It's called a Microservices Architecture.

Briefly, the microservice architecture is a way of designing software apps as a suite of independent deployable services that can scale and grow as needed.

Most large scale web sites including Netflix, Amazon and eBay have evolved from a monolithic architecture to a microservices architecture.

To understand microservice architecture, we should compare it to the monolithic approach - a monolithic application that is typically built as a single unit. Enterprise Applications in general have following components:

1. Presentation layer (client side UI) with HTML pages, JavaScript.
2. Business logic/server side layer – handles application logic, handle request/responses to presentation layer, and DB calls.
3. Database

These applications are developed in a Waterfall or Agile approach, by many or a few engineers. An e-commerce application may have an architecture like the one represented in the diagram.

The system may be architected with modularity in mind, but code is stored in a single code base When it comes to packaging and deploying, you create one or a few huge executables. This deployment of large files exists even with minor changes. Although there are benefits, the downfalls have led to this newer microservice architecture.

Benefits of monolithic architecture

- Easy to test with automated test scripts using tools like Selenium.
- Easy to deploy in the early stages of a project.
- You can scale the application by running multiple copies behind a load balancer.
- The modular approach allows for code clarity and less abstraction.
- By keeping an existing architecture, you don't have the cost of redesign and re-architecturing.

These factors apply well in early stages of application. But successful apps grow over time, become complex and eventually bloat into multi-million lines of code. At this stage, any change to the app or any bug fixes will become increasingly difficult, time consuming, and risky.

Once an application grows and becomes large, the monolithic approach will run into significant problems:

- **Complex code base** – Typically development slows down as the code base grows in size and complexity, with several developers working on the application. This tends to have a big impact on the quality of the application.

- **Complex monolithic apps become an obstacle for continuous deployment.** In today's "get new features out quickly" approach, frequent code deployments are the norm. This is very difficult with a complex monolithic app since the entire app has to be deployed even for a small change. Also to maintain quality, extensive testing is needed. This makes continuous deployment next to impossible.

- **Monolithic apps can be difficult to scale.** For example, some modules might be CPU-intensive and some might need more memory, say for caching. As all the modules are deployed together, there can be compromises on the choice of hardware and the cost of the hardware.

- **As applications become large and complex we can run into stability issues**. For example, a memory leak in a module can potentially bring down the whole application.

- **Monolithic apps make it extremely difficult to adopt new technologies/frameworks**. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer framework, even if that framework is 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.

"A Monolithic architecture only makes sense for simple, lightweight applications."
Chris Richardson, software architect

Evolution out of Monolithic Application Development

With the advent of service-oriented architectures (SOA) and web services in the cloud, the industry began to steer towards what is now Microservices. But this initial SOA methodology had flaws as well and didn't take hold in the way the industry prognosticators predicted.

The notion that you could architect away from large single code bases was solved, but the Enterprise Service Bus (ESB) didn't really solve the performance issues. Moreover, firms that adopted SOA often needed to buy-in with a vendor for the ESB services.

Firms would typically buy a large vendor-specific solution , and it would be cost prohibitive to architect out of that solution. Great for the vendors, not so great for the engineering teams.

In addition, the SOA environment itself required new skill sets to manage the environment and didn't scale well horizontally to handle performance issues. And it certainly didn't scale on demand.

The Road leads to Microservices

The SOA approach was a step in the right direction and paved the way for microservices. There is another major factor that contributed to the evolution into this newer framework – that is the advent of cloud computing which allows for compute servers and containers to be fired up quickly, on demand, and then turned down when not needed.

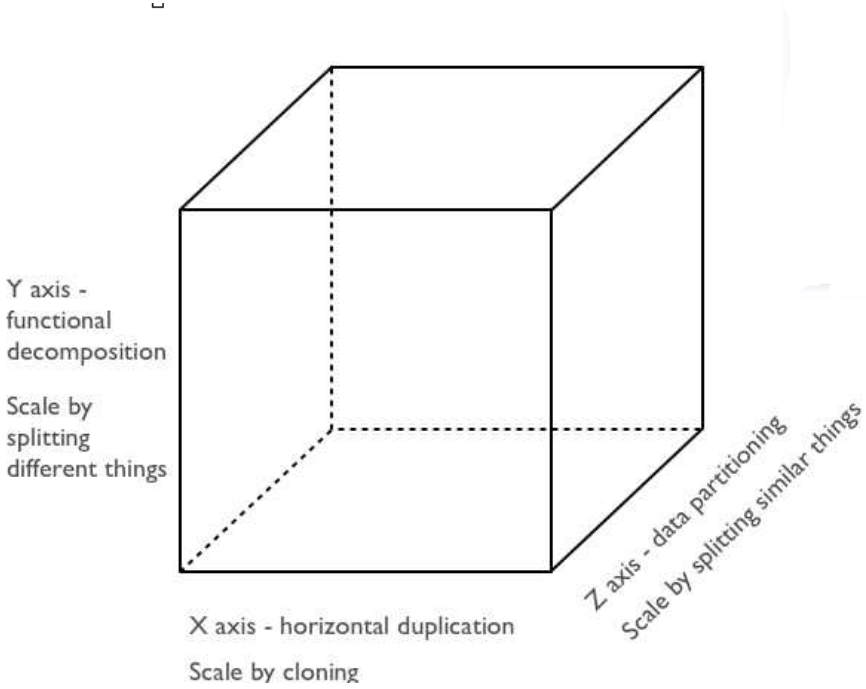
Both virtual machines in the cloud and compute services, including Azure Functions and AWS Lambda, provided the infrastructure needed upon which these microservices can thrive.

The issues described here can be addressed with a microservices architecture.

Instead of building a single huge, complex app, we can split it into a suite of services which are independently deployable and scalable. A service then implements specific and distinct functionality and has firm boundaries. Each service can have its own architecture and handle business logic and adapters to implement the desired features. A service can publish APIs to be consumed by other services.

Microservices can publish updates and subscribe to updates as necessary.

The Microservices Architecture pattern corresponds to the Y-axis scaling of Scale Cube, which is a 3D model of scalability from the book [The Art of Scalability](), which was written by industry experts and application architects Martin L. Abbott and Michael T. Fisher.



The two readily known scaling dimensions 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.

Y-axis scaling decomposes the application into Microservices, providing a higher level of scalability by splitting workloads into different functional components. 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.

Why is microservices an emerging trend?

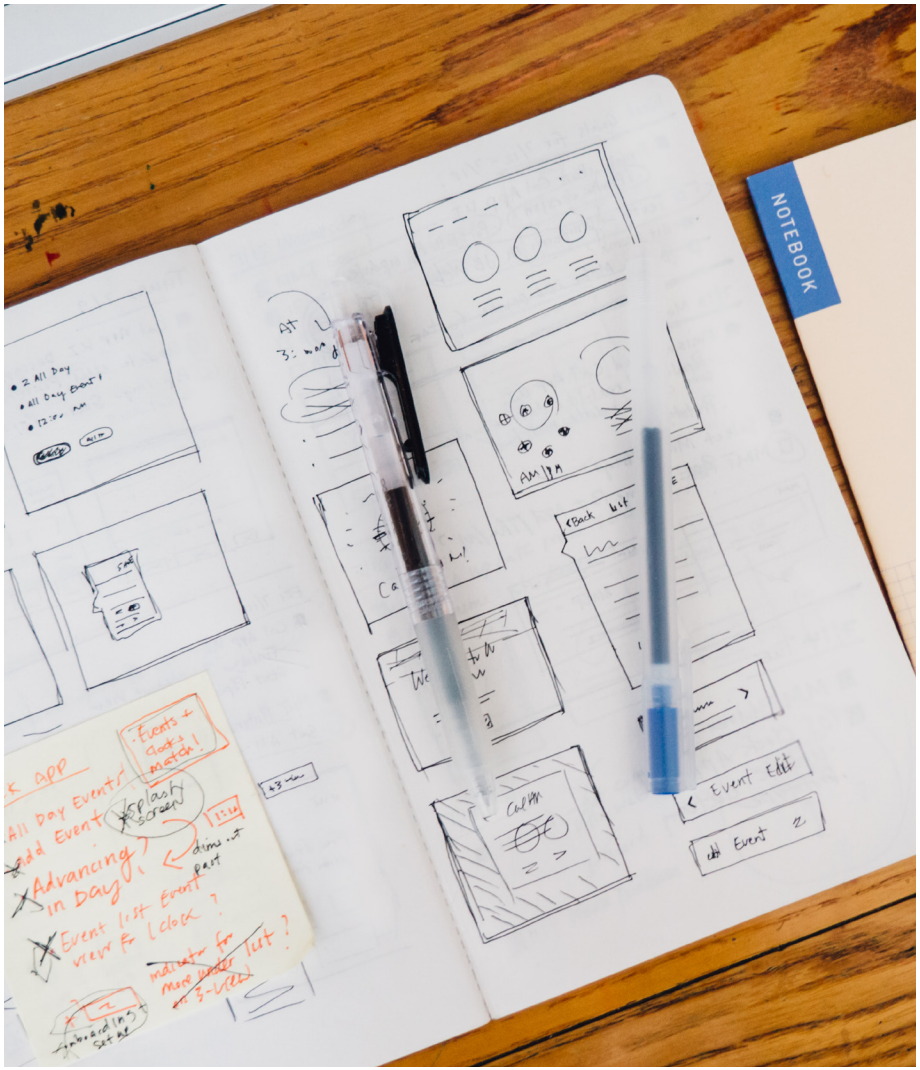
Microservices exists because of the need to deploy changes quickly, reliably, on an infrastructure with the lowest cost. It is an answer to the competitive demands for new features and better performance from users and businesses alike.

A microservice architecture contains the following:

- The services are easy to deploy
- The services are organized around functional capabilities
- The services can be implemented with different languages, database, and hardware
- Services are independently deployable and fully encapsulated
- Lends itself to continuous delivery
- Enhancements and fixes can be isolated to just the service that requires the modification, thus reducing regression risk.

Some of the benefits are naturally inherent in the characteristics, however other advantages to this architecture include:

- Smaller individual code bases don't slow down IDE's the way large monolithic code bases can. Programmers feel more productive with snappier responsiveness on their IDE's
- You can differentiate the responsibilities of your dev team around the functional services and changes can be deployed independently
- Services can be scaled easier and independently with cloning and partitioning one a single service if necessary – maybe to handle peak loads
- Memory leaks or defects will less likely disable the entire system, but rather be isolated to the service in question
- Because the services are small in nature, it is easier to adopt new technologies and languages because you don't need to do a rip and replace on a large code base.



Microservices Architecture Drawbacks

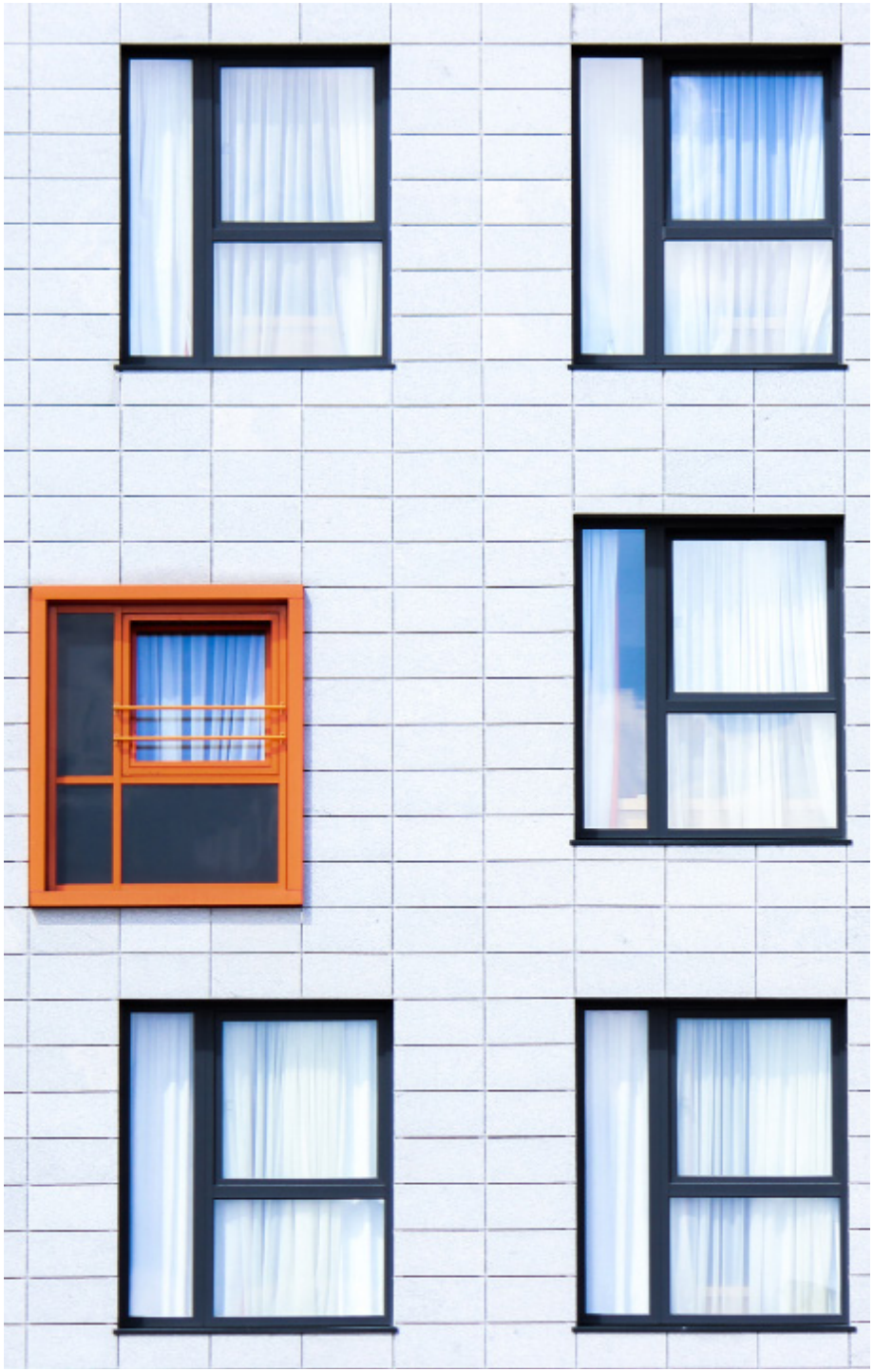
As with any architecture, there are drawbacks to this approach.

Drawbacks include the various components of these services and the management of the volume and dependencies. It can be tricky to get your arms around them and architect it effectively. Some of your effort must be in inter-process communication and ensuring performance bottlenecks do not occur.

Additionally, testing and building test scripts can be challenging – especially at the unit test level. Deciding what and when to implement a microservice architecture can be a challenge. Typically, a new application may not have the volume nor complexities that warrant it.

However, waiting until later won't work either as developers can't retrofit a monolithic app into a microservices architecture very easily.

"Microservices are the first post DevOps revolution architecture.
Neal Ford, Application Architect



Menlo Technologies Approach to Microservice Architecture

At Menlo Technologies, we typically wouldn't recommend 100% microservices any more than we would support a 100% Agile approach. In order to achieve the benefits of microservices and not experience the drawbacks, we support a hybrid approach. Let's discuss this briefly by way of an example.

Let's assume you are asked to build a new mobile and web app that has registration and location services. The location services may include GPS and mapping as well as multiple reads and writes from thousands or millions of users.

Let's also assume that the registration process is isolated and occurs once for a user with minimal updates. With this scenario, we would build the registration with a single code base but expose API's to support mobile and web registration.

However, for the location services, we would analyze and divide every feature into its tiniest components and microservices. We would categorize the services and log performance data continuously in order to allow for automatic scaling depending on volume and performance. In this manner you have isolated the microservices into the features that really need it.

If your application could benefit from Microservices, now is the time to architect it into your environment. Start small and grow as you gain knowledge. If the history of software development evolution is any indicator, you can bet that this architecture will morph and improve over time. You'll need to get on the road first though in order to take advantage and continuously improve.

There you have it. A brief overview of Microservice Architecture.

Stay tuned for more web architecture resources from
Menlo Technologies as we explore more examples and discuss API management
and inter-process communication for a Microservice Architecture.

Menlo Technologies
www.menlo-technologies.com
CONTACT US