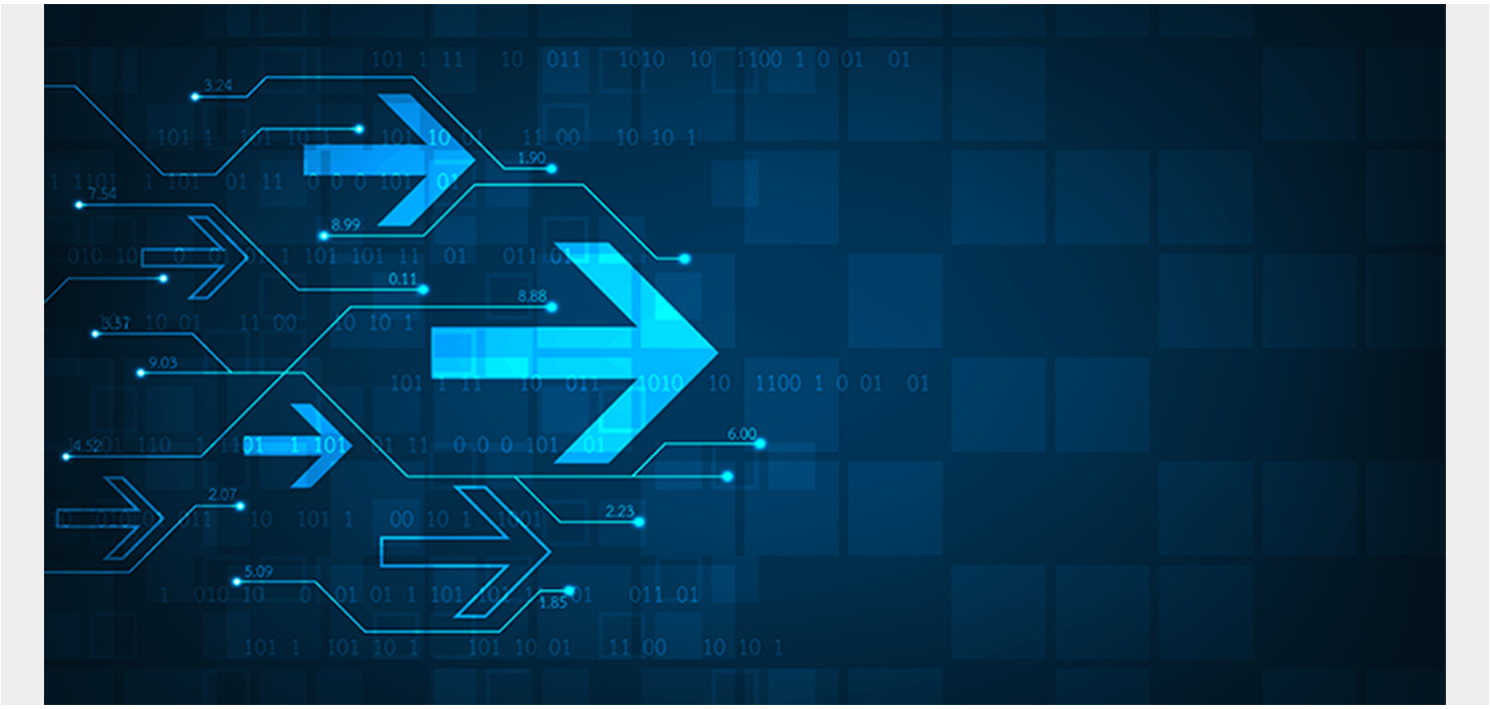
15 BEST PRACTICES FOR BUILDING A MICROSERVICES ARCHITECTURE

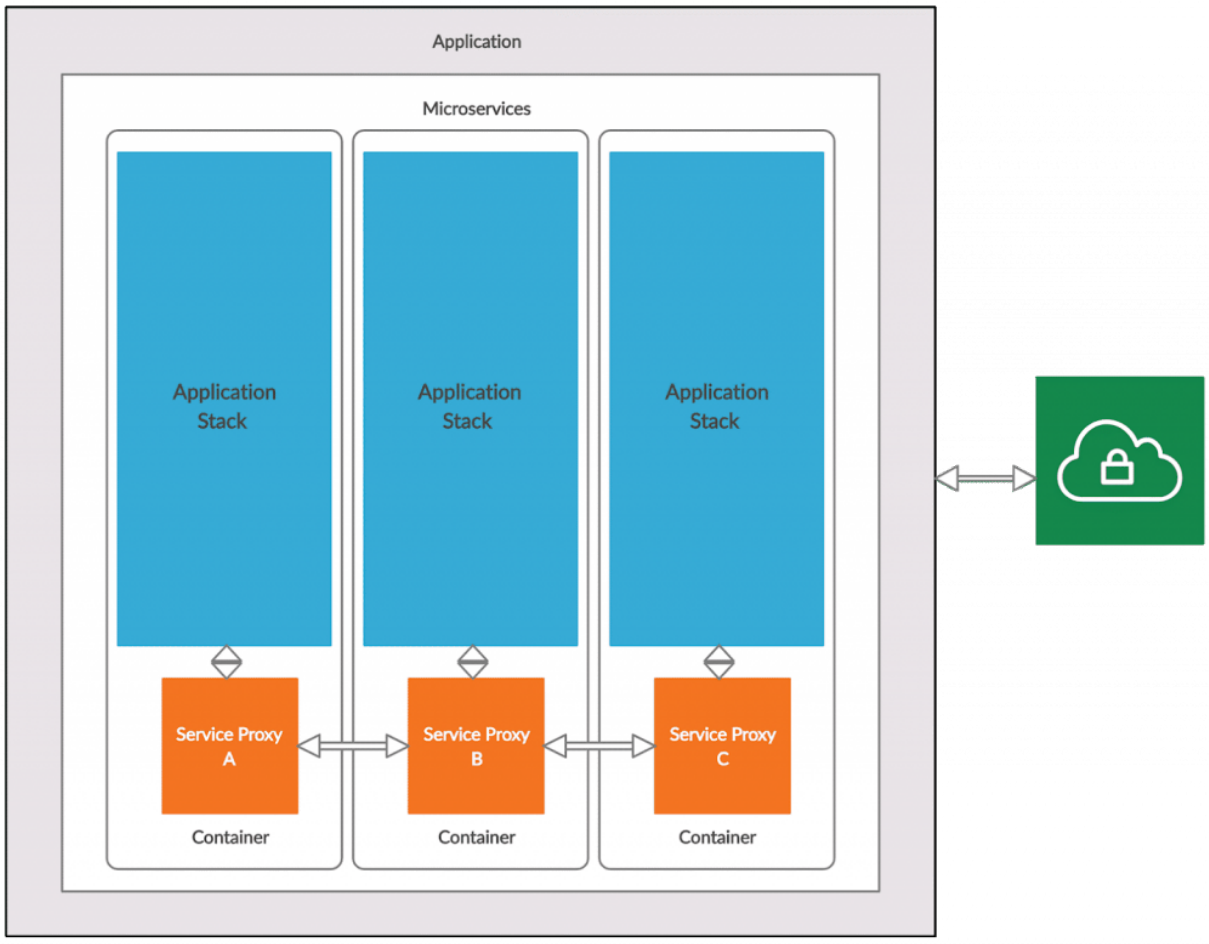


Microservices are an evolved architectural pattern that involves the design and development of an application as a collection of small, autonomous, loosely coupled services that communicate with each other.

At its elemental level, each individual microservice acts as an application in itself. Structuring applications as a collection of microservices encourages:

- Easy and faster deployment
- Scalability
- Maintainability

That makes managing and maintaining the application as a whole a lot easier.



A typical microservices framework

These advantages give credence to the wider adoption of microservices by DevOps organizations both large and small. It's a safe bet that microservices adoption will continue to rise thanks to these advantages—but to maximize the benefit, it's crucial that you embrace best practices for running resilient distributed systems.

Here are 15 best practices that are fundamental to microservice success. Let's take a look!

Best practices for microservices architecture

In this article, I group those best practices into categories that reflect the progression of various software development lifecycle (SDLC) phases, starting from the time you begin considering microservices for your next project, to the end of application deployment.

The tips also contain practices specific to new adopters, so you can transition successfully from a monolithic to a microservice framework.

For planning & organizing...

- **Determine if the microservices architecture is a good fit based on your requirements.** Do not adopt a microservices architecture because the big names are doing it. You should analyze your requirements to see where you can segment them into functions that provide value. Do your due diligence to ensure that your application can be sub-divided into microservices while still retaining its core features and operability.

- **Get everyone on board with the idea.** The transition from monolithic architecture to microservices is initially a lengthy and tedious process—and the impact is not limited only to the development team. Stakeholders should consider the time, money, and technical expertise required for the infrastructural changes that must be adopted. The engineering team specifically faces a considerable disruption being both the *implementer* and the *user* throughout the transition.
- **Build teams around microservices.** Each microservice acts as an independent application in itself, so teams should, too. Form separate teams for handling different microservices. This also requires that such teams have the necessary skills and tools to develop, deploy, and manage a specific service on their own. The teams should be versatile and big enough to handle their operations autonomously without wasting time communicating.

When designing the microservice...

- **Differentiate your microservices from your business functions and services.** Doing this will help you avoid building microservices that are either too large or too small. If the former occurs, you will see no benefits from using the microservice architecture. The latter will lead to an exponential increase in operational costs that outweighs any benefits gained.
- **Design your services to be loosely coupled, have high cohesion, and cover a single bounded context.** A loosely coupled service depends minimally on other services. Having high cohesion requires that the design of the service should follow the single responsibility principle—that is, it should perform only one main function and do it well. Lastly, design your service such that they are domain-specific while containing internal details of the domain and domain-specific models. This ensures that a microservice covers a single-bounded context, achieving a Domain-Driven Design (DDD).
- **Use APIs and events to communicate between services.** Services should not call each other directly. Instead, design an API gateway that handles authentication, request, and responses as well as throttling for the services. With an API gateway in place, you can easily redirect traffic from the API gateway to the updated version whenever there are changes to your service.
- **Consider security vulnerabilities within a microservice architecture.** As a best practice, adopt the DevSecOps model to ensure a secured microservices framework. Importantly, microservices in general are more susceptible to attack vectors due to their distributed structure. As a result, provisioning security requires a completely different approach when compared to a monolithic framework—that's why DevSecOps is perfect.

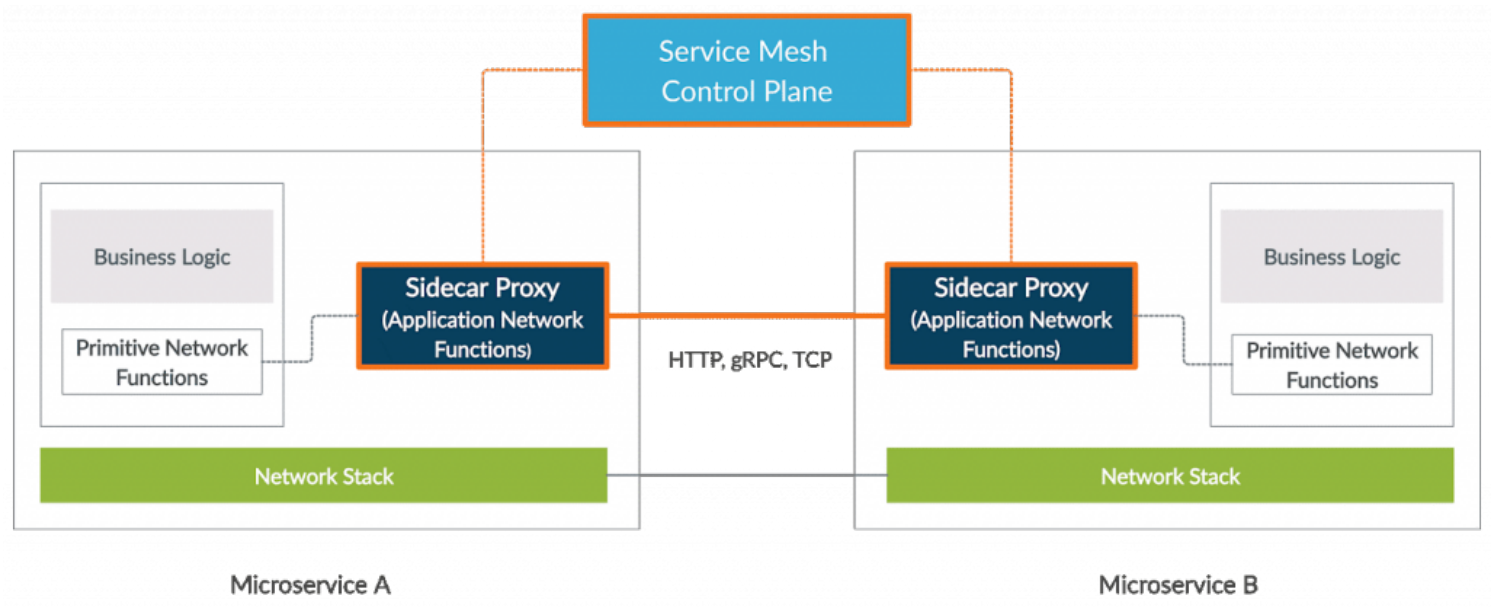
When developing the microservice...

- **Have a separate version control strategy for each service.** Each service should have its repository for ease of access provisioning while keeping version control logs clean. This also comes in handy if you are implementing any change that could potentially break other services.
- **Development environments should be consistent across machines.** Set up the development environment of your service as virtual machines to enable developers to adapt the framework and get started quickly.
- **Include backward compatibility for the service endpoints you are exposing.** You do not want to break any callers. To do so, implement rigid contract tests to protect against breaking changes. This also enables backward compatibility based on the API calls in response to every

user query. Doing this helps your organization build production-ready applications faster.

For data storage & management...

- **Have different databases or data stores for each microservice.** Select a database that suits the needs of your microservice, customize the infrastructure and storage to match the data it will contain, and use it solely for that microservice. This remains one of the key enablers of achieving a robust microservice framework, where each service is maintained independently, while working in cohesion with other services through a service mesh.



Service mesh-enabled microservices

For deploying & hosting...

- **Deploy your microservices separately.** Doing so helps save time while coordinating with multiple teams during regular maintenance or upgrade efforts. You also do not want a single service to use an unfair amount of resources while impacting other services, just because they're sharing resources. I highly recommended using a dedicated infrastructure to host each microservice. Doing this isolates each microservice from faults in other components, thus helping in fault tolerance and avoiding a full-blown outage.
- **Containerize your microservices.** Containers and microservices often go hand-in-hand. With containerized microservices, you can deploy and management individual services independently, without affecting services hosted on other containers. Containers also offer platform-independence and interoperability that blend exactly with the goals of microservice architecture.
- **Have a separate build for your microservices and automate the deployment process.** An essential aspect of realizing the DevOps model is to enhance efficiency by enabling automation. With automation tools like Jenkins, you can automate DevOps workflows by facilitating Continuous Integration and Continuous Delivery (CI/CD).

For maintenance & operations...

- **Use a centralized logging and monitoring system.** A centralized logging system ensures that all microservices ship their logs in a standardized format, though saves logs discreetly for each of them. As opposed to a monolithic model, this aids in faster error handling and root cause analysis. In addition, using an advanced monitoring solution not only helps monitor resource-availability, but also maintains security by identifying compromised resources early.

Microservice architecture transition

Microservices help manage applications better, but the transition to a microservices architecture is complex. The approach of implementing microservices may differ for different use-cases—but these fundamental best practices are universal.

As always, the goal for all microservices is to achieve a framework that is loosely coupled, distributed, and independent while constituting a DevOps model that enables automation and efficiency.

Related reading

- BMC DevOps Blog
- Microservices vs SOA: How Are They Different?
- The Death (or Not) of Microservices
- Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools
- PostgreSQL & K8s: Run a Stateful Legacy App on a Stateless Microservice, part of our Kubernetes Guide
- How & Why To Become a Software Factory
- Managing IT as a Product—Not a Project