MICROSERVICES VS MINISERVICES: CHOOSING THE RIGHT FRAMEWORK



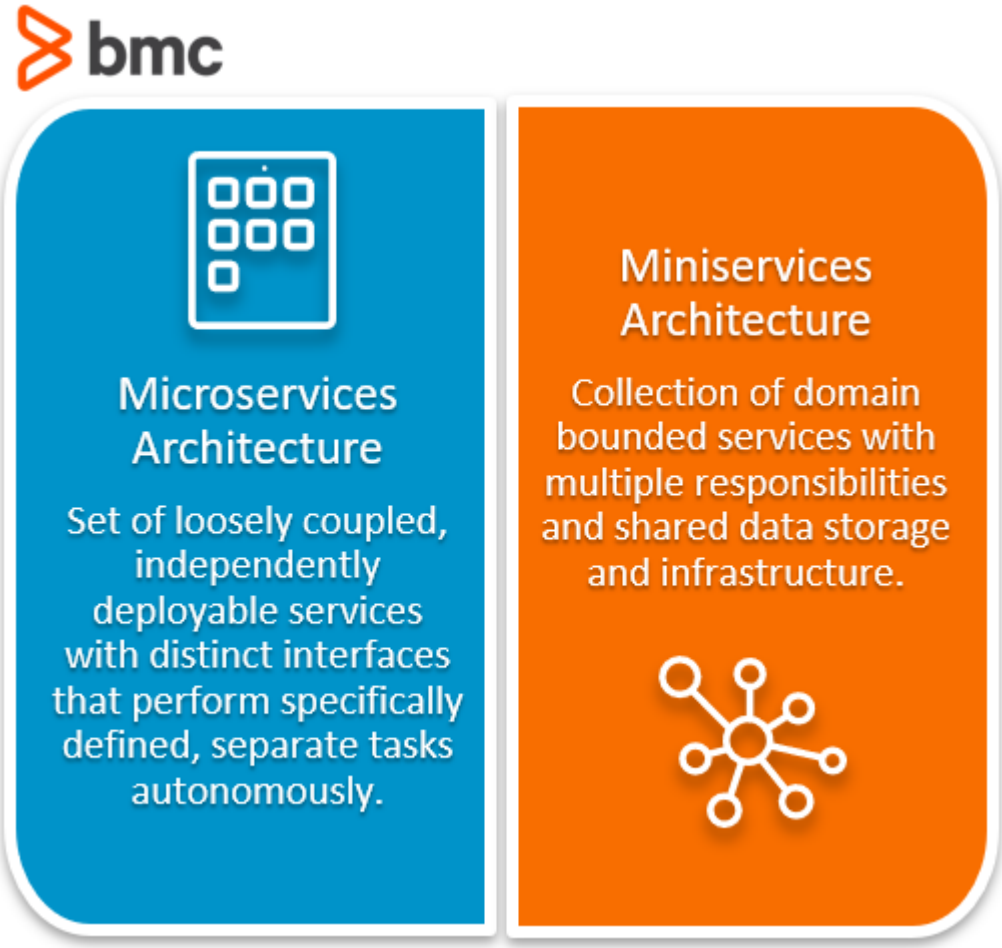
The homogenous nature of a monolithic architecture has both strengths and challenges:

- Because monolithic applications have all business services and functions, including their supporting databases, deployed as a single platform, software development and deployment are relatively faster and easier.
- Debugging is also more straightforward because you can open up the entire project within a single IDE instance.

However, with those benefits, monoliths are complex to maintain, refactor, and scale.

Over the years, several architectural patterns evolved out of monoliths, aiming to address these challenges by separating business functions into individually deployable services. Two such evolved patterns are the microservices and miniservices architectures.

This article compares these two architectures and the benefits they offer as alternatives to a monolith.



Microservices architecture

A microservices architecture follows a development approach that designs software applications as a set of loosely coupled, independently deployable services with distinct interfaces. These individual functional modules perform specifically defined, separate tasks autonomously.

Characteristics of a microservice architecture

A microservice-based framework is a collection of autonomous microservices designed around specific business capabilities. Essentially these services are miniature applications that function collectively to support the main application.

Essential attributes of a microservices design requires:

- **Each microservice contains one, and only one, responsibility**, which is built around a particular business function such as sending emails, raising alerts, assigning tickets, etc.
- **Every microservice has its own database** that does not share data storage with other services.
- **All services are developed, deployed, maintained, and run independently of other microservices.** Thus, each microservice has its own codebase and deployment environments.
- **Microservices are loosely coupled**, i.e., you can change one microservice without updating/impacting others.
- **All microservices communicate with each other via an event-driven communication** that runs a Publisher-Subscriber pattern.

(Explore 15 best practices for microservices.)

Benefits of using microservices

Adopting a microservice architecture brings a range of added benefits that aid efficiency to the software development lifecycle (SDLC). These benefits include:

- **Improved scalability.** Each microservice can be scaled independently of others, instead of scaling up the entire application framework. For instance, a specific service can be allocated additional resources to scale its efficiency. Such an ability to scale resources for specific services makes microservices more operationally efficient than monoliths.
- **Improved system resiliency.** An application consists of multiple, independent services. By design, when one service fails, the entire system remains fairly unimpacted. This allows the application to remain functional, while the right team works on the affected service.
- **Better fault isolation.** The loosely coupled nature of microservices makes it easier to find and isolate faults of a particular service and fix them, reducing resolution times
- **Enhanced maintainability.** It is easier to maintain a microservices-based application because each service can be maintained, optimized, or enhanced for better performance without impacting other services.
- **Bit-sized quicker deployments.** Each service has its own codebase running in individual containers. This enables quick development and deployment cycles that follow an efficient DevOps model.
- **Flexibility in choosing a technology stack.** Developers are not boxed in by particular programming languages or libraries. This means teams have the freedom of choosing whatever language and libraries that are most appropriate for implementing a service. More so, every service is designed to run its own technologies that may be different than technologies used by other services.

Limitations of microservices

Although a microservice architecture is gaining popularity due to its benefits of enhanced efficiency and improved resiliency, it also comes with its limitations and challenges.

- **Increased complexity.** Having a collection of polyglot services introduces a higher level of complexity into the development process. There are more components to manage, and these components have different deployment processes. The introduction of event-driven communication is another challenge: such design is comparatively complex and requires new skills to manage.
- **Complex testing.** It is challenging to test microservice-based applications because of the various testing dependencies required for each microservice. It is even more tasking to implement automated testing in a microservice architecture because the services are running in different runtime environments. Besides, the need to test each microservice before running a global test adds more complexity to maintain the framework.
- **Higher maintenance overhead.** With more services to handle, you have additional resources to manage, monitor, and maintain. There is also the need for full-time security support: due to their distributed nature, microservices are more vulnerable to attack vectors.

(See when microservices might not be the best fit for your software.)

Now we'll turn to miniservices.

Miniservices architecture

As monoliths are challenging to scale because of size, and microservices are a lot more complex to orchestrate and maintain, there was a need for a framework that addressed these challenges.

To solve this, a miniservices architecture fits the middle ground between monolith and microservices architectures, a design that assumes a more realistic approach to implementing the microservices concept.

The miniservices architecture is an architectural framework that has a collection of domain bounded services with multiple responsibilities and shared data stores. Unlike microservices with a complete de-coupling of services and their implementation details, miniservices can share libraries and databases.

Characteristics of a miniservice architecture

- **Related services can share the same database.** This allows modules that are related to each other in the functions they perform to share a database. For instance, a miniservice may perform multiple functions including image processing, rendering of images, or any other related functions for an application.
- **Communication between services is through REST APIs.**
- **Related services can share codebase and infrastructure** used for deployment.

Benefits of using miniservices

Due to its derived design, miniservices inherit all benefits of a microservice architecture including scalability, fault tolerance, and robustness.

Additionally, other benefits of adopting miniservices include:

- **Improved performance.** By reducing the number of services, interconnections, and network traffic between domains, miniservices enhance application performance
- **Shared maintenance overhead.** With services handling various related functions, the maintenance overhead associated with microservices is reduced.
- **Developer friendly.** Miniservices are often more suitable for companies that cannot afford to create smaller development teams dedicated to working on each individual service.

Limitations of miniservices

- End-to-end testing can be a challenge with a miniservice framework due to the number of dependencies associated with a single service. This also raises complexities with respect to efficient error handling and bug discovery.

Microservices vs miniservices

Fine-grained alternatives to a monolithic framework, both microservice and miniservice architectures divide applications into smaller pieces within specific bounded contexts.

At its elemental level, miniservices differ from microservices by allowing shared data storage and infrastructure. Miniservices are steadily gaining momentum as a more pragmatic approach over

microservices.

As each of these architectures has its benefits and limitations, it's vital that your organization perform thorough due diligence before choosing the right one. It is equally important to factor in the technologies, skills, and effort each of these frameworks require to maintain in the long run to avoid budget overruns and operational hiccups.

Related reading

- BMC DevOps Blog
- APIs vs Microservices: What's The Difference?
- Microservices vs Serverless: Comparing Benefits & Differences
- Kubernetes Guide
- Top DevOps Trends Today