Microservices and microservices architecture

Learn the pros and cons of microservices and how they differ from monoliths.

Try it now

Not very long ago, the preferred method of building software applications was a monolithic architecture, which was as a single, autonomous unit. This approach worked well for many developers until applications increased in complexity. When a modification to a small section of code is made in a monolithic system, it requires rebuilding the entire system, running tests on the entire system, and deploying an entirely new version of the application.

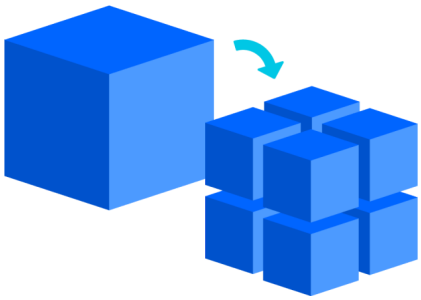
Then along came microservices, an approach that breaks down software systems into smaller units that are autonomously developed and deployed. The microservices architecture was driven by the DevOps movement that seeks to frequently deliver updates, like new features, bug fixes, and security improvements. It also became, in many instances, a path for businesses to rewrite legacy applications using modern programming languages and updates to a technology stack.

Microservices are a collection of small units that continuously deliver and

deploy large, complex applications.

What are microservices?

A microservices architecture, also simply known as “microservices”, is an approach to building an application as a series of independently deployable services that are decentralized and autonomously developed. These services are loosely coupled, independently deployable, and easily maintainable. While a monolithic application is built as a single, indivisible unit, microservices break that unit down into a collection of independent units that contribute to the larger whole. Microservices are an integral part of DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements.



A microservice is a web service that is responsible for one piece of domain logic. Multiple microservices combine to create an application, with each providing a piece of functionality for a domain. Microservices interact with each other using APIs, like REST or gRPC, but don’t have knowledge of the internal workings of the other services. This harmonious interaction between microservices is a microservice architecture.

With a microservices architecture, developers can organize in smaller teams specializing in different services, with different stacks and decoupled deployments. For example, Jira is powered by multiple microservices, each providing specific capabilities, including searching issues, viewing issue details, comments, issue transitions, and more.

Characteristics of microservices

There is no formal definition of a microservices architecture, but there are some common patterns or characteristics that are important to know.

Autonomous components

The basic building block of a microservices architecture is a component, which can be a package of software, web service or resource, application, or a module that contains a series of related functions. Or, more simply explained by Martin Fowler: "software components are things that are independently replaceable and upgradeable."

> In a microservices architecture, each component can be developed, deployed, operated, changed, and redeployed without compromising the function of other services or an application's integrity.

Components are used with other components to deliver a customer experience or business value. The most common are services and libraries but can encapsulate CLI tools, mobile apps, front-end modules, data pipelines, machine learning models, and many other concepts that can be applied within a microservices architecture.

Clean interfaces

Once individual components are established, they require a substantial amount of logic to communicate with each other through a communication mechanism like RPC, REST over HTTP, or an event-based system. These mechanisms may be synchronous or asynchronous methods and microservices may use a combination of both.

The most important aspect is that each microservice should provide a clear, intuitive contract that describes how a consumer of that service can use it. This is typically done through an API published alongside the service.

You build it, you run it

The DevOps philosophy of "you build it, you run it" emphasized that you cannot change an architecture into microservices without thinking about how teams are structured. DevOps re-aligns incentives across functional roles – development, QA, release engineering, operations – into a single team with a shared goal of building high-quality software. DevOps practices like CI/CD, automated testing, and feature flags accelerate deployment and help maintain system stability and security. Plus, each team can build and deploy microservices that it owns without affecting other teams.

The evolution of the cloud simplified the building, deploying, and operating of microservices. Infrastructure automation techniques like continuous integration, continuous delivery, and automated tests help teams.

Service-oriented architecture vs. microservices

Service-oriented architecture (SOA) and microservices are two types of web service architectures. Like microservices, an SOA comprises reusable, specialized components that work independently of each other. The distinction between the two architecture types is the bureaucratic classification of service types.

An SOA has four basic service types:

- Business
- Enterprise
- Application
- Infrastructure services

These types define the related domain-specific responsibility of the underlying services. Comparatively, microservices only have two service types: functional and infrastructure.

Both architectures share the same set of standards at different layers of an enterprise. The existence of a microservices architecture comes down to the success of an SOA pattern. Hence, a microservices architecture pattern is a subset of SOA. Here, the main focus is on the runtime autonomy of each service.

Benefits of microservices

Agility

Since small, independent teams typically build out a service within microservices, it encourages teams to adopt agile practices. Teams are empowered to work independently and move quickly, which shortens development cycle times.

Flexible scaling

Since microservices are distributed by design and can be deployed in clusters, they enable dynamic horizontal scaling across service boundaries. If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure.

Frequent releases

One of the primary advantages of microservices is frequent and faster release cycles. As a key element of continuous integration and continuous delivery (CI/CD), microservices allow teams to experiment with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time-to-market for new features.

Technology flexibility

Microservice architectures don't necessarily follow a set approach with one toolchain but allow teams the freedom to select the tools they desire.

Focus on quality

The separation of business concerns into independent microservices means that the service team owning that service focuses on the completed quality deliverable.

High reliability

By separating functionality, microservices reduce the risk of breaking an entire application or code base when releasing updates. It is easy to isolate and fix faults and bugs in individual services. You can deploy changes only for a specific service, without bringing down the entire application, which provides higher reliability.

Challenges of microservices

Development sprawl

The move from a monolith to microservices means more complexity. There are more services in more places created by multiple teams. This makes it challenging to see how different components relate to each other, who owns a particular component, and how to avoid causing a negative impact on dependent components. If sprawl is left unmanaged, it results in slower development speed and poor operational performance. As a system grows, it requires an experienced operations team to manage constant redeployments and frequent changes in the architecture.

Lack of clear ownership

A microservices architecture adds confusion to who owns what. A DevOps team may run a combination of APIs, component libraries, monitoring tools, and Docker images for users to deploy an application. It's important to have insight into information about components, including their owners, resources, and evolving relationships between other components. Precise communication and coordination is required between numerous teams so that everyone involved can easily find the required knowledge to understand a product.

Exponential infrastructure costs

Each new microservice added to production deployment comes with its own cost of test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.

Added organizational overhead

An added level of communication and collaboration is needed to coordinate updates and interfaces between microservice architecture teams.

Debugging

It can be challenging to debug an application that contains multiple microservices, each with its own set of logs. A single business process can run

across multiple machines at different times, which further compounds debugging.

Incident response

It's important to have microservice incident response intelligence that includes information such as who is using the microservice, where the microservice was deployed, how the microservice was deployed, and who to contact when things go wrong.

Microservices and DevOps: Two peas in a pod

Given the increase in complexity and dependencies of microservices, the DevOps practices of deployment, monitoring, and lifecycle automation are seen as integral to microservices architectures. That's why microservices are often considered the first step to embracing a DevOps culture, which enables:

- Automation
- Improved scalability
- Manageability
- Agility
- Faster delivery and deployment

Key technologies and tools for a microservices architecture

Containers, Docker, and Kubernetes

A container is simply the packaging of an application and all its dependencies, which allows it to be deployed easily and consistently. Because containers don't have the overhead of their own operating system, they are smaller and lighter-weight than traditional virtual machines. They can spin up and down more quickly, making them a perfect match for the smaller services found within microservices architectures.

With the proliferation of services and containers, orchestrating and managing large groups of containers is essential. Docker is a popular containerization platform and runtime that helps developers build, deploy and run containers. However, running and managing containers at scale is a challenge with Docker alone. Kubernetes and other solutions like Docker Swarm, Mesos, HashiCorp Nomad, and more help to address containerization at scale.

Containerization and the deployment of containers is a new pattern of distributed infrastructure. Docker and Kubernetes package a service into a complete container that can be rapidly deployed and discarded. These infrastructure tools are complementary to the microservices architecture. Microservices can be containerized, easily deployed, and managed using a container management system.

API gateways

The individual services in a microservices architecture communicate with each other through well-defined APIs. An API gateway acts as a reverse proxy by accepting API calls, collecting the servicers to fulfill them, and returning results. API gateways provide an abstraction serving a single API endpoint, even though APIs are serviced by multiple microservices. API gateways can also consolidate concerns like rate limiting, monitoring, authentication, authorization, and routing to the appropriate microservice.

Messaging and event streaming

Due to the distributed nature of microservices, teams need a means of sharing state changes and other events. Messaging systems communicate between microservices, allowing some microservices to process events as part of their primary interface. For example, changes to Confluence pages results in an event that triggers a reindex for search and notifications to those watching the page.

Logging and monitoring

Microservices make it difficult to identify an issue and resolve it across multiple services. It's important to have observability tools for logging, monitoring, and tracing. This helps with understanding how microservices behave, identifying potential problems, troubleshooting issues, and debugging failures.

Continuous integration/continuous delivery

One of the primary advantages of microservices is frequent and faster release cycles. As a key element of CI/CD, microservices allow teams to experiment

with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time to market for new features.

Developer portal

As distributed architectures increase in complexity, development teams can benefit from a tool that consolidates information about engineering output and team collaboration in one place. For example, Atlassian Compass was designed to help tame microservice sprawl by providing data and insights across DevOps toolchains.

Future of microservices

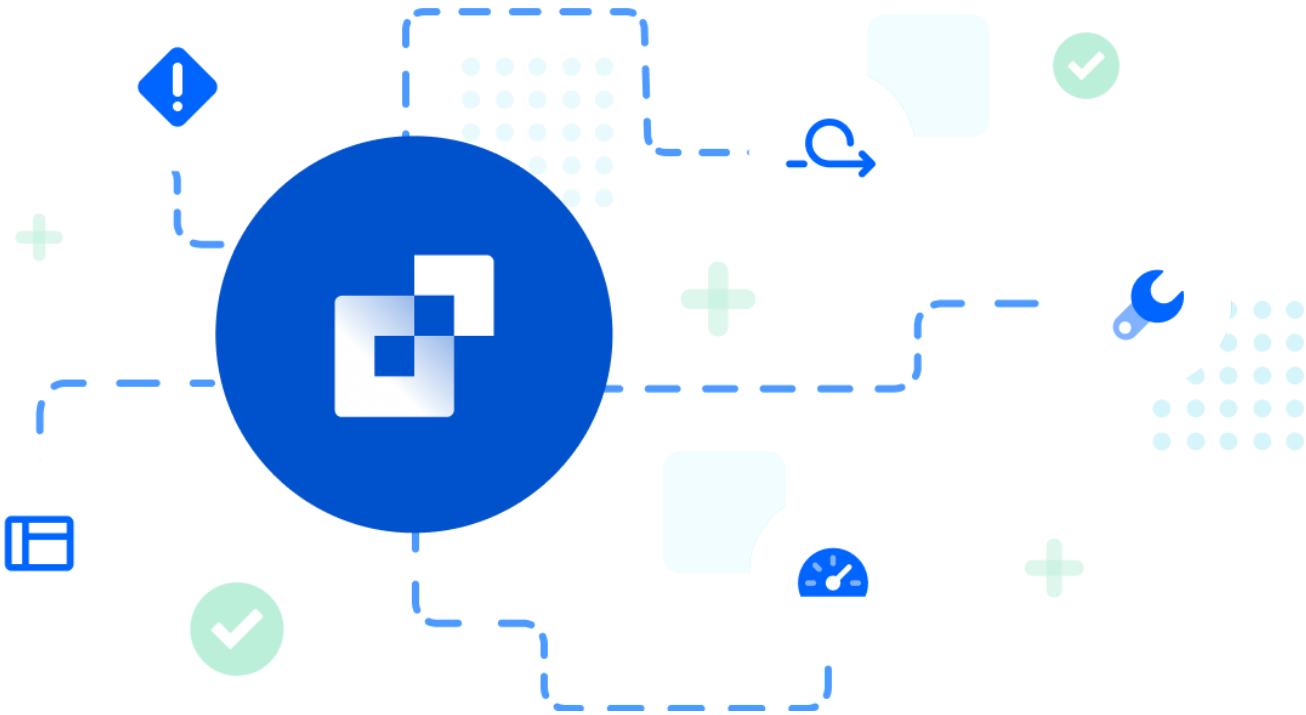
Containerization and the deployment of containers is now a common pattern of distributed infrastructure. Tools like Docker and Kubernetes package up a service into a complete container, which can be rapidly deployed and discarded. These new infrastructure tools are complementary to the microservices architecture, since microservices can be containerized, easily deployed, and managed using a container management system.

> The adoption of microservices should be seen as a journey rather than the immediate goal for a team.

It can be helpful to start small in order to understand the technical requirements of a distributed system and scale individual components. Then gradually extract more services as you gain experience and knowledge.

Navigate your microservices with Compass

When working with a microservices architecture, Atlassian's Compass manages the complexity of a distributed architecture that scales. It's an extensible developer experience platform that brings disconnected information about engineering output and team collaboration together in a central, searchable location. In addition to helping you tame your microservice sprawl with the Component Catalog, Compass can help you establish best practices and measure the health of your software with Scorecards and provide you with data and insights across your DevOps toolchain using extensions built on the Atlassian Forge platform.
Learn more about Compass →



ATLASSIAN

PRODUCTS

Jira Software
Jira Align
Jira Service Management
Jira Product Discovery
Confluence
Trello
Bitbucket

View all products

RESOURCES

Technical Support
Purchasing & licensing
Atlassian Community
Knowledge base
Marketplace
My Account

Create support ticket

EXPAND & LEARN

Partners
Training & Certification
Documentation
Developer Resources
Enterprise services

View all resources

ABOUT ATLASSIAN

Company
Careers
Events
Blogs
Atlassian Foundation
Investor Relations
Trust & Security

Contact us

English

Privacy policy
Notice at Collection
Terms
Impressum