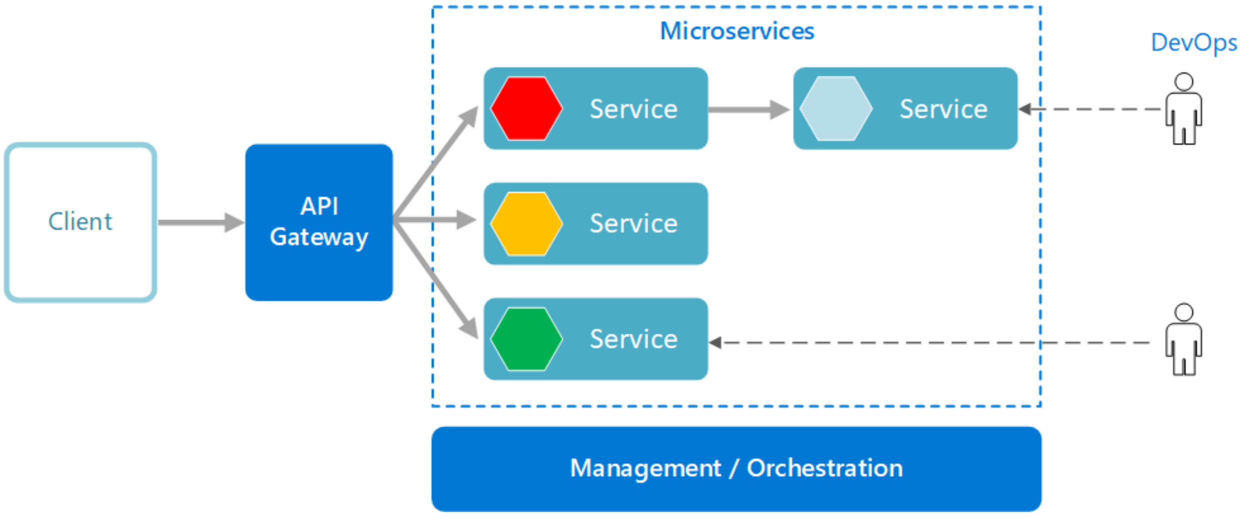
Microservice architecture style

Azure

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.



What are microservices?

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

Besides for the services themselves, some other components appear in a typical microservices architecture:

Management/orchestration. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

API Gateway. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features might be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams**. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base**. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.

- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Fault isolation**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly. For example, you can implement the Circuit Breaker pattern, or you can design your solution so that the microservices communicate with each other using asynchronous messaging patterns.
- **Scalability**. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application might all touch the same data, making any alterations to the schema risky.

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- **Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance**. The decentralized approach to building microservices has advantages, but it can also lead to problems. You might end up with so many different languages and frameworks that the application becomes hard to maintain. It might be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load leveling.

- **Data integrity**. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

- **Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.

- **Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.

- **Skill set**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

Best practices

- Model services around the business domain.

- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.

- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.

- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.

- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.

- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.

- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic.

Otherwise, the gateway becomes a dependency and can cause coupling between services.

- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.

- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading. See Resiliency patterns and Designing reliable applications.

Next steps

For detailed guidance about building a microservices architecture on Azure, see Designing, building, and operating microservices on Azure.