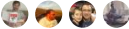


# Design for evolution

08/30/2018 • 3 minutes to read • Contributors 

## In this article

[An evolutionary design is key for continuous innovation](#)

[Recommendations](#)

## An evolutionary design is key for continuous innovation

All successful applications change over time, whether to fix bugs, add new features, bring in new technologies, or make existing systems more scalable and resilient. If all the parts of an application are tightly coupled, it becomes very hard to introduce changes into the system. A change in one part of the application may break another part, or cause changes to ripple through the entire codebase.

This problem is not limited to monolithic applications. An application can be decomposed into services, but still exhibit the sort of tight coupling that leaves the system rigid and brittle. But when services are designed to evolve, teams can innovate and continuously deliver new features.

Microservices are becoming a popular way to achieve an evolutionary design, because they address many of the considerations listed here.

## Recommendations

**Enforce high cohesion and loose coupling.** A service is *cohesive* if it provides functionality that logically belongs together. Services are *loosely coupled* if you can change one service without changing the other. High cohesion generally means that changes in one function will require changes in other related functions. If you find that updating a service requires coordinated updates to other services, it may be a sign that your services are not cohesive. One of the goals of domain-driven design (DDD) is to identify those boundaries.

**Encapsulate domain knowledge.** When a client consumes a service, the responsibility for enforcing the business rules of the domain should not fall on the client. Instead, the service should encapsulate all of the domain knowledge that falls under its responsibility. Otherwise, every client has to enforce the business rules, and you end up with domain knowledge spread across different parts of the application.

**Use asynchronous messaging.** Asynchronous messaging is a way to decouple the message producer from the consumer. The producer does not depend on the consumer responding to the message or taking any particular action. With a pub/sub architecture, the producer may not even know who is consuming the message. New services can easily consume the messages without any modifications to the producer.

**Don't build domain knowledge into a gateway.** Gateways can be useful in a microservices architecture, for things like request routing, protocol translation, load balancing, or authentication. However, the gateway should be restricted to this sort of infrastructure functionality. It should not implement any domain knowledge, to avoid becoming a heavy dependency.

**Expose open interfaces.** Avoid creating custom translation layers that sit between services. Instead, a service should expose an API with a well-defined API contract. The API should be versioned, so that you can evolve the API while maintaining backward compatibility. That way, you can update a service without coordinating updates to all of the upstream services that depend on it. Public facing services should expose a RESTful API over HTTP. Backend services might use an RPC-style messaging protocol for performance reasons.

**Design and test against service contracts.** When services expose well-defined APIs, you can develop and test against those APIs. That way, you can develop and test an individual service without spinning up all of its dependent services. (Of course, you would still perform integration and load testing against the real services.)

**Abstract infrastructure away from domain logic.** Don't let domain logic get mixed up with infrastructure-related functionality, such as messaging or persistence. Otherwise, changes in the domain logic will require updates to the infrastructure layers and vice versa.

**Offload cross-cutting concerns to a separate service.** For example, if several services need to authenticate requests, you could move this functionality into its own service. Then you could evolve the authentication service — for example, by adding a new authentication flow — without touching any of the services that use it.

**Deploy services independently.** When the DevOps team can deploy a single service independently of other services in the application, updates can happen more quickly and safely. Bug fixes and new features can be rolled out at a more regular cadence. Design both the application and the release process to support independent updates.